ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Second Cycle Degree Programme in

Digital Humanities and Digital Knowledge

A design approach for sighted programmers developing accessible Web content

Final Dissertation in

Information Modeling and Web Technologies

Supervisor: Prof. Fabio Vitali

Presented by: Chantal Lengua

Co-supervisor: Dr. Vincenzo Rubano

Session II

2020 / 2021

Table of contents

Abstract **5**

Introduction **7**

Chapter 1

Accessibility standards **13**

1.1 WCAG 1.0: guidelines to accessibility 13

1.2 WCAG 2.0: better, clearer, stricter 15

1.3 Too dynamic to be accessible: Ajax 18

1.4 WAI-ARIA to the rescue 21

Chapter 2

Limitations of accessibility standards **27**

2.1 WAI guidelines: an online paper maze 27

2.2 Turning an eye blind: the inadequate issues coverage 30

2.3 The almighty dollar: a barrier to accessibility? 34

Chapter 3

Accessibility testing tools **37**

3.1 AChecker, Tenon, Lighthouse, WAVE, axe 37

3.2 Reliability is not an option 51

Chapter 4

Problems for the sighted developers **55**

4.1 Unhelpful eyes: the accessibility role reversal 55

4.2 Tricky tech: is HTML5 really accessible? 65

Chapter 5

SahARIAn **69**

5.1 SahARIAn: switching to blindsight 69

5.1.1 Browser extensions 69

5.1.2 SahARIAn browser extension 71

Chapter 6

SahARIAn architecture **81**

6.1 Deep down the rabbit hole: the architecture 81

6.2 Toggling SahARIAn without information leakage 86

6.3 The delicate handling of images 87

6.4 Accessible name and accessible description 90

6.5 In the blink of an eye: a dynamic visualization 93

6.6 SahARIAn interface widgets and page structure 96

6.6.1 SahARIAn Document Structure roles 97

6.6.2 SahARIAn Window roles 101

6.6.3 SahARIAn Live Regions roles 103

6.6.4 SahARIAn Widget roles 104

6.6.5 SahARIAn Landmark roles 109

Conclusion **113**

References **117**

Abstract

The digital revolution has reshaped the access to and consumption of information in a staggering and accelerating manner. From a simple text-based interface, the Web is now the theater of dynamic, flamboyant and explosive graphic applications. Consequently, aligning eye-catching design with accessibility requirements is an ever-increasing challenge for development teams. Over the years, even if standards, rules and principles have been published to provide support for this arduous task, we are still far from witnessing a fully accessible Web. The underlying causes of this phenomenon include not only the cost of implementation, the scarcity of awareness and the high level of required technical knowledge, but also the difficulty of assessing and perceiving the accessibility issues themselves. Existing testing tools and checkers provide a useful approach to detecting them, but their actual completeness and reliability rates remain low. This dissertation therefore discusses an innovative software, SahARIAn, that allows sighted developers to directly perceive accessibility issues as they arise, without the deployment of guidelines or heavy technical knowledge. SahARIAn, indeed, visually transforms the Web site relying only on accessible markup (WAI-ARIA by W3C) in such a way that non-disabled developers naturally sense the correctness or incorrectness of their page structure and interactive widgets.

Introduction

Few revolutions have spread as rapidly around the world as the digital one. From the latter half of the 20th century, access to information has begun to reshape society as we used to know it, affecting our economic, social, cultural, and human environments. Since then, improvements to the Web have made it increasingly easy to use: icons, point-and-click methods, queries first, e-payments, user-friendly designs and social media apps later, all users can now benefit from recent innovations and enhancements.

But is this true?

Not for all users. Not always. A significant portion of the population remains cut off from Web content: people with certain disabilities, particularly visual impairments, cannot access it or can only access some limited functionalities.

The World Health Organization (WHO) reports[1] that today more than 1 billion people live with a form of disability. This equates to almost 15% of the world's population, 3.8% of whom (190 million) face significant difficulties in functioning and require healthcare assistance. Moreover, the Organization emphasized the strong lack of appropriate services for people with disabilities, an issue that is further mirrored in the Information Technologies realm. New technologies bring new opportunities, definitely. But also new challenges and new needs. Web 2.0 users with disabilities in fact require alternative tools and strategies to benefit from the rich and endless content available on the Web. "The quality of being easily accessible, accessible, or usable by people with disabilities" is defined by the Oxford Dictionary[2] as *accessibility*. Therefore, *Web accessibility* denotes the same quality but specifically concerning websites,

[1] World Health Organization (WHO) Web site. Online available at https://www.who.int/news-room/fact-sheets/detail/disability-and-health. Last accessed Sept. 29, 2021.

[2] OED Online (2021). Oxford University Press. Online available at https://www.oxfordlearnersdictionaries.com/definition/english/accessibility. Last accessed Sept. 29, 2021.

tools, and technologies[3]. Web accessibility, or "accessible Web design" or "design for all", is achieved through the harmonious conjunction of two primary aspects: the employment of special assistive equipment (or assistive technologies, AT) and the provision of good interface design practices. The former are leveraged by users with disabilities to access on-screen information and provide output tailored to their needs. The latter, on the other hand, deliver the on-screen information in the most interpretable way possible by the assistive technologies.

Disability is extremely diverse, and likewise diverse are the many technologies used to access Web content. In particular, according to researches[4] and studies[5], some particular groups benefit the most from accessible Web design:

- people with vision impairment, in the form of moderate vision loss in one or both eyes ("low vision"), who need, for instance, to enlarge text and image sizes, and to customize fonts, colors and spacing;
- people with substantial and uncorrectable vision loss in both eyes ("blindness"), who must rely on technologies that operate text-to-speech synthesis of content or equipment that provides refreshable Braille;
- people with cognitive, learning, and neurological disabilities such as dyslexia or autism spectrum disorder (ASD), who require clearly structured content that facilitates orientation, with consistent layouts;
- people with physical impairments such as quadriplegia, tremor, spasms and muscular dystrophy, who make use of alternative devices such as head pointers, mouth sticks, on-screen keyboard with trackball, voice recognition, eye tracking, etc.;
- people with hearing impairments who rely on transcription and captioning of audio content.

This dissertation focuses specifically on non-sighted users, whose main support in the interpretation and interaction of on-screen information is represented by screen readers, software applications that render Web content as synthetized speech or braille output. These

[3] W3C | Web Accessibility Initiative. Available at https://www.w3.org/WAI/fundamentals/accessibility-intro/. Last accessed Sept. 29, 2021

[4] P. Brophy, and J. Craven. (2007). *Web accessibility*. In Library trends, 55(4), 950-972.

[5] S. Harper, and Y. Yesilada. (2008). *Web accessibility: a foundation for research*. In Springer Science & Business Media.

assistive technologies also provide useful functionalities such as navigating through headings, simplifying the search of relevant information and identifying internal and external links. Different screen readers are available depending on the chosen operating system: for instance, macOS includes VoiceOver, while for Microsoft Windows, the most popular[6] today are JAWS ("Job Access With Speech") by Freedom Scientific, and the open-source NVDA ("NonVisual Desktop Access") by NV Access.

The worth of screen readers is clear and well acknowledged. In addition to the functionalities briefly outlined before, their useful features also comprise customizable options: users can tailor their needs, for example, by specifying the verbosity settings on their device that dictates how much page detail they would like to receive. They can even choose to read punctuation or ignore it, speed up reading velocity, and switch to a different language, including accents (e.g., in the case of a website from the UK).

However, their limitations are recognized and debated as well[7]. Among these, the major shortcoming is the failure to convey the overall organization of a Web page, with relative priorities, which is normally interpreted by non-disabled users through vision. In other words, the "book-strategy" (left-right, top-bottom) of screen readers is not at all equivalent to the "at a glance" comprehension of sighted users. This dilemma exacerbates with the vertiginous evolution of user interfaces over time: originally conceived to be mainly text-based, in fact, the Web is today the epicenter of graphic applications, where visual features (images, graphics, etc.) convey an important and fundamental part of the overall semantics.

In order to cope with these inherent limitations of screen readers, it was deemed necessary to structure and enrich Web pages with all the attributes and annotations to make them interpretable in the most optimal way and without loss of meaning. To define clear methods and methodologies, the World Wide Web Consortium (W3C) promoted the Web Content Accessibility Guidelines (WCAG), nowadays the international standard of reference when considering the accessibility of digital content. This specification has been developed over decades, evolving from the first version 1.0 to 2.0, and then finally to 2.1. This process is described in greater depth in Chapter 1. The W3C's effort was paramount in raising awareness

---

[6] WebAIM | *Screen Reader User Survey #9*. (2021). Online available at https://webaim.org/projects/screenreadersurvey9/. Last accessed Oct. 14, 2021.

[7] Di Blas, N., Paolini, P., and Speroni, M. (2004). *Usable accessibility to the Web for blind users*. In Proceedings of 8th ERCIM Workshop: User Interfaces for All, Vienna.

of Web accessibility, but some limitations and problems soon arose in consulting its guidelines and specifications, including their over technicality and their inadequate issues coverage. Chapter 2 examines how these guidelines have been received by the developer and author community and how successful they really were (and are nowadays) in solving actual real-life accessibility problems.

To support developers in the troublesome subject of Web accessibility, not only official guidelines have been proposed over the years, but also resources capable of detecting accessibility issues, flagging them and providing helpful solutions: these accessibility testing tools, or checkers, have proven to be very useful in coping with the tedious manual work of evaluating even large and complex Web sites. However, studies reviewed in Chapter 3 have proven that they themselves cannot compete with the expertise of a senior developer with a deep knowledge of accessibility. After beginning by providing an overview of the main testing tools available on the market, the Chapter then concludes by asking what their percentage of completeness, correctness and reliability really is.

In the subsequent chapter, Chapter 4, the various problems of sighted developers are further examined. Indeed, webmasters who aspire to develop accessible Web sites must also contend with the actual technology used to build and shape the Web: HMTL5. Among widgets not recognized by browsers, and code strings that work perfectly for sighted users but not for users with disabilities, HTML5 stands out as a tricky technology, declared natively accessible but without fully being so. This frequently leads developers to implement widgets and interactive elements not even noticing that they are not accessible to a broad segment of the population. And noticing it is not at all easy: in this context, indeed, sight appears to be a major "handicap" which prevents the real perception of how a Web site or an application is experienced by a user with disabilities. Thus, sighted developers suffer from a natural “cognitive bias” that prevents them from successfully testing interactive components and page structures without visual clues and hints. To reduce the handicap of sighted developers is therefore introduced SahARIAn (Chapter 5), the software at the heart of this thesis project and this dissertation.

SahARIAn visually transforms a Web site it in such a way that developers naturally sense the correctness or incorrectness of their markup and, thus, readily grasp accessibility issues. To accomplish this task, it empties Web pages of their presentation and injects specific style sheets that emphasizes their plain structure on the basis of their ARIA accessible markup. As a final result, the less accessible the page is for a user with a disability, the more obvious it is rendered

in a very cluttered and not graphically pleasing Web content, with inoperable widgets even using a mouse or a trackpad. The final Chapter digs deeper into the "rabbit hole", reporting the architecture of SahARIAn and some of its implementation choices, both at the level of algorithms and at the level of CSS rules used to visually render the transformed Web page.

This dissertation closes with the "Conclusion" section, in which some final considerations are proposed: SahARIAn's strengths, limitations and bug fixes are reported, together with areas of improvement to be implemented. Moreover, foundations and ideas for fruitful future developments are given.

Chapter 1

Accessibility standards

1.1 WCAG 1.0: guidelines to accessibility

The need for World Wide Web resources to be accessible to people with disabilities has always been imperative. In 1994 Tim Berners-Lee, inventor of the World Wide Web (1989), founded the World Wide Web Consortium (W3C), a non-profit organization with the aim of developing protocols and standards for the Web to ensure its long-term growth and interoperability. Few years later, in 1997, he launched the International Program Office (IPO) for the Web Accessibility Initiative (WAI). "The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect"[8], he stated. Sponsored by, among others, IBM/Lotus Development Corporation, Microsoft Corporation, the US Department of Education's National Institute on Disability and Rehabilitation Research, the purpose of this initiative was, and still is, to promote Web functionality for people with disabilities, with guidelines, techniques, reports and supporting resources.

In 1999 the WAI defined the first version of the "Web Content Accessibility Guidelines" (WCAG 1.0), intended for Web content developers, page authors and site designers. It was the first official document that provided a comprehensive set of guidelines for accessibility principles and design ideas that took into account, and indeed focused primarily on, users with disabilities. The following year (2000), the requirements expressed in WCAG 1.0 were expanded and detailed in a series of related documents:

- "Techniques for Web Content Accessibility Guidelines 1.0";
- "Core Techniques for Web Content Accessibility Guidelines 1.0";

[8] Tim-Berners Lee. (1997). *World Wide Web Consortium Launches International Program Office for Web Accessibility Initiative. Government, Industry, Research and Disability Organizations Join Forces to Promote Accessibility of the Web*. Available at https://www.w3.org/Press/IPOannounce. Last accessed Sept. 23, 2021.

- “HTML Techniques for Web Content Accessibility Guidelines 1.0”;
- “CSS Techniques for Web Content Accessibility Guidelines 1.0”.

Each of these documents provided techniques, examples, and strategies for meeting the WCAG 1.0 specifications in different ways, such as, in the case of the last two, in the writing of HTML and CSS markup languages.

WCAG 1.0 consisted of fourteen guidelines addressing two general themes:

- *Ensuring graceful transformation of Web*. Here, “graceful transformation” refers to the capacity of a Web page to remain accessible despite the existence of constraints such as physical, sensory, and cognitive disabilities, work constraints, and technological barriers;
- *Making content understandable and navigable also to people with disabilities*, by providing navigation tools and orientation information to help users understand complex pages or elements..

The guidelines were, indeed, general principles of accessible design, explained briefly and with some mention of HTML and CSS implementations. The first guideline, for instance, recommended writing an alternative text (via the “alt” attribute) for elements such as `<img>`, `<area>` and `<input>`, or additional descriptions for complex contents such as charts, in order to provide equivalent alternatives to auditory and visual content. The second rule ensured that text and graphics could be read and semantically understood when viewed without color, and that foreground and background color combinations could offer enough contrast for users having color deficiency or blindness.

Each of the fourteen guidelines was related with one or more checkpoints (CPs) that detailed how developers could adapt their Web content to make it accessible, for a total of sixty-five checkpoints. In addition, these checkpoints were assigned a priority level, Priority 1 through Priority 3, based on the importance of the checkpoint itself in impacting the accessibility of the Web content. A checkpoint was marked with *Priority 1* if it was *required* to be met by the Web content developer. Failure to observe this checkpoint would have made it impossible for one or more user groups to access information contained in Web content under consideration.

A checkpoint with *Priority 2* marking meant that a Web content developer did not have to satisfy it but was only recommended to do so. The underlying reason is that meeting that checkpoint would have removed significant barriers to accessing Web documents.

Finally, *Priority 3* was the lowest level of priority: not meeting a checkpoint at this level implied that it would have been difficult for some user groups with disabilities to access a certain information. Conversely, satisfying it would have improved access to it. A Web content developer was not obliged to meet this checkpoint, but could just address it. For example, in the context of guideline 13 ("ensure that documents are clear and simple"), checkpoint 14.3 asserted that creating a style of presentation that was consistent across pages promoted effective communication and easier access to written information for people who had cognitive or learning disabilities.

If a Web page satisfied all Priority 1 checkpoints, it was said to be conformant at Level A. Likewise, conformance to Priority 1 and 2 CPs is described as Level AA (or Double-A). Finally, if a website satisfied all CPs, it was conformant to Level AAA (or Triple-A).

1.2 WCAG 2.0: better, clearer, stricter

However, even in its profound innovation, WCAG 1.0 suffered from some limitations: first of all, its guidelines were strongly based on HTML and CSS, while they needed to be technology-neutral instead. What is more, they required more general applicability and higher precision. Not to mention that, with the passing of time, they began to become obsolete as an increasingly sophisticated and complex Web began to challenge principles of accessibility that were slow to update.

In 2008 WAI published WCAG 2.0 as a W3C recommendation that applies not only to more advanced technologies, but also to technologies that will be developed in the future. Its criteria are written as testable statements that are not technology-specific, and the guidance about satisfying them in specific technologies, as well as general information about interpreting them, is provided in separate documents. WCAG 2.0 is flanked, in fact, by "Authoring Tool Accessibility Guidelines" (ATAG) and "User Agent Accessibility Guidelines" (UAAG). The first one is a set of guidelines for developers of any kind of authoring tools used to create Web content: code editors, Web content management systems (CMS), social media and many more. The main goal here is not only to create accessible tools for authors with disabilities but also to

produce accessible content by default and, as a result, to encourage other authors to create it. The latter (UAAG) details how to make user agents accessible: web browsers, media players, and other software tools that are used to access, render, and interact with Web content.

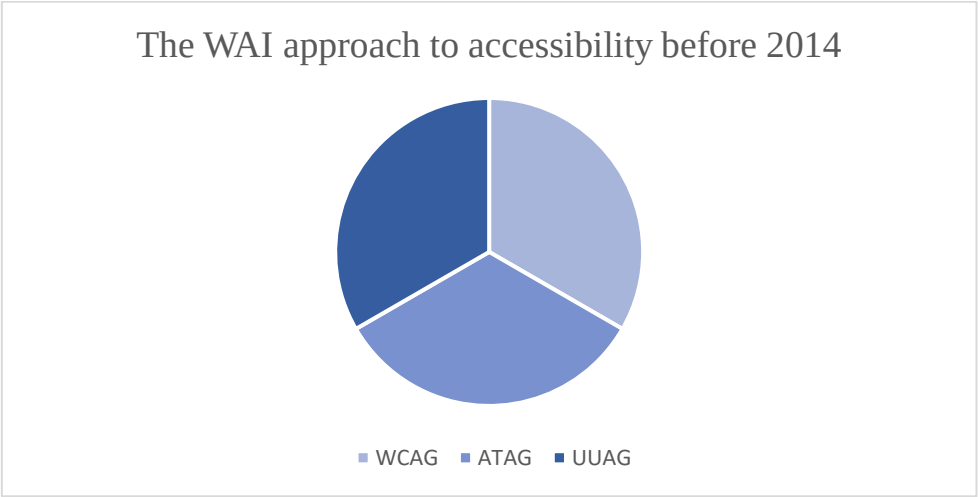


Figure 1.1. The WAI approach to accessibility before 2014

As was the case of WCAG 1.0, WCAG 2.0 not only promotes recommendations for improving the accessibility of Web content, but also makes it more enjoyable and interactive for the general public. Significant innovations were made. First of all, the criteria are clearer, more flexible and supported by practical implementation examples. Furthermore, WCAG 2.0 is designed to be durable as technology evolves over time, and to be applicable as new approaches emerge. In order to reach its accomplishment, it has to be generalizable and technology-independent. As a result, WCAG 2.0 specifies what needs to be done for accessibility in terms of the functionalities that are needed for users. It is not prescriptive in terms of how to implement something: a related document, "Techniques and Failures for Web Content Accessibility Guidelines 2.0" was published exactly for this reason, to explain in depth how to achieve accessibility criteria. The "Techniques" provide specific details, examples (such as HTML and CSS code), resources, tests and common failures of applications not meeting the criteria.

Primarily a series of fourteen guidelines, WCAG 2.0 was transformed into twelve guidelines with the same goal of ensuring that Web content is directly accessible to as many people as possible. The "checkpoint" label of WCAG 1.0 was discarded in favor of the

expression “Success Criteria” (SC): each of the sixty-one Success Criteria of WCAG 2.0 is placed under one of the guidelines to specifically describe what must be accomplished in order to meet the standard. And that's not all: they all feature a link to the living document “How to Meet WCAG 2.0”, that provides advisory techniques for meeting the Success Criteria, tests, code examples and explanations of the intent of the guidelines and how they help people with different disabilities.

The guidelines and Success Criteria are organized around four major accessibility principles that define the basic level for accessing and using Web content by any user, with and without disabilities. In particular, for people with disabilities, in order to use the Web, the content must be:

- *Perceivable*. Users need to be able to perceive with their senses the information within the content. For example, if a person is visually impaired, an alternative way must be found to get the information to them that they cannot get through their eyes. In the simplest case, i.e. images and graphics on the Web, textual alternatives must be offered. Creating content that can be presented in simpler or different structures without losing information or depth is another key requirement.
- *Operable*. Web pages are mostly about interaction and navigation. It may happen that the interface, although user-friendly, requires an interaction that the user is not able to perform: this leads to the site being inaccessible in whole or in part and, therefore, must be avoided. For example, actions carried out with the point-and-click method (mouse) must be designed to be performed with various inputs as well, including the keyboard, the main device at the disposal of visually impaired users. Similarly, good practices include providing alternatives to anything that is time-based, avoiding imposing time limits on how long it takes someone to read a piece of information and always determining where the users are providing ways to easily navigate and browse the Web page.
- *Understandable*. A perceivable and operable Web content is not accessible if it lacks the principle of understandability. Users must be able to understand the information and the operation they are performing on the Web page, along with how to use and navigate the Web site. Thus, the text content must be readable and understandable, in clear language, the Web pages must appear and operate in predictable ways, according to the overall style but also to the structure and the

functionality, and input assistance must be tailored in such a way to help users avoid and correct mistakes.

- *Robust*. This principle shifts the focus into the future, centering accessibility not only on the present condition of the Web, but also on how it will adapt, change and develop as technologies, including assistive one, progress. Perceiving, interacting and understating a Web content, but not accessing it (or partially accessing it) due to a failure to update and align with the latest browsers, makes it equally inaccessible. Alongside ongoing maintenance, the site must be robust in the sense that it must take into account the needs of a wide range of users with different disabilities.

Again, as with WCAG 1.0, the three levels of conformance to accessibility requisites are reconfirmed, with A meeting the bare minimum standard required and AAA representing the highest desirable rating.

1.3 Too dynamic to be accessible: Ajax

For nearly a decade, WCAG 1.0 and 2.0 had achieved their goal: to raise accessibility awareness worldwide and provide global guidelines for the development of its practices. However, the very Web that was born as a hypertext application of simple HTML static text files linked together soon began to grow both in size and complexity. Pages started to promote a staggering increase in embedded third-party services such as advertising, analytics, and CDNs. But that's not all: their structure, with the advent of Ajax ("Asynchronous JavaScript and XML"), Flash, Microsoft Silverlight and other powerful development tools and techniques, became incredibly sophisticated, fast and dynamic.

All of this at the expense of accessibility.

Let's have a closer look at it. At the beginning, most Web sites were based on complete HTML pages. The traditional Web interaction required each client to make an http request to the Web server, which would have returned a full new Web page. In other words, each user action prompted a new full page to be loaded by the server, with a time lag ranging from a handful of milliseconds to several seconds. This process was therefore characterized by a quite long waiting time and a high inefficiency: a Web page was forced to reload every time it presented a minimal or partial change, as in the case of a news feed Web app or site that had to

update its entire page upon publication of a new news, and thus re-sent all of the content, instead of adding it to the top of the list. In 2005, as the Web was becoming richer and responsive, Jesse-James Garrett published the online article[9] "Ajax: A New Approach to Web Applications", defining Ajax as a set of Web development techniques that uses various Web technologies, architectural styles and ideas to manage an asynchronous server-client interaction. That is, multiple data exchange requests executed by the server in parallel, causing responses to be received quickly and in the background, without the user experiencing any visual interruption.

Today, its popularity stems from high-profile applications like Gmail, Flickr, Apple iCloud, Google Maps, eBay and many more. Ajax, along with other web technologies such as Adobe Flash, Flex, ActionScript and Silverlight, has allowed Web developers to let their creativity run wild, introducing eye-catching designs, with widgets that appear, disappear or change on the fly. The final products took the form of stunningly fast, dynamic, and interactive applications with complex architectures that molded to users' preferences and expectations. These enhanced contents are known as Rich Internet Applications (RIAs).

RIAs are employed not only because they provide better usability and an adaptive and attractive design, but also because they provide significant savings on the data consumed by the site in order to be accessed. This, together with the elimination of page-reloading-related interruptions, results in Ajax applications performing better than outdated synchronous Web applications.

Improvements for users, undoubtedly. But not for all of them. These sophisticated dynamic Web applications represent a major stumbling block for people with disabilities, whose screen readers, communication boards and other devices either can't recognize (yet) widgets that appear dynamically on the screen or, if they do recognize them, can't access them. This may be the case, for example, of HTML alert windows that stuck the users, or image carousels that change without warning. Users of assistive technology are also not aware of updates when they occur on a different section of the Web page than the one they are interacting with, or when clicking a link updates a different section of the page, perhaps without even informing them that the control has been activated. This happens because screen readers perform well on static

[9] J. J. Garrett (2005). *Ajax: A new approach to web applications*. Adaptive Path Inc, 30.

Web pages and are perfectly capable of alerting users of page changes on refresh, but not on most dynamic content changes.

At a technical level, these problems arise because RIAs dynamically manage the content presented to users by proposing a navigable model, and this often conflicts with the model itself proposed by the screen reader, which also works with a similar *modus operandi*, by proposing a model itself. This feature, called virtual buffer, retrieves information from the Web page and generates an in-memory copy of it, allowing users to easily navigate the page using the screen reader navigation commands. As one can imagine, it's not difficult, indeed it's quite easy, that the two models result not synchronized, disorienting the users in the actions, navigations and information presented to them.

This discrepancy of models, without the support of feedback that alerts the user, generates undesired issues that can be collected into three groups:

- *Notifications*: a user may not realize that the content has been updated and therefore cannot locate neither the location of the hypothetical new text, nor his or her own position in the refreshed page.
- *Layout*: a user may be forced to re-read again an entire Web page after realizing that the structure of the page has changed. In the optimal cases, it may happen that his or her position (the Virtual Cursor) in the buffer has moved slightly somewhere else. But it may happen that the screen reader can't recognize the structure of the Web site anymore, so it may start reading HTML/CSS from the source instead of the meaningful content.
- *Confirmation feedback*: this is needed whenever the user performs an action, to discern whether it was successful or failed. It is found in the case of all links or buttons and in general interactions with the Web site.

Accessibility at the onset of each of these scenarios is greatly reduced and renders the site or part of it unavailable. Unfortunately, the problems that arise within them are thousands and all different. No wonder: screen reader software was created in the 1980's and has benefited from a static Web for many years. But as the years went by, the gap between functionality and usability was alarmingly widening more and more.

1.4 WAI-ARIA to the rescue

The response of the W3C was not long in coming: it started in 2006 with a first Working Draft and continued with an official Recommendation in 2014, when WAI published WAI-ARIA 1.0 ("Web Accessibility Initiative - Accessible Rich Internet Applications"), a technical specification focused on the accessibility of web pages in dynamic contexts and with user interfaces developed with Ajax and the new asynchronous technologies. Since this new WAI Accessibility Guideline was published as a W3C Recommendation, it became one of the four Web Standards promoted by W3C, together with the aforementioned WCAG, ATAG and UUAG.

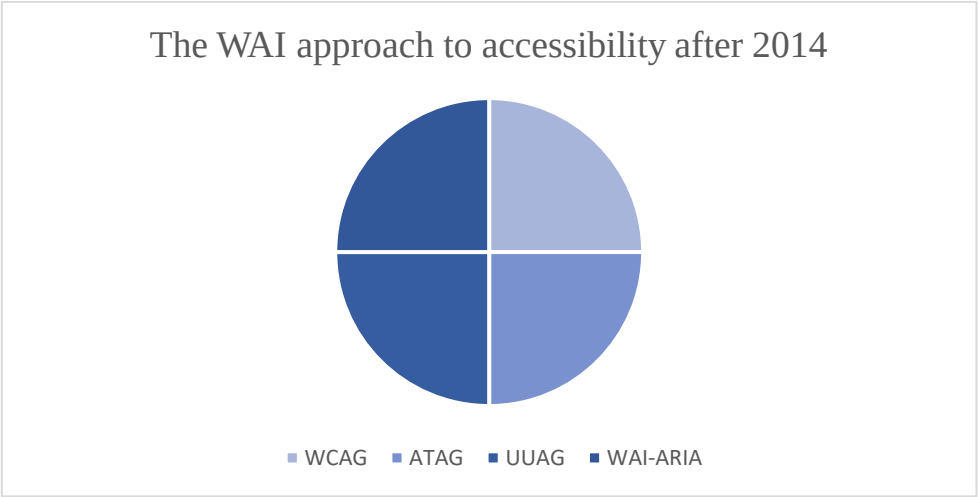


Figure 1.2. The WAI approach to accessibility after 2014

WAI-ARIA Recommendation was published along with a suite of documents, including:

- "WAI-ARIA Overview": an introduction to WAI-ARIA. It describes the underlying concepts, the technical methods, and business motivations for implementing it.
- "WAI-ARIA Authoring Practices": a guide for understanding how to use WAI-ARIA to create an accessible Rich Internet Application. It explains in detail with examples, patterns and HTML and CSS code how to implement the concepts of WAI-ARIA in more than thirty accessible widgets (such as dialogs, grids,

carousels, sliders, etc.), with keyboard navigation, reusable component libraries and testing.

- "Accessible Name and Description Computation": a document that provides algorithms and information on how assistive technologies compute the name and description of accessible elements and, in turn, how it is possible to compute them by Web developers. In particular, as will be discussed later on in the dissertation, the accessible name may be derived from visible (e.g. the text on a button) or invisible (e.g. the text alternative that describes an icon) properties of the element. The accessible description provides additional information that complements the accessible name.
- "Core Accessibility API Mappings": describes how user agents should expose semantics of Web content languages to accessibility APIs (Microsoft Active Accessibility, ATK, macOS Accessibility Protocol, Microsoft User Interface Automation, and others). The core module defines support that applies across multiple content technologies, including general keyboard navigation support and mapping of general-purpose WAI-ARIA features.
- "Roadmap for Accessible Rich Internet Applications" (WAI-ARIA Roadmap): mostly describes the new navigation techniques of accessible RIAs, such as menus, primary and secondary content, banners and other structures. It also tracks the path needed to make Web content accessible, including what has already been done and what has to be done in the future.

The fundamental contribution of WAI-ARIA lies in the definition of HTML attributes that can be applied to classic HTML elements in order to enrich them with semantics and to improve accessibility wherever it is needed.

These ARIA attributes are divided in three categories, according to how they describe the structure of the Web page and how they define the current conditions of the elements (which, as dynamic they are, can change by their own or by interactions with the user). They are *roles*, *properties* and *states*.

Roles are machine-extractable semantic information about the purpose of an element. WAI-ARIA provides a taxonomy of roles, expected behaviors and features, including:

- a detailed explanation of the role;

- hierarchical information between roles (e.g. a `gridcell` must be a child of `grid` and a `row` must be contained within a `group`);
- connection and context of role with the other (e.g., a `listitem` is contained inside a `list`);
- compatible states and properties for each role (e.g., a `slider` supports a maximum value allowed through the attribute `aria-valuemax` as well as a minimum value via `aria-valuemin`); and
- references to related concepts in other specifications.

A WAI-ARIA role is assigned to an element using `role` attributes. These are divided into six categories based on whether they define user interface widgets (checkboxes, comboboxes, sliders, etc.) or page structure (sections, articles, navigation, etc.):

- *Abstract Roles* define general concepts and must not be used by authors and developers in their Web content. They are, for example, `landmark`, `composite` and `window`;
- *Widget Roles* can be implemented as standalone interactive components of the interface or assembled as part of composite widgets. Many of them follow the semantic value of HTML structural elements, e.g. `role="radio"` (HTML: `<input type="radio">`) and `role="button"` (HTML: `<button>`), but there are also others that infer accessible meaning and information to complex HTML elements, such as `role="switch"` and `role="treeitem"`;
- *Document Structure Roles* describe structures that arrange and organize content in a page, such as `figure`, `list` and `article`;
- *Landmark Roles* are set on parts of the Web page which the user may want quick access to because of specific purpose, such as reading the primary content, navigating or searching. These elements are intended as navigational landmarks and some of the most important are `main`, `navigation`, `search` and `form`;
- *Live Region Roles* represent dynamic contents which update without a page reload. They can be interactive or not and their changes must always be announced by the assistive technologies by means of well-implemented ARIA attributes like `aria-live`. Elements that are considered live regions, for example, are `log`, `status` and `alert`;

- *Window Roles*, lastly, are used to set dialogs or windows that separates content from the rest of the Web page. They are `alertdialog` and `dialog`.

Properties represent the second category of attributes introduced by ARIA. They are used to give elements extra meaning or semantics, for example representing a data value associated with them. For instance, `aria-label` allows the author to define a string that labels the element without showing it on the screen, while `aria-required` indicates that user input must be entered before a form can be submitted, in order to be valid.

Properties differ from the last category presented by ARIA, *states*, which are nothing but special properties that depict dynamic characteristics of an element that may change in response to automated processes or user actions. They define their current conditions: for example, `aria-checked` state is always present in a `role="radio"` element because it indicates whether the option is selected or not. In the first case, it will change on-the-fly to `aria-checked="true"`, while in the second case it will remain `aria-checked="false"`. Another useful example is `aria-disabled="true"`, which specifies to a screen-reader that an element is perceivable but disabled. As one might expect, a change of state during the element life-cycle may significantly impact its meaning or presentation.

Currently, after ARIA versions 1.0 (2007) and 1.1 (2014), ARIA version 1.2 is under development and today, together with WCAG 2.1 (2018), ARIA has become one of the most widely adopted accessibility standards, with more than 25% of the top million websites using it to some extent. However, it is important to note that, if used improperly, ARIA is as useful as harmful. Even though ARIA roles, properties and states provide powerful and efficient semantics to widgets and elements, authors and Web developers should always stick to native HTML features, whenever possible. This avoids setting up erroneous semantics and prevents redundancy.

HTML5, the latest version of HTML (2014) and the most widely used markup language on the Web today, in fact, is for the most part naturally accessible. Some HTML elements, in fact, clearly define their scope, structure and definition, such as `<button>`. In this case, the syntax `<button role="button">` should be avoided. When possible, using the least amount of additional code to convey the necessary information is ideal. However, several features of HTML5 are not properly supported by all browsers (for instance, the modal dialog for Apple Safari) and others (such as tabs, tab panels, toolbars, trees, etc.) haven’t yet come up with a suitable agreed-upon representation of HTML markup to base robust accessibility

support on. In these cases, enriching HTML tags with WAI-ARIA is the best and recommended practice. Specifics of HTML5 accessibility are discussed in Chapter 4.2, where the issues of sighted and non-disabled developers are considered in more detail.

The “WAI-ARIA Authoring Practices 1.1” itself emphasizes that “No ARIA is better than Bad ARIA”, as the incorrect markup may misrepresent visual experiences without the Web designers even noticing, with potentially devastating effects on the perception of the Web page or application by users with disabilities. This concept is what underlies this dissertation and places emphasis on the inherent disability of sighted developers in the design of Web applications. Several tools have been designed to address this “impairment” and numerous studies have been conducted to address accessibility issues arising from the large number of published guidelines and their ability or otherwise to solve practical problems. The next Chapter examines how the guidelines have been received by the developer and author community and what the main problems have been in their implementation.

Chapter 2

Limitations of accessibility standards

2.1 WAI guidelines: an online paper maze

Chapter 1 presented only a narrow overview of the spectrum of documents provided by W3C to foster an accessible Web. However, W3C has been much more prolific in promoting standards: to date (Sept. 2021), the considerable number of published specifications and guidelines amounts to eighty-one, divided into thirty-one W3C Working Group Notes, thirty-three W3C Working Drafts and seventeen W3C Recommendations.

To encourage adherence to accessibility principles and avoid unjustifiably discrimination through the provision of a Web site with accessibility barriers, legislations have also emerged in various parts of the world. Among others, it is worth mentioning Section 508 of the Rehabilitation Act[10] (US, 1998), the more recent Americans with Disabilities Act[11] (ADA, 2009), the Disability Discrimination Act[12] (Australia, 1992), the Stanca Act[13] (Italy, 2004) and the European Web Accessibility Directive[14], or EU Directive 2016/2102. They spell out mandatory government requirements for Web sites and applications.

---

[10] U.S. Access Board | *Section 508*. (1998). Online available at https://www.access-board.gov/ict/. Last accessed Oct. 9, 2021

[11] Ada.gov | *Americans with Disabilities Act* (2009). Online available at https://www.ada.gov/pubs/adastatute08.htm. Last accessed Oct. 9, 2021

[12] Australian Government Federal Register of Legislation | *Disability Discrimination Act*. (1992). Online available at https://www.legislation.gov.au/Details/C2018C00125. Last accessed Oct. 9, 2021.

[13] Gazzetta Ufficiale della Repubblica Italiana | *Disposizioni per favorire l'accesso dei soggetti disabili agli strumenti informatici*. (2004). Online available at https://www.gazzettaufficiale.it/eli/id/2004/01/17/004G0015/sg . Last accessed Oct. 9, 2021

[14] EUR-lex | *Web Accessibility Directive*. (2012). Online available at https://eur-lex.europa.eu/eli/dir/2016/2102/oj. Last accessed Oct. 9, 2021

Yet, despite the existence of well-acknowledged guidelines, legislations and expanding support networks and communities committed to Web accessibility, a great majority of Web sites and applications today can't claim compliance with WCAG Level AA or even just Level A. In their study[15] on developers' and designers' perceptions of accessibility, Lazar, Dudley-Sponaugle and Greenidge conducted the first large-scale survey to reveal what the biggest obstacles to implementing accessible sites were. Participants (175 webmasters and developers) were asked to take part in the study using a Web questionnaire distributed via e-mail. Most respondents (65.7%) said that they had previously developed an accessible Web site, 73.7% stated they were aware of accessible legislation, and 69.1% reported they had already used an assessment tool. Despite this, they pointed out that there were numerous challenges of making a Web site accessible for users with visual impairments. Among the main ones, they mentioned the difficult balance between accessibility and graphical design: clients demanded glitzy dynamic Web sites full of widgets and features that were hard to make accessible. The very importance of accessibility to stakeholders is often considered trivial or unimportant, resulting in a lack of funding. But that's not all: accessibility also requires study and development time that needs to be considered in every phase of the software workflow.

A second study was sponsored by ENABLED Group (Enhanced Network Accessibility for the Blind and Visually Impaired) and found that, among the 269 participants (webmasters, managers, and content editors) only 36% were committed to the accessibility of their Web sites. The underlying reasons came in the form of the increased time and technical knowledge required, and most importantly, in the challenging comprehension of accessibility guidelines.

According to the study[16] carried out by Ferreira, dos Santos and da Silveira, which involved the evaluation of Brazilian government Web sites, 47% of the surveyed organizations stated that their sites were not accessible at all and that no adaptations were being made. The justification behind this was that the biggest challenges to adapt their websites were in the huge amount of information, the lack of experts and the too many standards to be observed.

---

[15] J. Lazar, A. Dudley-Sponaugle, and K. Greenidge. (2004). *Improving Web Accessibility: A Study of Webmaster Perceptions*. In Computers and Human Behavior, 20(2):269.

[16] S. B. L. Ferreira, R. C. dos Santos, and D. S. da Silveira. (2007). *Panorama of Brazilian Web Accessibility*. In Proceedings of the XXXI ANPAD Meeting-EnANPAD.

Finally, with their educational experiment[17], Alonso, Fuertes, and González found that twenty-five evaluators struggled to consistently identify accessibility problems according to WCAG 1.0, due to the difficult interpretation of the guidelines and the language used in their Success Criteria and concepts. Colwell and Petrie's study[18] confirmed these findings: their aim was to investigate whether WAI guidelines could be effectively used by webmasters and authors. In order to figure it out, twelve experienced Web developers newbies to accessibility evaluated WCAG and used its principles to adapt Web pages; as a conclusion, these pages were assessed and rated by visually impaired Web users. This evaluation demonstrated that developers have struggled with the language used in WCAG documents, characterized by ambiguity and logical flaws. They also encountered problems navigating between them and stressed the fact that it takes a high level of technical knowledge to read and interpret them.

These studies opened the doors to interesting questions concerning the difficulty of employing and comprehending the dungeon of accessibility guidelines. In their enquiry[19], Kelly and Sloan highlighted issues with the WCAG guidelines and challenges in their implementation. Initially, they questioned the overly theoretical nature of the guidelines: for example, only a decade after the publication of WCAG, WAI considered the techniques for PDF accessibility, while right from the start it focused only on opened W3C technologies such as RDF and SVG (more infrequent and, at the time, less likely to be found in practice). Not to mention the complexity of the guidelines: over time, many documents have been written by other organizations, including governmental ones, in order to interpret and explain the principles and procedures contained in them. However, this has sometimes led to even greater confusion and more documents to consult.

[17] F. Alonso, J. L. Fuertes, Á. L. González, and L. Martínez. (2010). *On the testability of WCAG 2.0 for beginners*. In Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A).

[18] C. Colwell, and H. Petrie. (2001). *Evaluation of guidelines for designing accessible Web content*. In SIGCAPH Comput. Phys. Handicap., 70, 11-13.

[19] B. Kelly, D. Sloan, L. Phipps, H. Petrie, and F. Hamilton. (2005). *Forcing standardization or accommodating diversity? A framework for applying WCAG in the real world*. In Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A).

2.2 Turning an eye blind: the inadequate issues coverage

The limitations of the guidelines do not end there. Even if WCAG standards were applied consistently, they themselves are often inadequate to fully guarantee accessibility and cover the real issues encountered by users with disabilities, such as visually impaired users. In their study[20], Power, Freire, Petrie and Swallow revealed a complex relationship between problems encountered by screen reader users and WCAG 2.0. In their empirical inquiry involving thirty-two blind participants and sixteen Web sites, they have achieved worrying results: only 50.4% of the issues encountered by users were covered by Success Criteria expressed in WCAG 2.0. And among those that were actually covered by WCAG 2.0, the techniques recommended by WCAG 2.0 and successfully implemented by webmasters, for 16.7% of Web sites still couldn't solve the problems. Specifically, they divided the overall set of the 1383 user problems into three types (Figure 2.1):

- issues covered by guidelines but not implemented (42%);
- issues covered by guidelines and implemented by developers (8.4%); and
- issues not covered by guidelines (49.6%).

[20] C. Power, A. Freire, H. Petrie, and D. Swallow. (2012). *Guidelines are only half of the story: accessibility problems encountered by blind users on the web*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12), Association for Computing Machinery, New York, NY, USA, pp. 433–442.

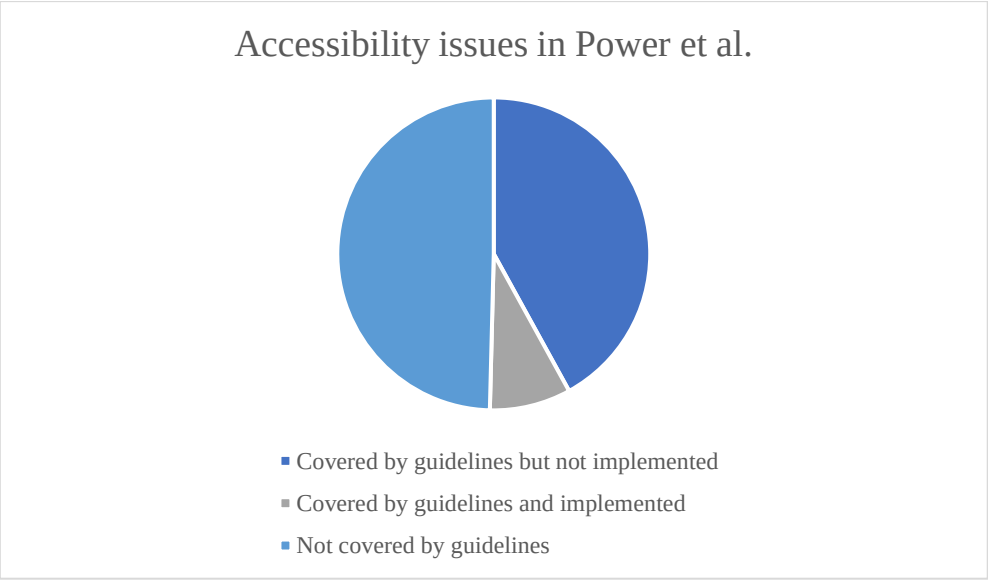


Figure 2.1. Accessibility issues in Power et al.

In other words, the last worrying percentage (49.6%) meant that almost half of the accessibility problems was not captured by the current official guidelines. User issues not covered by WCAG 2.0 fell primarily into six categories:

- *Content found in pages where it is not expected by users;*
- *Content not found in pages where it is expected by users;*
- *Too complex information architecture;*
- *Absent or insufficient feedback to inform that actions have had an effect;*
- *Irrelevant content before task content;*
- *Broken links.*

The first category, *Content not found in pages where it is expected by users*, refers to situations in which users confidently followed a link to a page, only to realize that a piece of information they expected to find there was absent. The grouping *Content found in pages where it is not expected by users* describes the inverse scenario. Together, they account for 13.5% of all user problems.

The inadequacy of the WCAG 1.0 and 2.0 guidelines was also investigated by Rømen and Svanæs, who focused on issues that were not being covered by the WAI documents. In a first

study[21], they conducted usability tests on two Web sites with users with various disabilities, including partial or total visual impairment, dyslexia, and motor impairment. The analysis found that only 27% of the detected problems were covered by WCAG 1.0 checkpoints. A subsequent study[22] was made after the publication of WCAG 2.0 standard, to check if there had been improvements in the coverage of user issues: in fact, the percentage increased, but remained low, resulting in a modest 32%.

These researches revealed several concerns about evaluating Web sites only on the basis of WCAG and related documents, which may not address all of the issues that disabled users face online. Thus, Calvo, Seyedarabi and Savva proposed a study[23] that provided additional knowledge into potential problems not covered by WAI guidelines, suggesting some future improvements. Their research, conducted on sixty-two Web sites and mobile applications, identified a total of 1214 issues, including WCAG issues and additional issues not addressed by them. The latter belong to seven categories:

1. *Information incorrectly hidden*. In the development of responsive Web sites, webmasters hide information visually. If it is not done properly, people with disabilities navigating with keyboards and screen readers might have problems to understand where the focus is placed, and to interact with elements that cannot be selected or pressed but indeed can be erroneously read by screen readers.
2. *Misusage of common design patterns*. Expected specific components should act in specific ways and be located in specific positions.
3. *Wide gaps between related information*. Users might not identify the relationship between related contents or related information when the design of the Web sites wrongly separates them. This leads to difficulties in interaction.

[21] D. Rømen, and D. Svanæs. (2008). *Evaluating web site accessibility: validating the WAI guidelines through usability testing with disabled users*. In Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges (NordiCHI '08). ACM, New York, NY, USA, 535- 538.

[22] D. Rømen, and Svanæs. (2012). *Validating WCAG versions 1.0 and 2.0 through usability testing with disabled users*. In Universal Access in the Information Society, 11(4), 375-385.

[23] R. Calvo, F. Seyedarabi, and A. Savva. (2016). *Beyond Web Content Accessibility Guidelines: Expert Accessibility Reviews*. In Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (DSAI 2016), Association for Computing Machinery, New York, NY, USA, pp. 77–84.

4. *Use of custom components*. Custom components are small, self-contained elements of code that can be re-used when building widgets. Often, these components are not explained properly for screen reader users or cannot be interacted using common keys, such as “Enter” or the space bar.
5. *Buttons and text size*. Users may face challenges in identifying elements as buttons, selecting and pressing them, or reading text on a Web site, especially if the responsive page is adapted from desktop to mobile viewports.
6. *Color contrast ratio between icons and background*. In order to avoid accessibility barriers to users with visual impairments, WCAG standard states that the color contrast ratio between background and foreground should be higher than 4:5:1. However, they do not specify whether it is the case also of interactive icons, buttons, elements not including text, and their surroundings.
7. *Placement of important information*. WCAG guidelines do not specify how information should be structured and ordered in a Web page. Yet, since users, especially screen reader ones, focus on the information shown at the top of the page, it is there that the important information should be placed.

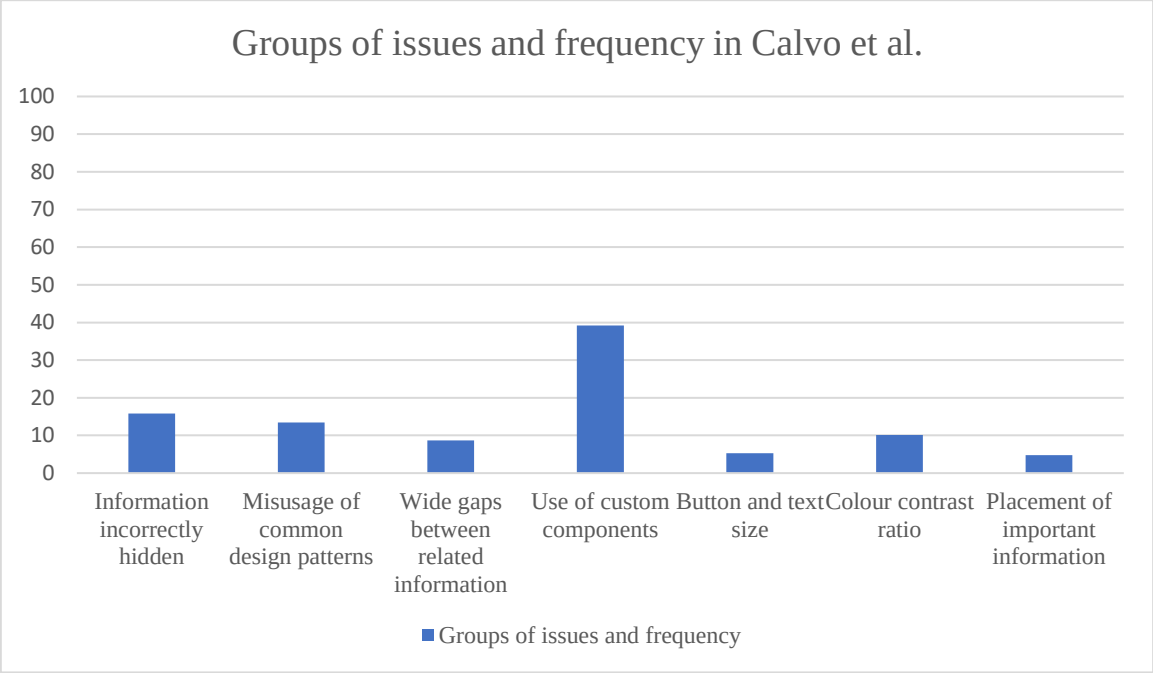


Figure 2.1. Groups of issues and frequency.

2.3 The almighty dollar: a barrier to accessibility?

Difficult to employ and inadequate to provide full accessibility, WAI guidelines must also face one of the biggest obstacles in their implementation: the almighty dollar. The cost of developing accessible Web content is, indeed, more than substantial, no matter how noble the cause behind it is. A small segment of the population may not sufficiently translate into significant economic benefits for those addressing their specific needs.

Or might it?

It is estimated that people with disabilities actually dispose of higher-than-average incomes[24], denser and broader social networks[25], and can act as influencers in their communities. Furthermore, by its very nature, accessibility enables for long-term savings: Sierkowski's study[26] investigates precisely this longstanding economic benefit with respect to U.S. public schools. It emphasizes how, as Web accessibility is based on design optimization strategies, it allows management of on-screen information in a more cost-effective way: mere migration of content from technology to technology, for example, is more cost-efficient than older designs which blend content with formatting.

Goggin and Newell's study[27], moreover, states that inclusive technology is indeed profitable as well as socially desirable: inaccessible final products translate to the marketplace as lost business opportunities. This topic of research was delved into in great depth by the 2019 Click-Away Pound Report[28], which revealed that nearly three-quarters of online users with disabilities (69%) abandon websites they find difficult to interact with because implemented with weak or no accessibility strategies. As a result, by failing to satisfy the needs of online

[24] A. Zaidi, and T. Burchardt. (2005). *Comparing incomes when needs differ: equivalization for the extra costs of disability in the UK*. In Review of income and wealth, 51(1), 89-114.

[25] T. Burchardt. (2004). *Capabilities and disability: the capabilities framework and the social model of disability*. In Disability & society, 19(7), 735-751.

[26] B. Sierkowski. (2002). *Achieving web accessibility*. In Proceedings of the 30th annual ACM SIGUCCS conference on User services (pp. 288-291).

[27] G. Goggin, and C. Newell. (2007). *The business of digital disability*. In The Information Society, 23(3), 159-168.

[28] R. Williams, and S. Brownlow (2020). *The Click-Away Pound Report 2019*. Online available at https://clickawaypound.com/downloads/cap19final0502.pdf. Last accessed Sept. 30, 2021.

customers with disabilities, retailers risk losing out on approximately £17.1 billion in annual online sales (the study was conducted in UK).

In order to be economically sustainable, accessibility should be taken into account at every stage of software development, from the initial idea to the architectural interface and content design, from the Web site construction to the evaluation and maintenance. In their work[29], Schulz and Fuglerud have studied how the use of personas, rich descriptions of potential users with well-defined biological and psychological traits, must necessarily include those representing users with disabilities as well. Creating personas with disabilities that consider the different assistive technologies and employing them from the beginning reduces costs much more than considering them as a final product process.

As might be expected, however, the greatest expense in terms of accessibility Web development is in the implementation of accessible technologies. In their study[30], Puzis et al. have stressed the cost of WAI-ARIA implementation to be its greatest weakness. They have therefore proposed an affordable version of WAI-ARIA, obtainable by extending the specification and at the same time by refining it in order to reach its best quality-ratio.

Other approaches to cost reduction have been proposed by the use of a wide range of technologies: from ontologies[31] of accessibility requirements, to dynamic injections[32] of WAI-ARIA into web content, from semi-automatic evaluation processes[33], to semi-automatic Web

[29] T. Schulz, K. Skeide Fuglerud. (2012) *Creating Personas with Disabilities*. In Miesenberger K., Karshmer A., Penaz P., Zagler W. (eds) Computers Helping People with Special Needs. ICCHP 2012. Lecture Notes in Computer Science, vol 7383. Springer, Berlin, Heidelberg.

[30] Y. Puzis, Y. Borodin, A. Soviak, V. Melnyk, and I. V. Ramakrishnan. (2015). *Affordable web accessibility: a case for cheaper ARIA*. In Proceedings of the 12th International Web for All Conference (W4A '15). Association for Computing Machinery, New York, NY, USA, Article 32, 1–4.

[31] K. R. Masuwa-Morgan and P. Burrell. (2004). *Justification of the need for an ontology for accessibility requirements (Theoretic framework)*. In Interacting with Computers, vol. 16, no. 3, 2004, pp. 523-555.

[32] A. Brown and S. Harper. (2013). *Dynamic injection of WAI-ARIA into web content*. In Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13), Association for Computing Machinery, New York, NY, USA, article 14, pp. 1–4.

[33] M. Naftali. (2010). *Analysis and integration of web accessibility metrics*. In Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A) (W4A '10), Association for Computing Machinery, New York, NY, USA, article 6, 1.

site repair tools[34]. In the wake of these efforts, this dissertation suggests an innovative tool that enables non-disabled inexperienced developers to visualize "on the fly" the accessible quality of their product, within their workflow.

Fortunately, despite the high cost of development, the Web is not entirely insensitive to these concerns: examples[35] of collaborative web improvement by volunteers do exist and have the potential to make the Web more accessible through free metadata creation, such as the case of the social labeling of the 1997 ALT-server[36]. W3C itself acknowledged all the observations that arose from the scientific community. In July 2020, it published the document "Challenges with Accessibility Guidelines Conformance and Testing, and Approaches for Mitigating Them", to address and understand those challenges more holistically, and explore approaches to mitigating them, especially in large and dynamic Web sites.

[34] C. Asakawa and H. Takagi. (2000). *Annotation-based transcoding for nonvisual Web access*. In Proceedings of ASSETS'00 (Arlington, VA), ACM Press, pp. 172-179.

[35] D. Sato, H. Takagi, M. Kobayashi, S. Kawanaka, and C. Asakawa. (2010). *Exploratory Analysis of Collaborative Web Accessibility Improvement*. In ACM Trans. Access. Comput. 3, 2, Article 5.

[36] D. Dardailler, *The ALT-server. An eye for an alt* (1997). Online available at https://www.w3.org/WAI/altserv.htm. Last accessed Oct. 9, 2021

Chapter 3

Accessibility testing tools

Chapter 2 analyzed how the scientific community consistently found that levels of Web accessibility remain disappointing, due to lack of technical specialization of webmasters, cost issues, inadequacy, weak coverage of errors by accessibility guidelines and difficulties in reading and implementing them. This Chapter completes the discussion by providing an overview of resources that Web authors and developers can leverage to overcome these troubles and improve the accessibility of their Web sites and applications. The above-mentioned resources are the accessibility testing tools, or checkers.

3.1 AChecker, Tenon, Lighthouse, WAVE, axe

Accessibility testing can be performed in three ways: manually, automatically or semi-automatically. The biggest difference between them lies in who executes the test case: in the first case, a human performs it step by step, without test scripts. In the second one, it is conducted automatically by a tool, software, or framework. Finally, in the semi-automatic case, testing tools perform automated operations leaving part of the verification process to the manual tester by means, for example, of editable checklists that must be filled in by the specialist before

the automatic test can restart. According to researches[37,38,39], manual testing requires analysts and developers to be highly involved in managing complex scenarios in a more time-consuming and tedious dynamic. However, their greater efficiency in the quality and depth of analysis is demonstrated against any of the various automated tools available on the market. Additionally, even if automated evaluation tools quickly identify potential accessibility issues, too often they can produce incorrect or misleading outcomes.

W3C itself provides a comprehensive list of accessibility testing tools and a helpful document ("Selecting Web Accessibility Evaluation Tools") that outlines their features and discusses different aspects to consider when choosing them. Regarding their characteristics, testing tools can be selected according to different aspects:

- *Compliance with specific guidelines*. Since the fact that there are many accessibility standards required by organizations and governments (WCAG being the most significant), different accessibility evaluation tools have been developed to comply with different accessibility standards.
- *Conformance with rating levels*. Some checkers, rather than offering compliance with specific guidelines, test whether the Web site or application meets the three levels of success criteria (A, AA, AAA).
- *Language*. Evaluation tools support different languages, not only in the interface and in the commands of the tools themselves, but also in the tested Web content, which must be linguistically accessible to everyone and, thus, without language barriers.
- *Supported formats*. Different testing tools check the accessibility of different Web technologies, from the basic HTML content to WAI-ARIA, CSS, SVG and PDF.
- *License*. As one might expect, some of the testing tools are licensed and some are open source.

---

[37] G. Brajnik. (2008). *Beyond conformance: The role of accessibility evaluation methods*. In Proceedings of the 2008 international workshops on Web Information Systems Engineering, WISE '08, pages 63–80.

[38] C. C. Shelly and M. Barta. (2010). *Application of traditional software testing methodologies to web accessibility.* In Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A), W4A '10, pages 11:1–11:4.

[39] M. Vigo, J. Brown, and V. Conway. (2013). *Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests*. In Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13). Association for Computing Machinery, New York, NY, USA, Article 1, 1–10.

- *Presentation of results*. They can provide in-page feedback, reports to determine the conformance of the checkpoints with temporary icons inserted in the Web content, or complete transformations of the overall appearance of a site to help identify design issues.

The sustained growth of Web development standards and technologies never ceases to raise a demand for continuous improvement of accessibility assessment tools to provide consistent results. For this reason, they are available on the market in various fashions: as web services, browser extensions[40], built-in or frameworks.

One of the most used testing tools provided as a web service is AChecker[41], released in 2010. It allows developers to test a Web page by entering its url, uploading it as HTML file or directly typing parts of its code in an editable textbox. Before the testing, it is possible to enable the HTML and/or CSS Validator, and to select the guidelines against which the site is to be checked: among others, WCAG 1.0, WCAG 2.0 (both in their three levels, A, AA and AAA), Section 508 and Stanca Act.

Finally, the developer can decide whether the report format should be organized according to the lines of code containing them, or according to the unfulfilled principles of the chosen guideline.

[40] A browser extension is an addition or plug-in that extends the functionality of the browser itself. It exploits web technologies such as HTML, JavaScript and CSS with the user's permission. Further discussion is provided in Chapter 5.1.

[41] AChecker, 2011. Online available at https://achecker.achecks.ca/checker/index.php. Last accessed Sept. 26, 2021.

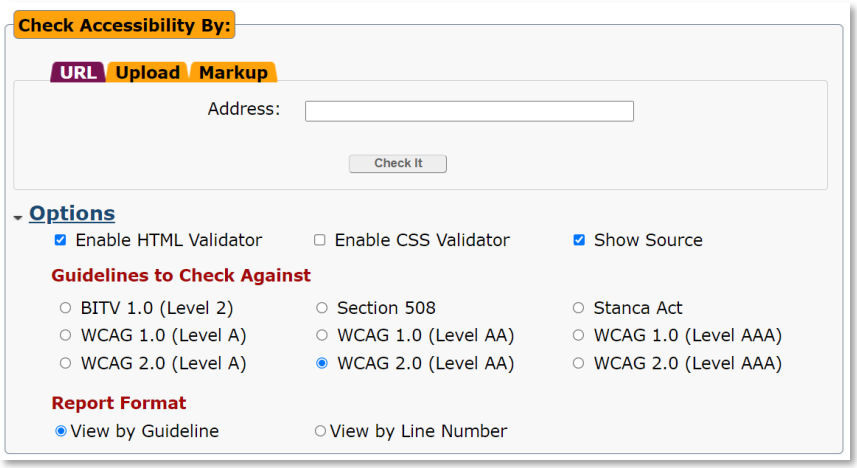


Figure 3.1. The AChecker launch window.

AChecker produces a report of all the accessibility problems it encountered, together with the lines of code to be changed and the unfulfilled principle of the guideline chosen for reference. Specifically, it identifies three types of problems:

- *Known problems*: certain accessibility barriers that must be fixed;
- *Likely problems*: issues that require a human to make a decision;
- *Potential problems*: issues not identifiable by AChecker, which may even be marked as irrelevant by the human developer. For example, it may return an image with an empty alt (`alt=""`), but the image itself can be decorative.

As an open-source, AChecker is a valid tool although not exempt from being flawless: the interface is strongly outdated and erroneous lines of code are listed one after the other without links to the full code, making manual changes extremely tedious.

Another automated testing tool available as a web service is Tenon[42]. Its home page, developed with more appealing graphics, already presents it as a more innovative tool. It identifies WCAG 2.1 issues, in addition to WCAG 2.0 ones, it supports training and consulting (in-person or online), and among its customers big names such as Mastercard and Delta Airlines stand out. Tenon provides a free accessibility consultation by entering the url or HTML code of the site to be tested and also comes in the form of a browser extension, i.e. a plug-in which extends the functionality of the browser itself. As can be seen in Figure 3.2, after analyzing the

[42] Tenon, 2017. Online available at https://tenon.io. Last accessed Sept. 26, 2021.

provided Web site (in this example, the Louvre Museum Web site[43]), Tenon provides a pie chart showing the error rates by success criteria. For the Louvre Museum website, it reported 163 issues.

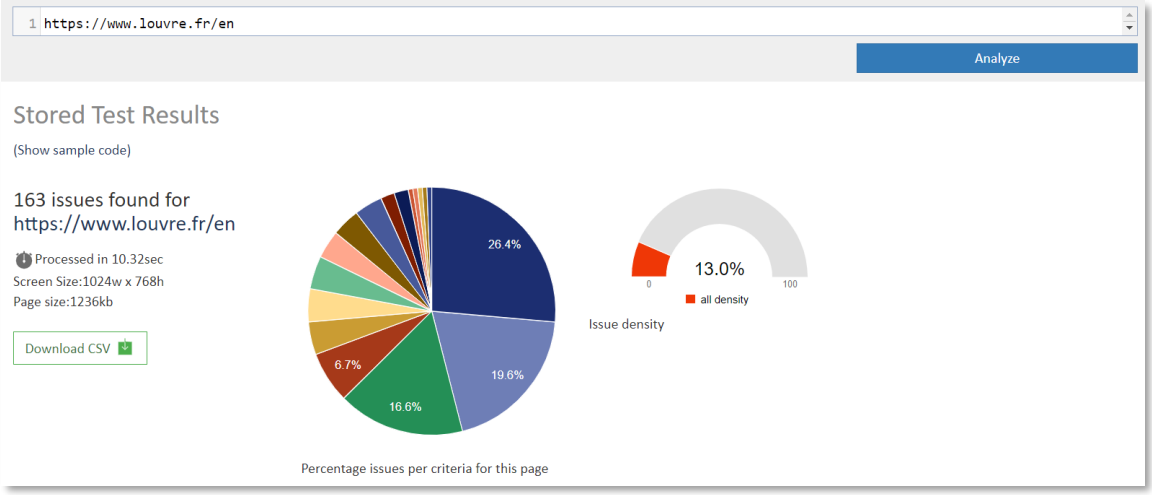


Figure 3.2. Tenon test results on Louvre Web site.

Which is interesting, since for the same Web site AChecker didn't report one (Figure 3.3).

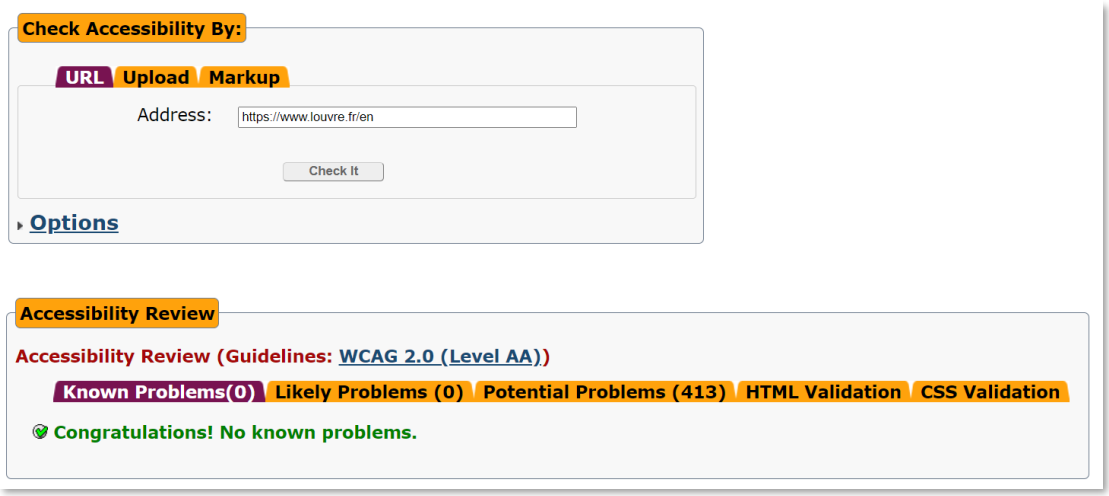


Figure 3.3. AChecker test results on Louvre Web site.

[43] Louvre Museum Official Website | Homepage. Online available at https://www.louvre.fr/en. Last accessed Sept. 26, 2021.

The logic of Tenon accessibility report is similar to that of AChecker: it divides issues in *Errors* and *Warnings* and for each of them it displays a code snippet that allows the developer to find it in the code of the Web page, along with the WCAG guideline it pertains to. But Tenon goes further: it also provides a link to a recommended fix for every issue. For instance, the error "*This list item has no meaningful content*" depicted in Figure 3.4 cites "*Web Content Accessibility Guidelines (WCAG) 2.1, Level A: 1.3.1 Info and Relationships*" as reference guideline and related success criterion.

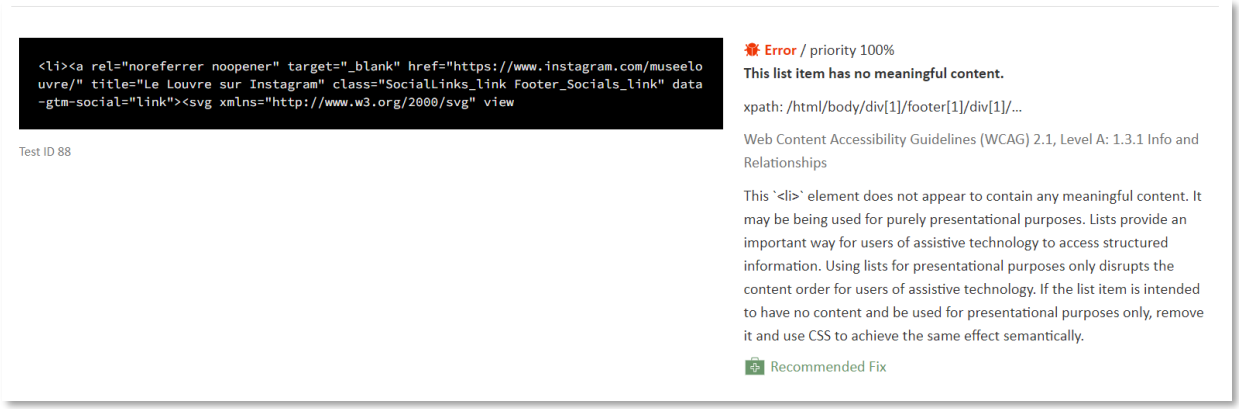


Figure 3.4. Tenon test results on Louvre Web site: example of an issue.

Under a brief description of the error, the green button "*Recommended Fix*" provides a link to a Tenon Web page (Figure 3.5) that explains a Best practice principle suitable for the error in question. Specifically, it suggests using list markup for lists and making sure it is appropriate for the content being conveyed. It also adds user impact (in this case "medium"), repair rate ("low"), interface impact ("low"), and population being impacted ("cognitively impaired", "low vision", "blind").

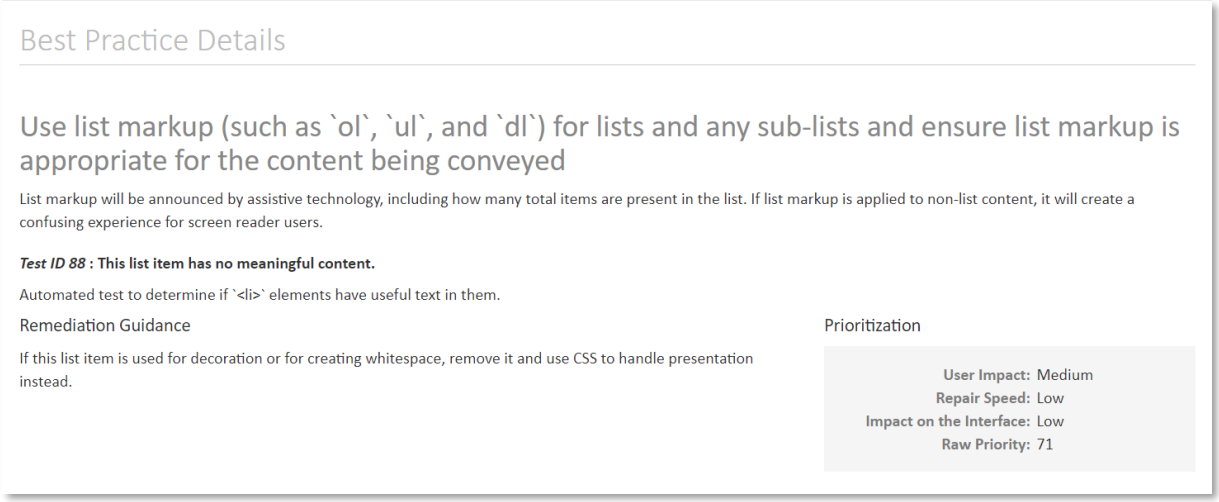


Figure 3.5. Tenon test results on Louvre Web site: example of the "Recommended fix" of an issue.

Tenon also offers a sophisticated dashboard where the developer can save and retrieve the results of all of his or her testing, percentages of types of errors over time, and more. For all of the above reasons and features, it proves to be a phenomenal tool for automated accessibility testing. However, in its documentation[44], its own developers state that it is impossible to provide complete coverage of the WCAG Success Criteria and that, indeed, no automated testing tool can guarantee it. They stress that Tenon is an important component in the development of accessible Web sites and applications, but that it should not replace manual testing, as some accessibility best practices are either too subjective or too complex to be accurately tested and could lead to false positives.

Automated accessibility tools are also available in the form of browser extensions, which is the simplest application of these tools. By simply browsing Web pages, they generate reports that flag accessibility issues, describe them and give solutions on how to fix them. In 2017 Google developed Lighthouse[45], an open-source Chrome extension, that is applicable not only to accessibility, but also to the Performance, SEO (Search Engine Optimization), PWA (Progressive Web Apps) and Best Practices. Lighthouse is configured as a very easy-to-use

[44] Tenon, *Documentation*. Online available at https://tenon.io/documentation/what-tenon-tests.php. Last accessed Sept. 26, 2021.

[45] Lighthouse, 2018. Online available at https://developers.google.com/web/tools/lighthouse. Last accessed Sept. 26, 2021.

tool: after opening the Web site to be tested, it is sufficient to click on the only button in the Lighthouse pop-up window to generate the accessibility report.

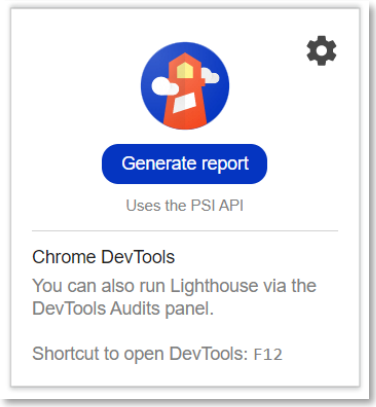


Figure 3.6. Lighthouse browser extension.

In this report, unlike those of AChecker and Tenon, the errors are organized under categories, making it easier to understand how to solve them. The categories, for example, are "*Contrast*", "*Names and labels*", "*Navigation*", etc., and under each of them the lines of code to be fixed are listed. In addition, the report scores the website from 1 to 100 and provides suggestions with links to detailed Google support pages.

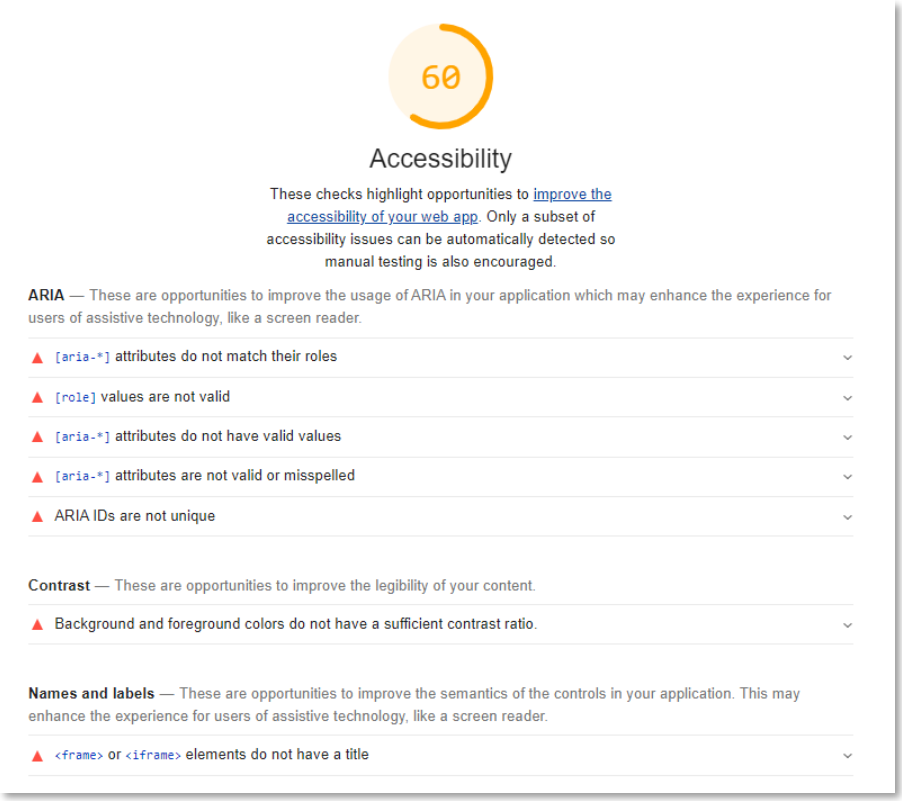


Figure 3.7. An example of a Lighthouse accessibility report: The Wall Street Journal Web site.

Since the software is directly featured Chrome DevTools, erroneous code snippets are highlighted in context and it is therefore much more immediate to correct them. This is not its only advantage: owned by Google, Lighthouse ensures constant updates and a great flexibility of use, and, unlike AChecker and Tenon, it also validates failures against the WAI-ARIA specification. However, Lighthouse doesn't allow for the refinement of testing options (it is not possible to decide whether to test compliance with WCAG 1.0, 2.0 or 2.1 guidelines) and, in general, it detects fewer accessibility issues than Tenon. This is the case of the Louvre Web site, that is inaccurately rated with a 100% score.

Still in the frame of browser extensions, one of the best testing tools currently available is WAVE[46]. Free to use, it operates on Firefox and Chrome and is also available as a Web service, where the developer has to enter the url of a desired Web page. Unlike the ones previously analized, WAVE is extremely interactive and dynamic. Once activated, it processes the tested Web page in real time, rendering it with embedded icons and indicators. Its interface is intuitive,

[46] WAVE, 2019. Online available at https://wave.webaim.org/. Last accessed Sept. 26, 2021.

and each reported error is accompanied by detailed information about the nature of the problem and its possible solution. For this reason, the tool is most effective when used by a senior Web developer with deep knowledge about Web accessibility, but it can also be effectively enjoyed by those who are not experts in accessibility.

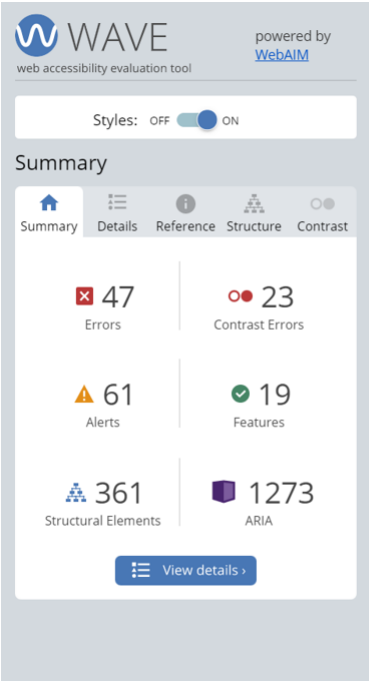
The first applicable functionality featured in the WAVE interactive screen is the ability to disable page CSS styles, whose layouts can be difficult to read after the WAVE icons have been applied. In this way, the developer can easily grasp the navigation order with which keyboard-only and screen-reader users browse the page. After the testing, WAVE provides an overview of found accessibility issues and features, dividing them into:

- *Errors*: red icons indicating errors that need to be fixed;
- *Contrast errors*: same for the one above, but regarding contrast color issues;
- *Alerts*: orange icons for potential problems that require human scrutiny;
- *Features*: green icons that do not point to problems but rather indicate accessibility features that would likely improve accessibility;
- *Structural Elements*: as the name goes, they are the structural elements of the page, for example headings, lists, navigation, aside and footer elements.
- *ARIA*: purple icons that evidence ARIA roles, states, or properties (for example, an `aria-label` attribute or an element with `role="button"`).

It is worth mentioning that the latter feature was already deployed by Visual ARIA[47] Chrome extension, a simpler accessibility viewer that physically rendered, and still renders, the invisible use of ARIA within Web technologies. However, its visualization is messy and cumbersome and, therefore, not very advisable.

In the WAVE command window, the *Details* tab lists the above mentioned features grouping them by problems (e.g. "*Empty heading*" or "*Broken ARIA link*"): clicking on the available icons redirects directly to the location on the Web page that presents the problem. Moreover, the *Reference* tab provides useful insights about what it means, why it matters and for which users, how to fix it, and what standards and guidelines cover it.

[47] Visual ARIA, 2017. Available at https://github.com/accdc/visual-aria. Last accessed Sept. 29, 2021.



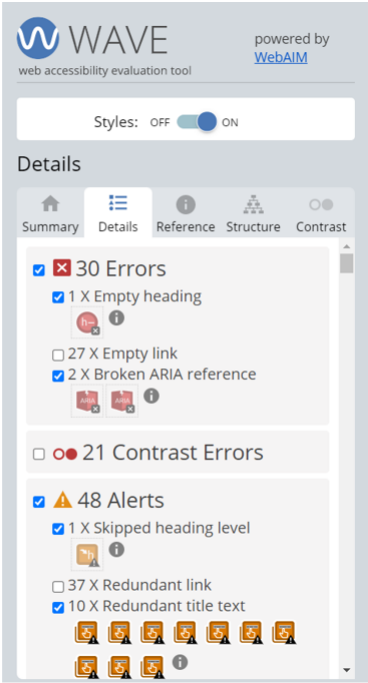


Figure 3.8. Summary of WAVE report.

Figure 3.9 Details of WAVE report.

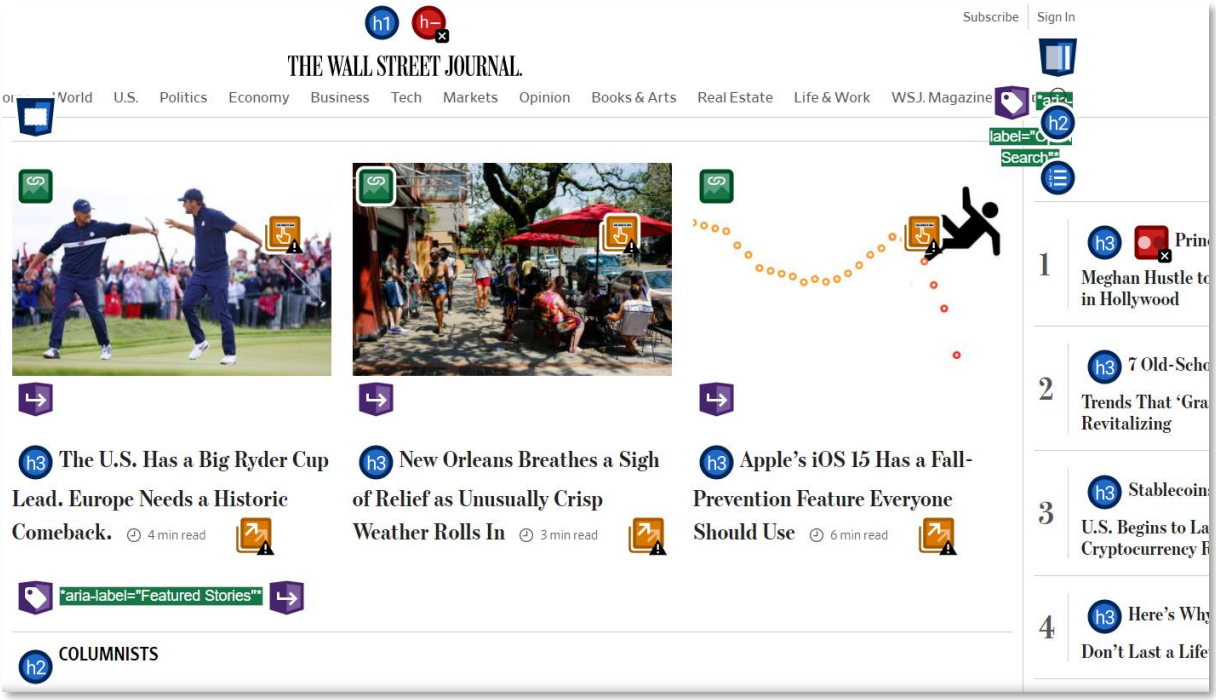


Figure 3.10. WAVE interactive interface and icons on The Wall Street Journal Web site[48].

[48] The Wall Street Journal Official Web site | Homepage. Online available at https://www.wsj.org/. Last accessed Sept. 26, 2021.

To view code relevant to a particular WAVE icon (be it an error, alert, feature, etc.), the developer can simply click on the *Code* tab at the bottom of the page. A window showing the underlying markup/DOM will open and the line in consideration will be highlighted, making it immediately identifiable.

However, WAVE lacks an important feature, which if implemented would improve it exponentially: its performance time is inserted at page loading in the browser, and not in the life cycle of user interactions. In other words, testing begins when the loading process is completed and there it ends, even if the existing content is modified. This does not reflect the real behavior of the page, and therefore limits WAVE to a thorough but at the same time superficial and theoretical evaluation. For instance, let's test a simple checkbox from the "WAI-ARIA Authoring Practices 1.1" example[49] (Figure 3.11).

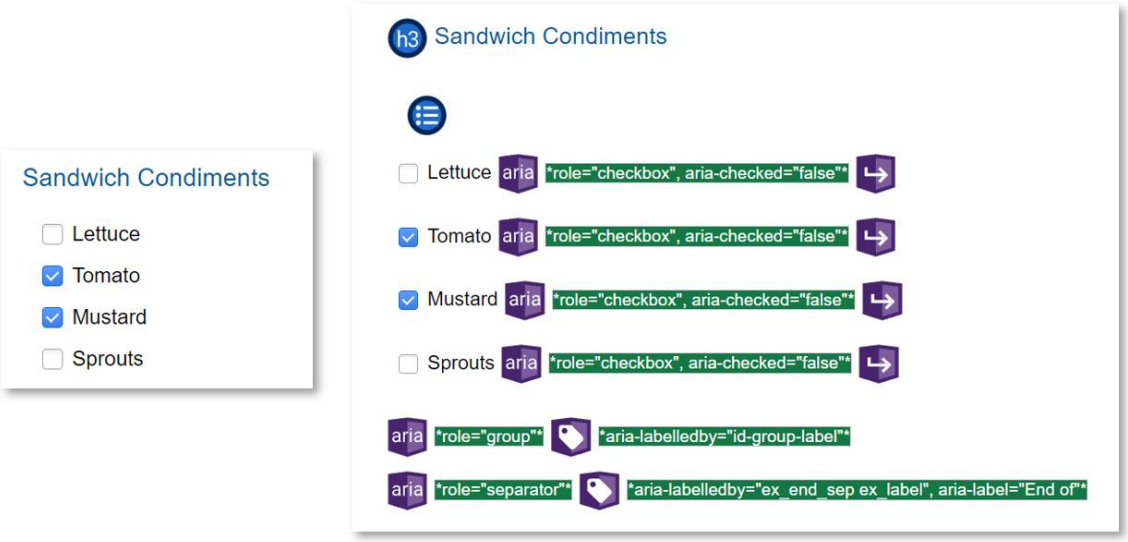


Figure 3.11. The checkbox example from "Authoring Practices". On the right, its WAVE rendering.

Purple icons successfully identify ARIA elements, indicating their role (`role="checkbox"`) and state (`aria-checked`). Intuitively, the latter is `false` when the checkbox is not checked. However, even when it is selected, the state does not dynamically

[49] *WAI-ARIA Authoring Practices 1.1*. (2019). Available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/checkbox/checkbox-1/checkbox-1.html. Last accessed Sept 29, 2021.

switch to `true` (Figure 3.12). An inexperienced developer who relies solely on this automated tool might mistakenly assume that an accessible issue lies in the JavaScript implementation or in the ARIA notation itself, when in fact it merely is a WAVE shortcoming.

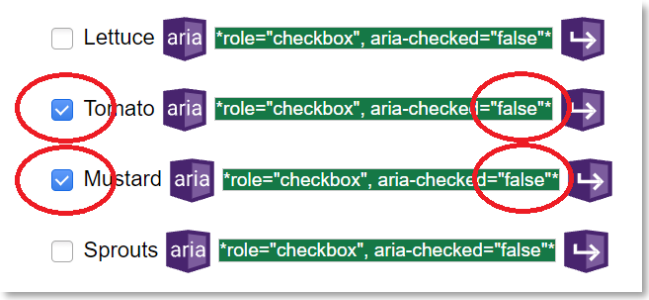


Figure 3.12. The failed WAVE update of "`aria-checked`".

More recently, automated testing tools have been proposed as frameworks to enable test-driven development. Nowadays, axe[50] is the world's leading digital accessibility toolkit. Chosen by Microsoft, Google, development and testing teams, it proposes a wide range of tools:

- axe DevTools: a comprehensive accessibility testing tool with automated and guided testing that inserts into CI/CD (continuous integration/continuous delivery) flows for consistent unit testing across teams;
- axe Auditor: a step-by-step manual accessibility testing tool that performs full-coverage, consistent WCAG audits of Web content by offering detailed issue descriptions, remediation and expected behavior advice;
- axe Monitor: a tool that dynamically scans, monitors and reports on the accessibility status of an entire Web site or PDF, pinpointing improvements and downfalls.

The first one, axe DevTools, is the only one that comes in a free lightweight form and as a browser extension for Chrome, Firefox and Edge. In its Pro version it proposes iOS and Android mobile testing, exporting, saving and sharing issues, enterprise support, and many other features. It promotes accessibility as an integral part of the regular development routine and it assures[51] at least 80% issue coverage during development with no false results, but only errors

[50] Axe, 2020. Online available at https://www.deque.com/axe/. Last accessed Sept. 26, 2021.

[51] Axe DevTools. Online available at https://www.deque.com/axe/devtools/. Last accessed Sept. 26, 2021.

with 100% certainty. The developer accesses the free version of axe DevTools opening the developer console (Ctrl+Shit+i or F12), as shown in Figure 3.11. Here, he or she can decide to scan an entire web page or only a part of it. As an example, once again the homepage of The Wall Street Journal is tested.

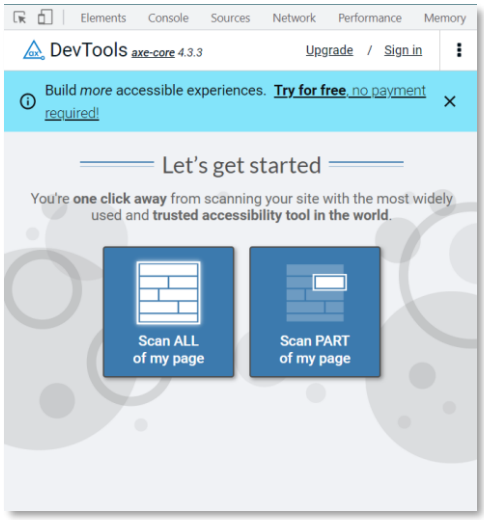


Figure 3.13. The starting interface of axe DevTools (free version).

Similarly to the other checkers previously reported, this review of issues divides them into:

- *Automatic issues*; and
- *Guided issues*, which require human intervention to verify that they are actual problems or not.

Based on the impact they have on users with disabilities, they are divided into "critical", "serious", "moderate" and "minor". For example, "*ARIA roles must conform to valid values*" is a critical error, while "*Users should be able to zoom and scale the text up to 500%*" is considered a minor one.

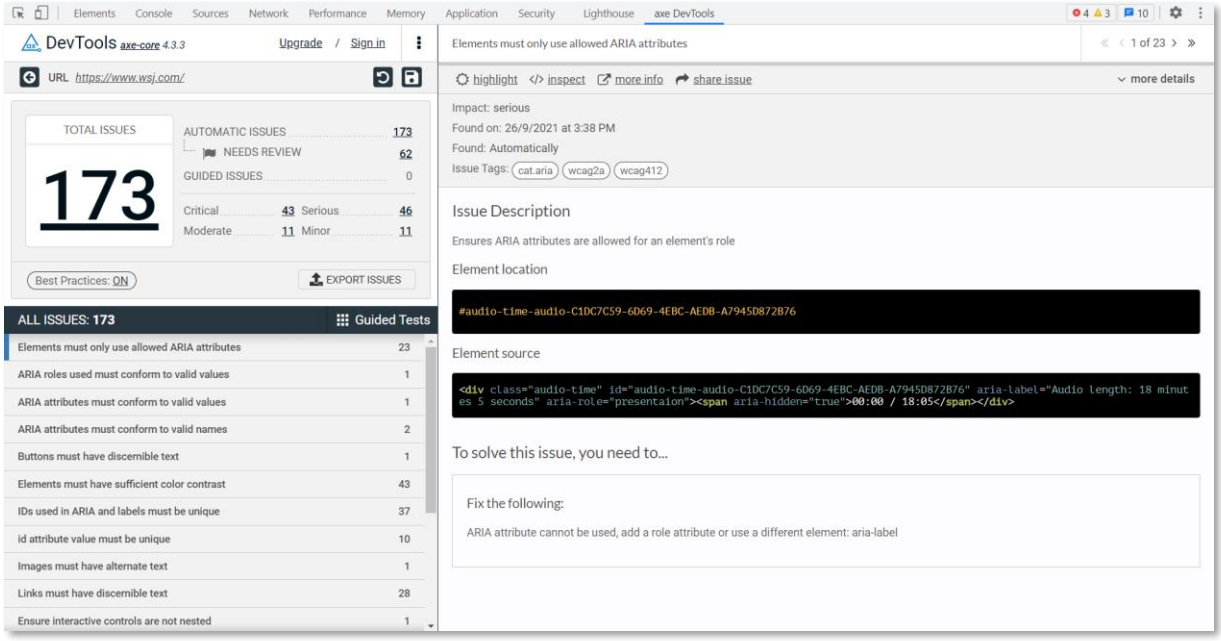


Figure 3.14. axe DevTools accessibility report on The Wall Street Journal Web site.

Furthermore, DevTools indicates which guidelines cover the error (e.g. WCAG 2.0, Section 508, etc.), proposes a suggestion to solve the issue, and provides the source of the item, showing the highlighted line of code. But that's not all: hitting "< /> *Inspect*" will open the complete HTML code of the Web page, showing the outlined culprit element in context.

Its most powerful and innovative features, however, are limited to the Pro version, and significantly exceed those of all the other checkers available in the market.

3.2 Reliability is not an option

Chapter 3.1 presented an overview of some automated testing tools, different in functionality, features and sometimes provided in various flavors to meet specific requirements in different Web development workflows. However, as already pointed out beforehand, no automated tool can equal a human tester in terms of quality and depth of the analysis. And it doesn't end here. The effectiveness and reliability of the testing tools themselves is yet to be proven, let alone the performance.

Numerous studies have been undertaken to shed light on the issue. Brajnik conducted one of the first inquiries[52] in this domain: he analyzed the effectiveness of two automated tools, BOBBY and Lift Machine, with "effectiveness" comprising the qualities of completeness, correctness and specificity. He limited the study to *fault identification*, i.e., the task of supporting the Web developer in identifying violations of selected guidelines, and *fault diagnosis*, i.e., the task of associating them with their possible causes. He did not consider the other functions that the testing tools could provide, such as support in fixing the errors. Results produced by LIFT Machine were at 22% false positives and 11% false negatives. BOBBY, on the other hand, produced 32% false positives and 26% false negatives. In both cases, the results were far from being satisfying.

On the basis of this survey, a more recent study[53] by Vigo, Brown and Conway investigated the effectiveness of current automated evaluation tests applying the metrics of:

- *Coverage*: the number of distinct success criteria that can be detected and described by the tools. The larger this set, the more specific alerts and suggestions the tool can supply;
- *Completeness*: the ratio of reported violations over the actual number of violations. Completeness is achieved by reducing the number of accessibility issues the tools didn't catch (false negatives);
- *Correctness*: how well tools minimize the number of mistakenly reported accessibility violations , i.e. statements of detected problems that are not true (false positives).

They selected three Australian government Web sites with Ajax updates, high levels of interactivity and multimedia content, and they evaluated their conformance to WCAG 2.0 by consulting a blind expert user. Consequently, they chose six state-of-the-art evaluation tools (AChecker, SortSite, Total Validator, TAW, axe, AMP) that could test Web pages against the WCAG 2.0 guidelines. In terms of *Coverage*, automated testing tools were only able to cover 23-50% of the violated success criteria. Moreover, they found only 14-38% of the actual

[52] G. Brajnik. (2004). *Comparing accessibility evaluation tools: a method for tool effectiveness*. In Universal Access in the Information Society, 3:252{263.

[53] M. Vigo, J. Brown, and V. Conway (2013). *Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests*. In Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility (W4A '13). Association for Computing Machinery, New York, NY, USA, Article 1, 1–10.

violations identified by the experts, making the *Completeness* of the tools weak as well, with a very high average false negative rate of 62-86%. Only the *Correctness* levels of the tools showed promising results, with a range between 93-96% except the case of TAW and TotalValidator, which performed 71% and 66% respectively. This meant that 1 out of 3 errors reported by them were false positives, and thus, unreliable. Details of the results can be seen in Figure 3.15 and Figure 3.16.

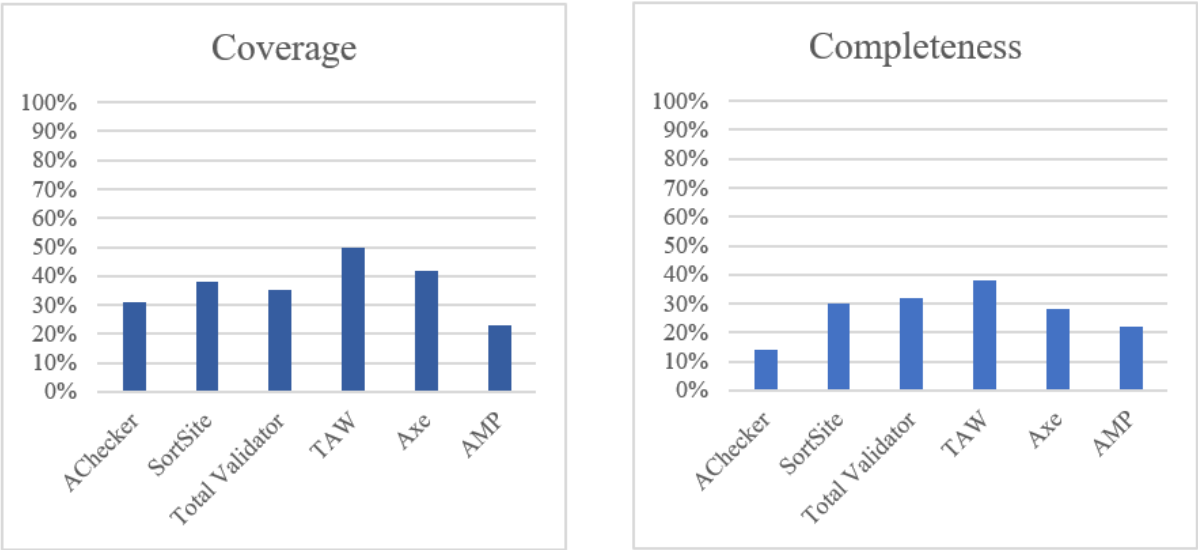


Figure 3.15. Coverage and correctness of the six automated testing tools.

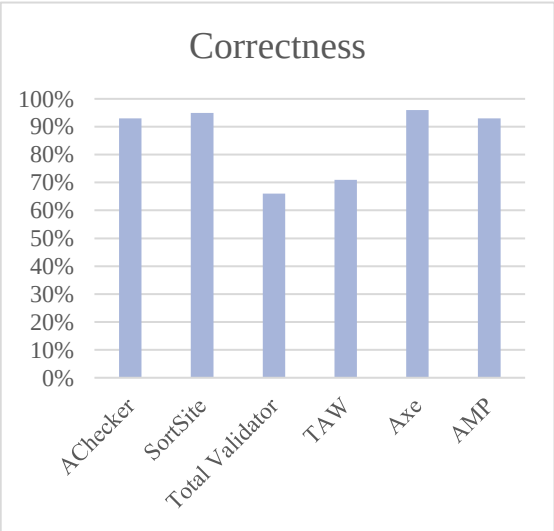


Figure 3.16. Correctness of the six automated testing tools.

Figure 3.17 groups the three metrics according to each of the accessibility tools. In this way, axe and TAW visibly stand out among all the others as the top ones a developer should choose. In conclusion, the study reported at least one accessibility violation in *all* WCAG 2.0 principles and therefore, based on its findings, ends with a strong suggestion to employ human intervention in usability testing and not rely solely on commercially available tools.

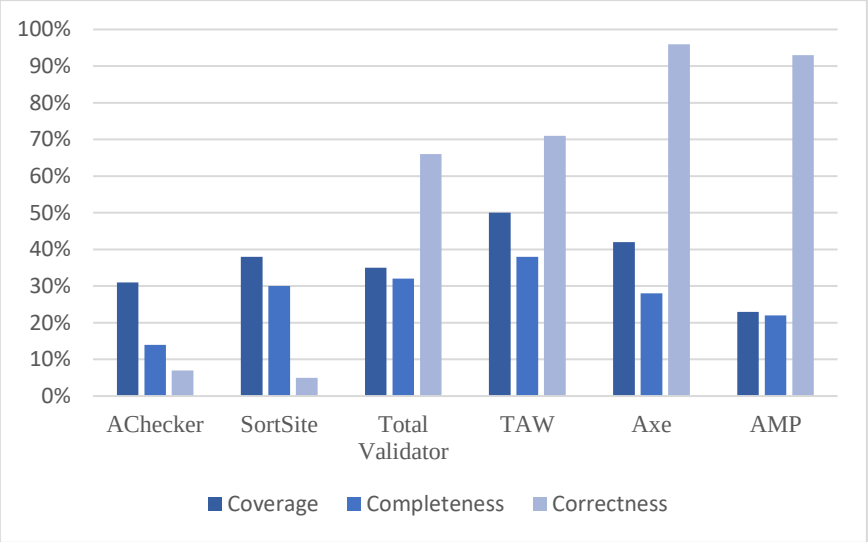


Figure 3.17. Results of Vigo, Brown and Conways' study on six accessibility evaluation tools.

Lastly, when comparing automated accessibility testing software, there's one more critical issue to keep in mind. Nowadays, the majority of Web pages are being generated and modified dynamically client-side thanks to JavaScript, resulting in a runtime Document Object Model (DOM) that differs from the one of the static source code. As a result, automated checkers must be able to interact with the actual DOM of the page, rather than simply the original source code, or else numerous accessibility issues, including severe ones, would result in being missed.

From these findings, it is evident that the market would greatly benefit from a tool that, leveraging the top of existing web technologies, integrates automated accessibility testing and the manual efforts of developers, designers and content authors, in specific and innovative ways. Altering the visual appearance of the page and the method of navigating or interacting with it (e.g., disabling the mouse) can be the key to reducing the disability of sighted designers, who are not naturally aware of how their product will actually be perceived by users with disabilities. The suggested approach will be introduced and discussed in the following Chapters.

Chapter 4

Problems for the sighted developers

4.1 Unhelpful eyes: the accessibility role reversal

Guidelines, rules, principles, legislations, community support and testing tools: nowadays, it seems that the Web provides developers and authors with many ways to create content perfectly accessible to users with various types of disabilities. Yet, the digital divide is still frighteningly broad. In 2019 Deque Systems, the accessibility software company behind axe tools suite, and Nucleus Research, a provider of technology research services, conducted a study[54] on hundreds of Web sites and returned daunting results: 70% of them exhibit critical accessibility blockers that don't conform even to a WCAG Level A. A dreadfully high number, which denotes how Web accessibility, a basic human right, is still a goal far from being achieved. The study conducted by Deque and Nucleus Search, however, also highlighted the considerable economic profit that a fully inclusive Web would bring. The ethical-economic dilemma discussed in Chapter 2.3 had not taken into account, for example, the annual $6.9 billion missed opportunity that e-commerce retailers would get if their users did not leave their Web site in favor of a handful of accessible alternatives.

But why are the data on the pivotal problem of accessibility, as old as the WWW itself, still so alarming? Parsed, scrutinized and studied by thousands of experts every year, Accessibility (capital A intended), should by now be part of the standard Web development cycle and seamlessly implemented, with a level of navigation, interaction and content usage optimized to, at least, AA.

[54] *Nucleus Research Note: The Internet is Unavailable* (2019). Online available at https://nucleusresearch.com/research/single/the-internet-is-unavailable/. Last accessed Sept. 30, 2021.

Take the scenario of users with partial or total visual impairment. One of the answers to the above question can be traced back to the demographic that is most heavily involved in Web development, design and engineering: sighted people. Paradoxically, it is their eyesight that represents the biggest impediment that affects the accessible implementation of Web content. In fact, because of this "handicap", they cannot perceive exactly how their Web site or application will be experienced by users with disabilities.

In other words, in the development of accessibility, ordinary roles are reversed: unimpaired people become the bearers of disabilities, while disabled people come to be the normal users to whom Web content is directed.

Several software tools have been developed to reduce the handicap of sighted developers, one of which, the subject of this dissertation, will be described thoroughly in Chapters 5 and 6. These tools are not automated checkers, but accessibility visualizers that can be exploited without prior accessibility knowledge or without consulting additional documentation and external resources. They act directly on Web content by visually transforming it in such a way that developers naturally sense the correctness or incorrectness of their markup and, thus, readily grasp accessibility issues.

Currently, the visualizers available in the online marketplace focus primarily on colors (and the many facets that must be taken into account when designing an accessible Web site), heading structure and keyboard focus order, also known as Tab order. This section delves into each of the three aspects by looking at what tools offer solutions for checking adherence to WAI guidelines.

First of all, as stated by the Color Blind Awareness[55] organization, about 8% of men and 0.5% of women are diagnosed with color vision deficiency (or CVD). Hence, the worldwide amount of people suffering from it amounts to about 300 million. Color is a crucial element in Web design: not only does it allow users to enjoy harmonious images and delightful palettes, it also, and primarily, allows them to a clear recognition of the key information of a Web site: the text. Users with low vision strive to read text that does not contrast with its background. And if they suffer from a color vision deficiency as well, the problem is enhanced. Not perceiving the

[55] Color Blind Awaress | Homepage. Online available at https://www.colorblindawareness.org/. Last access on Sept 29, 2021.

full range of colors, or not perceiving them at all (achromatopsia), indeed, significantly lowers contrasts.

As might be expected, the whole issue of color has been addressed since the earliest debates about accessibility. Today, WCAG 2.1 handles it in three Success Criteria: "1.4.1 Use of Color," "1.4.3 Contrast (Minimum)," "1.4.6 Contrast (Enhanced)," and "1.4.11 Non-text Contrast."

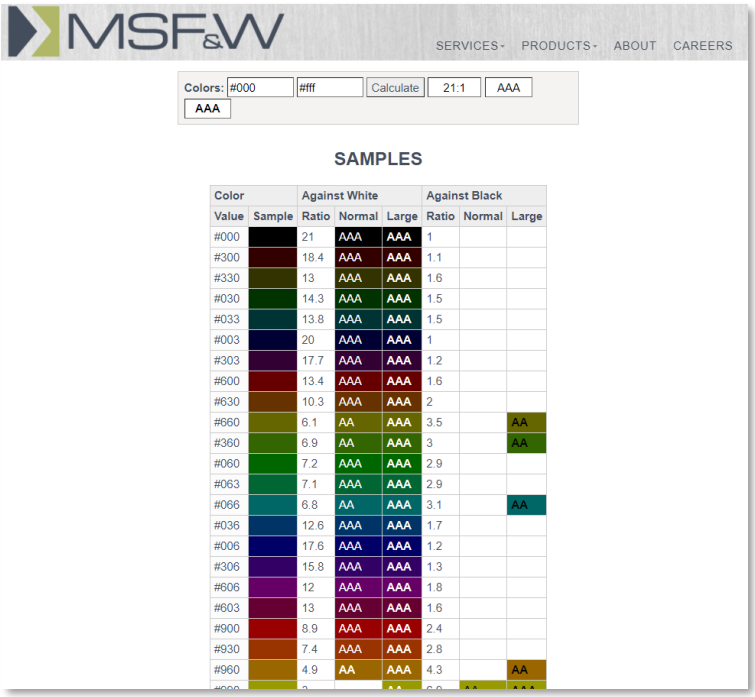
The documentation does not prohibit any specific color or color combination by itself, but sets contrast ratio values, specifically:

- 3:1 minimum contrast ratio for text, images (SC 1.4.3), user interface components, and graphic objects (SC 1.4.11);
- increased contrast ratio of 7:1 for normal text and 4:5:1 for large text (SC 1.4.6).

It also prohibits the use of color alone to present meaningful content or instructions (SC 1.4.1). In the "related resources" following each of the above principles, the documentation also lists a collection of tools that use contrast ratio to analyze the contrast of Web content, such as TPGi's "Color Contrast Analyser"[56] or MSF&W's "Color Contrast Ratio Calculator"[57]. The latter (Figure 4.1) features a tool for calculating the contrast ratio between two colors, and a handy sample of colors with their relative accessibility level (A, AA, AAA) against white and against black background.

[56] TPGi | Color Contrast Analyser. Online available at https://www.tpgi.com/color-contrast-checker/. Last accessed Sept. 29, 2021.

[57] MSF&W | Color Contrast Ratio Calculator. Online available at https://www.msfw.com/Services/ContrastRatioCalculator. Last accessed Sept. 29, 2021.



Color		Against White			Against Black		
Value	Sample	Ratio	Normal	Large	Ratio	Normal	Large
#000		21	AAA	AAA	1		
#300		18.4	AAA	AAA	1.1		
#330		13	AAA	AAA	1.6		
#030		14.3	AAA	AAA	1.5		
#033		13.8	AAA	AAA	1.5		
#003		20	AAA	AAA	1		
#303		17.7	AAA	AAA	1.2		
#600		13.4	AAA	AAA	1.6		
#630		10.3	AAA	AAA	2		
#660		6.1	AA	AAA	3.5		AA
#360		6.9	AA	AAA	3		AA
#060		7.2	AAA	AAA	2.9		
#063		7.1	AAA	AAA	2.9		
#066		6.8	AA	AAA	3.1		AA
#036		12.6	AAA	AAA	1.7		
#006		17.6	AAA	AAA	1.2		
#306		15.8	AAA	AAA	1.3		
#606		12	AAA	AAA	1.8		
#603		13	AAA	AAA	1.6		
#900		8.9	AAA	AAA	2.4		
#930		7.4	AAA	AAA	2.8		
#960		4.9	AA	AAA	4.3		AA

Figure 4.1. MSF&W's "Color Contrast Ratio Calculator".

The real perception of elements with altered colors is inconceivable: our eye cannot be fooled by the mind.

As an example, an unimpaired developer cannot imagine the effects of protanopia, a color blindness causing confusion of reds and greens, such as to unify the lights of traffic lights into a single dark golden-yellow color. A direct accessibility visualizer, therefore, would turn out to be more helpful rather than a tool that calculates ratio contrast formulas. ChromeLens[58], for instance, is a Google Chrome extension that provides visual impairment simulation. In its command window (Figure 4.2), the developer can choose to render the Web page as if a user with CVD were viewing it, by selecting different options:

- *Full blindness*. It makes the page completely black but still usable by moving with the mouse (blindly) or, better, by using the keyboard. It is still possible to interact with the page, for instance, by clicking on buttons and links.
- *Partial blindness* (mild, medium, serious). It blurs the Web page, to render the experience of users with limited vision.

[58] ChromeLens, 2016. Online available at https://github.com/chromelens/chromelens. Last accessed Sept. 26, 2021.

- *Protanomaly*, *Protanopia*, *Deuteranomaly* and *Deuteranopia*. They refer to red-green color vision deficiency.
- *Tritanomaly* and *Tritanopia*. They refer to blue-yellow color vision deficiency.
- *Achromatomaly* and *Achromatopsia*. The first one deactivates most of colors, the second deactivates all of them.

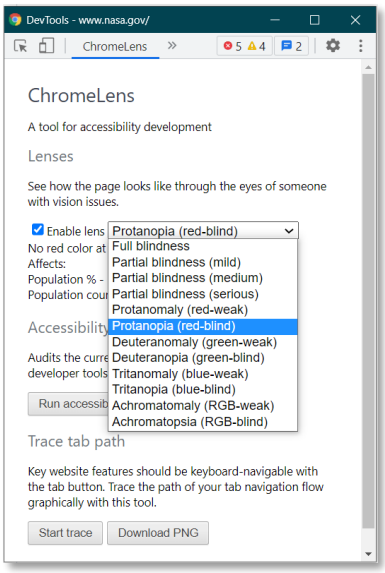


Figure 4.1. ChromeLens command window

Figures 4.2 shows, for example, the homepage of the University of Bologna Web site[59] transformed as if it were perceived by a user with tritanopia, the condition where blue and yellow colors are undistinguishable. Due to correct accessibility implementations, the color switch is noticeable, but it doesn't affect the correct understanding of the page elements.

[59] University of Bologna | Homepage. Online available at https://www.unibo.it/en. Last accessed Sept. 26, 2021.

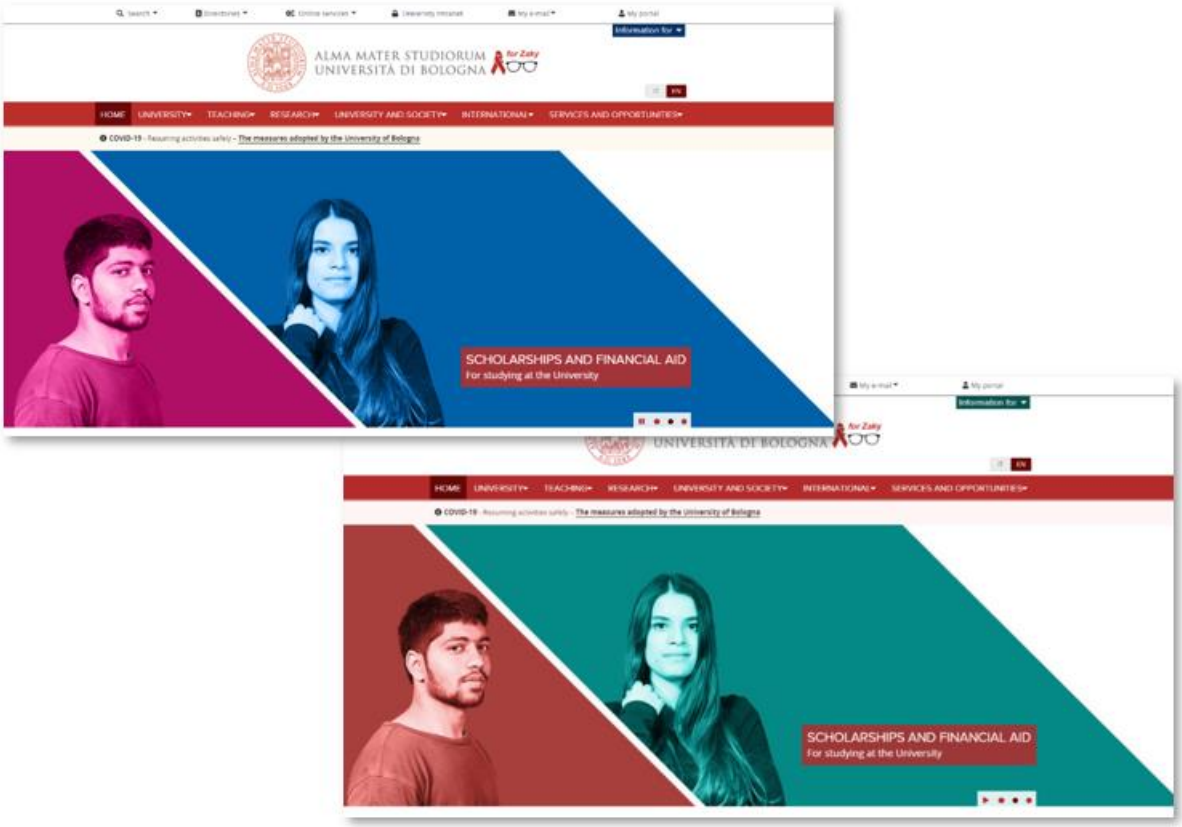


Figure 4.2. Comparison of perception of a normal user and a user with tritanopia: UniBo Web site.

In 2020, Funkify[60] took the legacy of ChromeLens and expanded it, developing a wide-ranging disability simulator. This Swedish software comes in the form of a browser extension that helps developers experience web interfaces through the eyes of personas, with first names and avatars, characterized by different abilities and disabilities. Neat and intuitive, this visualizer provides realistic imitations of:

- *Cognitive impairments*. Funkify's Dyslexia simulator addresses learning disability by scrambling the letters of the text contained in the Web page. As a result, the developer should immediately notice, for example, that serif font texts, smaller than 16 pt and inserted in justified paragraphs, seriously damage the readability of people affected by this condition.
- *Motor disabilities*. Whether it's a mild hand motor impairment or a total inability to control the mouse, Funkify encompasses specific simulators for reduced movement and/or coordination. In the first case, it makes the developer tremble,

[60] Funkify, 2020. Online available at https://www.funkify.org/. Last accessed Setp. 29, 2021.

losing some control over the mouse pointer in the form of trembling. In the second, it emulates a keyboard interface by disabling mouse kyes.

- *Vision impairments*. In addition to the needs of users suffering from color vision deficiency (already addressed by ChromeLens), Funkify allows the developer to experience peripheral vision loss and to increase or decrease the intensity of the blur: nothing but simulating the vision of the about 60% of people who wear glasses or contact lenses.

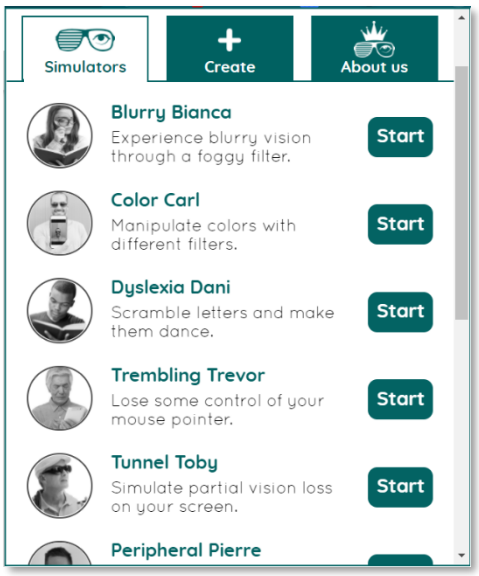


Figure 4.2. Funkify command window

But what if the content is supposed to be invisible? How can a sighted designer perceive it and actively work on it? Accessibility visualizers have also been designed to overcome more sophisticated tasks, such as addressing blind users.

HeadingsMap[61] Chrome extension, for example, shows, browses and audits the headings of a Web page, generating an index where they are listed and clickable, together with a HTML outline tree. It also scrutinizes hidden headers and considers ARIA elements if they feature `role` attribute with heading `value` (`role="heading"`). Their level is based on the value of `aria-level` property and is set to 2 if it's undefined. The open-source NerdeFocus[62]

[61] HeadingsMap, 2018. Online available at https://github.com/dzc34/headingsMap. Last accessed Sept. 29, 2021.

[62] NerdeFocus, 2019. Online available at https://github.com/wizzyfx/nerdeFocusPlugIn. Last accessed Sept. 29, 2021.

Developer Tools extension, on the other hand, overlays an easy-to-follow keyboard focus indicator on the Web page and displays the history with CSS selectors.

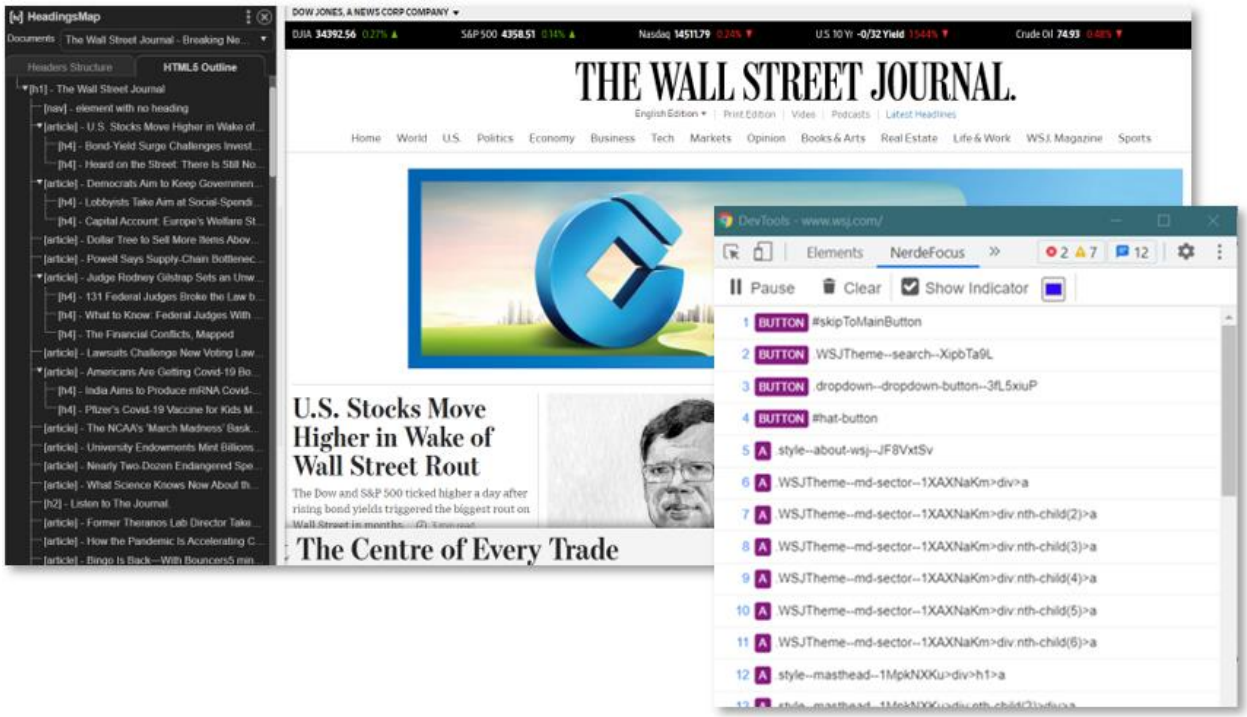


Figure 4.3. HeadingsMap panel (left) and NerdeFocus window (right) on The Wall Street Journal site.

An excellent accessibility visualizer is included in ARC Toolkit[63] as well. Developed in 2017 by TPGi (formerly The Paciello Group, the same company that provides the Color Contrast Analyser mentioned earlier), this free and easy-to-use Chrome extension performs an in-depth testing that shows results both in its panel and in the Web page itself. This evaluation assesses media (Images, BG images, Rich media), structure components (Headings, Landmarks, Lists, Paragraphs, etc.), keyboard access (related to Links, Buttons, Access keys, etc.), color contrasts, IDs and ARIA usage. Each test can be toggled on and off so that different categories of issues can be identified and isolated. Moreover, it provides code-level advice and recommendations to solve WCAG errors and warnings.

[63] ARC Toolkit, 2017. Online available at https://www.tpgi.com/arc-platform/arc-toolkit/. Last accessed Sept. 29, 2021.

As much as sophisticated this multifaceted testing is, its main traits lead to a feeling of déjà vu of the ones already proposed by the more advanced evaluation tools listed in Chapter 4.1, such as Axe and WAVE. The most innovative feature of ARC Toolkit is the explicit graphical illustration of the keyboard focus path, where each interactive element that accept keyboard events is marked as a numbered node and arranged in succession on a neat red line (Figure 4.4). The best practices for a tab order follow the logical reading order (from left to right and from top to bottom): in a Web page, it translates in header first, then main navigation, then page navigation (if present), and finally the footer (Figure 4.5).

This feature, which can be triggered in the "Tab order" panel, allows sighted developer to immediately perceive the end-user experience of people with disabilities who rely on keyboard (or other hardware that mimics the functionality of a keyboard) to navigate the Web.

Figure 4.4. ARC Toolkit's tab order visualization on The Wall Street Journal Web site

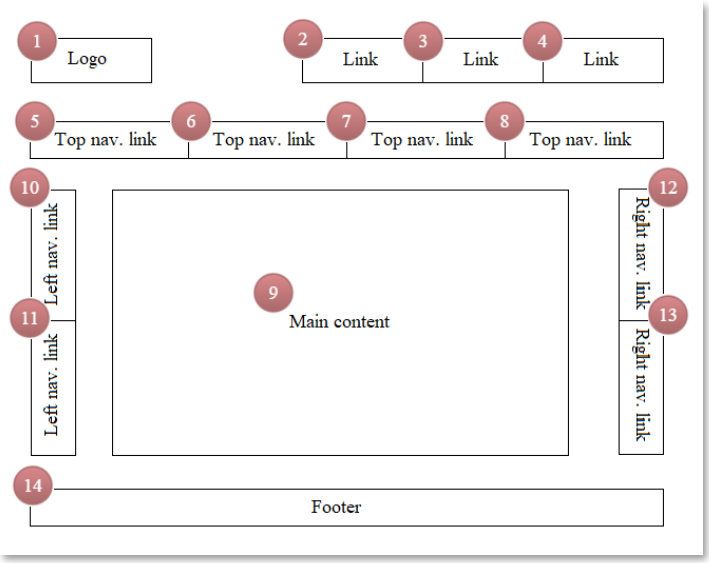


Figure 4.5. Correct implementation of an accessible tab order

In the accessibility visualization services proposed heretofore, the verification of Web accessibility is limited to the perception of the impact on disabled users of relatively simple (yet fundamental) features, such as colors, heading structure and tab order. More complex components, such as forms, switches, sliders, live-regions and more, are only flanked by error or warnings labels, but their behavior (correct or erroneous) is not simulated as if a dysfunction or difficulty invalidates it. For example, the developer can always fill in a form with non-existent labels, move the virtual knob of a slider that wouldn't allow a disabled user to select values, or enter and exit modal dialogs that would otherwise hinder the navigation of a screen reader or other assistive technologies. Nor would he have the guarantee that, by resolving the error displayed by the accessible visualizer or reported by the automated testing tool, the Web page or component would prove perfectly accessible.

In other words, the sighted developer naturally relies on his or her eyesight, the only sense that is used during the testing of the Web page, outclassed by hearing only in case he or she makes use of a screen reader software. Therefore, the sighted developer already knows what he expects from a widget and what task it allows him to do, or how a page structure is organized, because he sees and recognizes them. His sight therefore originates a *cognitive bias* derived from knowledge and experience that must be completely removed for an accessibility test to be total and without visual cues.

A tool that renders visible what is not, but above all *without* the normal visual presentation silently and inadvertently helping him or suggesting solutions: this is the real challenge that has not yet been taken up by any tool… yet. Chapters 5 and 6 will discuss in detail a software that leverages ARIA accessible markup of Web components and structure and presents it to developers as it is perceived by assistive technologies such as screen readers. The burden of determining whether or not the generated code is accessible is thus taken away from the developers, with features such as displaying accessible names and descriptions or not allowing the mouse to function on controls that have been mistakenly implemented as inaccessible by keyboard.

4.2 Tricky tech: is HTML5 really accessible?

This Chapter began by presenting developers' eyesight as one of their possible "handicaps", which hampers their reasoning and practical perception of how Web sites are navigated and interacted with by people with disabilities, particularly visual impairments. Yet, another problem faced by sighted developers is found directly in the very technology used to create and shape the Web itself: HTML5.

At the end of Chapter 1.4, HTML5 was introduced as a naturally accessible technology, with plenty of elements clearly defining their own purposes and roles, as in the case of `<button>`. However, this does not systematically occur: the sentence “the web is accessible by default” requires nuances, as there are some cases where even using plain HTML elements causes accessibility issues.

In 2020, Gov.UK found out[64] that number inputs (`<input type="number")` aren’t inclusive. In the case of

```
<label for="number-field">Number of tickets purchased:</label>
<input type="number" id=" number-field">,
```

for instance, the speech recognition software Dragon NaturallySpeaking can neither select the number field, nor dictate into it. In addition, the screen reader NVDA interprets it as an unlabeled spin button even if it actually presents a label. Not to mention the risk of users

[64] Gov.UK. Online available at https://technology.blog.gov.uk/2020/02/24/why-the-gov-uk-design-system-team-changed-the-input-type-for-numbers/. Last accessed 29 Sept. 2021.

accidentally incrementing a number while trying to do something else as, for example, scrolling up or down the page. In this case, there is no explicit feedback if they attempt to enter something that's not a number.

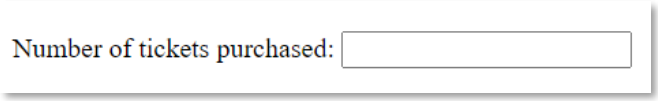


Figure 4.6. Visual representation of `<input type="number">`.

Still in the area of editable fields, another problem can be found in the implementation of a plain text field that collects the name of a user, such as one of this type:

```
<span>Name:</span>
<input type="text">.
```

A similar form, in fact, is perfectly correct from a HTML point of view, but it is defective from an accessible representation, since, for instance, a blind user is denied the option to associate the input with the text describing its purpose ("Name").

Finally, another example considers `<dialog>`: accessibility problems arise because of the dialog's focus algorithm, which states that unless a dialog contains a control with the `autofocus` attribute, then keyboard focus shall be given to the first focusable element within the dialog. For keyboard users, if a modal dialog is missing a focusable element, the focus appears to be visually lost. And if a focusable element is indeed present, screen readers will start announcing content from it, with content prior to the link resulting in it being ignored. Furthermore, some browsers don't support the `<dialog>` element at all (e.g., Apple Safari).

The problems don't end there. In general, when authors or developers create and combine elements to deliver well-formed documents, they have to tackle the relationship between syntax and semantics. Syntax is the arrangement of elements and attributes, which must follow style conventions in order to be correctly interpreted by user agents. Semantics, on the other hand, is concerned with meaning of information in Web pages rather than its merely presentation.

Whether they know it or not, authors begin with the semantics and often force it on elements that are not specific to the logical, structural and semantic characteristics of their intended use. It can happen if they are not experts, as in the case of a block-level logical division

`<div>` mistakenly enriched by an `id` or a `class` to make it become a navigation section element (`<div id="nav">`) or a specific introductory content (`<div class="header">`), when embedded semantic elements like `<nav>` and `<header>` already exist. The same applies to accessibility enrichment with WAI-ARIA markup (`<div id="nav" role="navigation">` and `<div class="header" role="heading">`) when built-in elements already provide all the required functionality. In the latter case, a redundancy problem arises, but in the former, assistive technologies will be confronted with a Web content entirely marked up using divs and spans, resulting in failure to index valuable content.

Even expert developers can abuse semantic characterization by leveraging JavaScript, JQuery, Angular, React or Vue over plain HTML for presentation purposes.

As has already been discussed at greater length by Rubano and Vitali[65], a simple yet significant mishandle of semantics occurs in the case of buttons. In the `<button>` element, syntax and semantics flawlessly coincide, leading to a natural interpretation by assistive technologies. But the push activity of a button can also be mimicked on elements that are not semantically buttons, such as spans, generic inline containers for phrasing content, which do not inherently represent anything. Hypothetically, a programmer may prefer stylistic choices such as:

- `<span onclick="myButton()"> … </span>`, a plain HTML markup with inline JavaScript;
- a plain HTML markup with separated JavaScript, such as

  ```
  <span class="myClass()"> … </span>
  …
  document.getElementByClassName("myClass").onclick = myButton;
  ```

- a plain HTML markup with separated JQuery, such as

  ```
  <span class="myClass()"> … </span>
  …
  $(".myClass").click(myButton);
  ```

The list certainly doesn't end there, and all of the included items may raise accessibility

[65] V. Rubano, F. Vitali. (2020). *Experiences from declarative markup to improve the accessibility of HTML*. In Balisage: The Markup Conference.

issues. Indeed, if they lack an ARIA connotation that suggests the conceptual bond between the span and a button (e.g. by means of a `role="button"` with a `tabindex` attribute), such elements undermine the meaningful accessible representation of assistive technology. This profoundly impacts, therefore, default keyboard and focus support. Span examples have been mentioned above, but basic accessibility issues involve links as well, which should be activated using the "Enter" key, while buttons are triggered by pressing the "Space" or "Enter" key. When links are used to behave like buttons, therefore, adding `role="button"` alone is not enough: it also requires a key event handler that listens for the "Space" key.

These examples demonstrate how easy it is to run into accessibility problems even when creating very simple HTML elements. HTML language, in fact, is neither sufficiently normative to prevent semantic misuses, nor sufficiently descriptive to encompass the markup of elements actually common and widely used in Web sites, such as tabs, carousels, accordions, etc. The next Chapter proposes an approach to improving Web accessibility through a markup visualization that immediately reveals whether the relationship between syntax and semantics of the components is correct and whether the accessibility information they convey is appropriate, making them usable or not.

Chapter 5

SahARIAn

This Chapter introduces SahARIAn, an accessibility validation software that operates in a significantly different way compared to the testing tools proposed in Chapter 3.1, designed only to verify the correctness of the code against various guidelines and/or legislations. Indeed, its main point of strength lies in the visualization of the Web content on the basis of accessibility markup, in a much more comprehensive way than the visualizers described in Chapter 4.1.

As a full-featured tool for sighted content authors, developers, and designers, SahARIAn will contribute to improving online accessibility, for a Web eventually characterized by equal access and equal opportunities.

5.1 SahARIAn: switching to blindsight

Before introducing SahARIAn, the accessibility visualisation tool at the heart of this dissertation, a brief explanation will be given of browser extensions. With no server behind it to send data to, SahARIAn naturally configures itself as a safe and reliable plug-in, even for sensitive projects.

5.1.1 Browser extensions

As their name states, browser extensions are additions or plug-ins that *extend* the functionality of a desired browser. According to the last version of Google Chrome Developers

Documentation, the Manifest V3[66]:

> *"Browsers are small software programs that customize the browsing experience. They let users to tailor Chrome [browser] functionality and behavior in many ways".*

They must fulfill a single purpose that is narrowly defined and easy to understand, that may belong to one of the following categories of functionalities:

- productivity;
- Web page content enrichment;
- information aggregation;
- fun and games;
- and others.

Built on web technologies such as HTML, JavaScript, and CSS, they run in a separate execution environment and interact with the Chrome browser. Their interfaces should be minimal and have intent and their icons are usually located on the top left of the browser (Figure 5.1), next to the searching bar.

Figure 5.1. Broswer extensions (Chrome). From left to right: axe DevTools, Visual ARIA, ARC Toolkit, SahARIAn, Google Scholar, WAVE.

Some browser extensions are aimed at enhancing a Web site's functionality and features, while others can also be employed to disable undesirable Web characteristics such as pop-up advertisements and other aspects.

In order to perform actions including reading or editing data on visited Web sites, browser extensions may require extensive *permissions* of the user's system, i.e. the information they can access. Or, even worse, they may request add-on permissions not truly necessary for the

[66] Chrome Developers, *Manifest V3*. (2021). Online available at https://developer.chrome.com/docs/extensions/mv3/overview/. Last accessed Oct. 3, 2021.

purposes they are supposed to accomplish, as in the case of the access to contacts, microphone, and more. According to Cisco's 2016 Annual Security Report[67], for instance, about more than 85% of organizations are affected by malicious browser extensions, that monitor and steal user data and activity.

When installing Chrome extensions, Google warns the user with three different warnings, which do not state that it is dangerous, but that it could be. They are:

- *high alert*: the extension in question is allowed to gain access to all data on your computer and the Web site the user visits;
- *medium alert*: the extension is requesting access to data on all or some of the visited Web sites (including bank accounts, usernames, etc.);
- *low alert*: the permission is limited to browsing history, physical location, copy and pasted data, bookmarks, list of installed apps, extensions and themes.

However, trusted e reliable browser extensions that require data and permissions are not malicious. Simply, in order to record click event handlers, access cookies and interact with the document object model of the current HTML page, they may need to interact with a server: this is the case with the accessibility testing tool axe, which records data to provide the user with a detailed overview of accessibility improvements over time. From this perspective, SahARIAn solves the problem at its root: it just does not rely on any central server to send data to. As it is not, for instance, a cloud service, where data must be sent in real time to be validated, it will never notify its developers that a user is browsing some specific Web sites, nor will it ever send them his or her information. The management of the accessibility visualization remains all client-side, ensuring that no privacy issues will ever arise in the case of sensitive projects.

5.1.2 SahARIAn browser extension

As Lighthouse, axe and ARC Toolkit, SahARIAn is a browser extension (currently available for Chrome and Firefox) for accessibility validation. Still under development, its acronym stands for "Sighted Architect's Helper for ARIA Notation" and easily encapsulates its purpose: to change the appearance of a Web page on the basis of its ARIA notation, in order to

[67] Cisco Security | *Cisco 2016 Annual Security Report*. (2016). Online available at http://mkto.cisco.com/rs/564-whv-323/images/cisco-asr-2016.pdf. Last accessed Oct. 3, 2021.

help sighted architects (developers, authors, designers, etc.) to directly perceive the accessibility of their Web content. Specifically, SahARIAn:

- removes the Web page style sheets (colors, fonts, element layouts);
- replaces all original images with a default placeholder;
- transforms all mouse interactions into keyboard input;
- show the accessible names and the accessible descriptions of elements;
- while leaving the textual content unchanged, it presents it in a simplified interface, within a gray frame, styled according to ARIA markup.

Therefore, if one or more ARIA shortcomings corrupt a code fragment, SahARIAn will make the element visually bizarre and practically inaccessible with the mouse, just as it would be if a blind user tried to interact with it.

SahARIAn benefits from an extremely simple interface, composed of a command pop-up (Figure 5.2) that is activated once the browser extension icon is clicked.

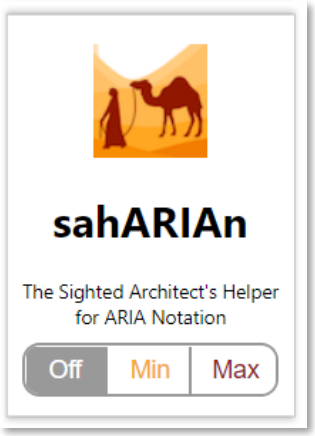


Figure 5.2. SahARIAn command pop-up.

The developer can choose from two available modes:

- *Min* ("minimum"), which contains a clean and functional markup of the ARIA rules. By activating it, the original page will be profoundly altered by losing its style sheets, which will be replaced by that of SahARIAn. In addition, the images will be replaced by a black and white placeholder. Thus, the interface will be streamlined, with emphasis on structure and not presentation.

- *Max* (“maximum”), which follows the same procedure already outlined in the *Min* modality, but, in addition, highlights in red the most critical areas of accessibility that are severely lacking in ARIA notations. Moreover, in this mode, Web content is encased in a pink frame, not a grey one.

The technical functioning of these modes and how it was possible to carry out the switching of styles and the transformation of inputs from mouse to keyboard, are described more extensively in the next Chapter, which delves into the architecture of the software.

The Web site of the University of Washington[68] (Figure 5.3) presents an interesting example of a successful accessible implementation, with zero errors reported by WAVE testing tool and a score of 100/100 by Google’s Lighthouse. It is thus a great starting point for showcasing SahARIAn's potential.

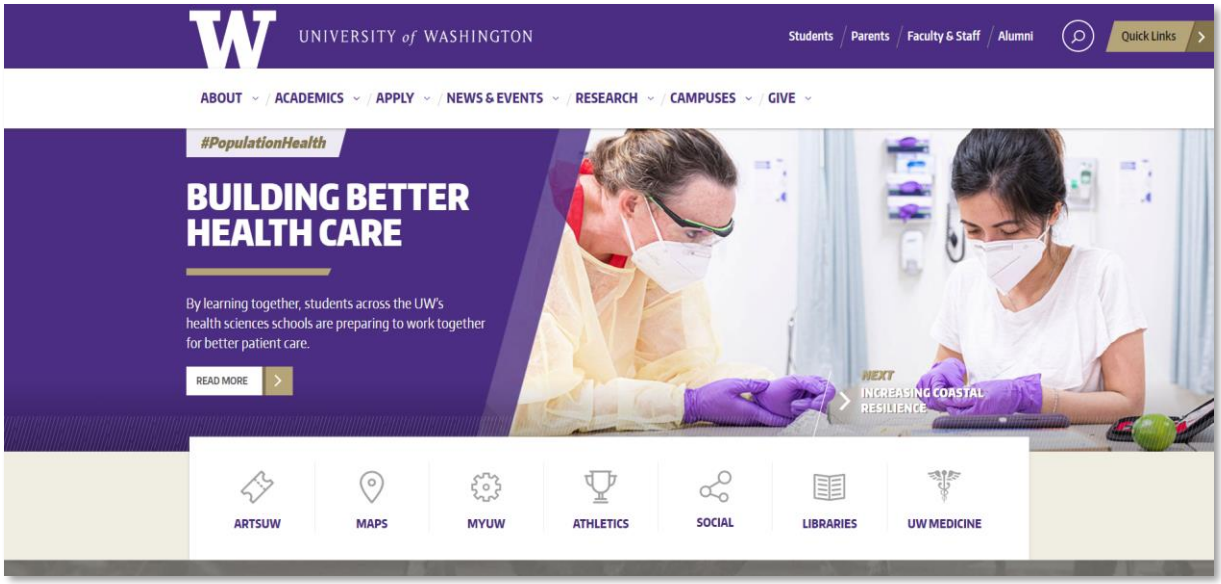


Figure 5.3. The homepage of the University of Washington Web site.

It is immediately noticeable that this site features an eye-catching design, with images and icons that play with each other with the shades of dark purple and light ochre. Once SahARIAn is activated via the "Min" or the "Max" button of the pop-up interface, all these skillful design

[68] University of Washington Web site | Homepage. Online available at http://www.washington.edu/. Last accessed Oct. 15, 2021.

choices (colors, positioning of elements, text styles, icons, images, etc.) immediately disappear (Figure 5.4). In their place, the structure of the site is well highlighted: the Web content remains unadorned and neatly arranged according to how the assistive technology of a user with a disability would access it.

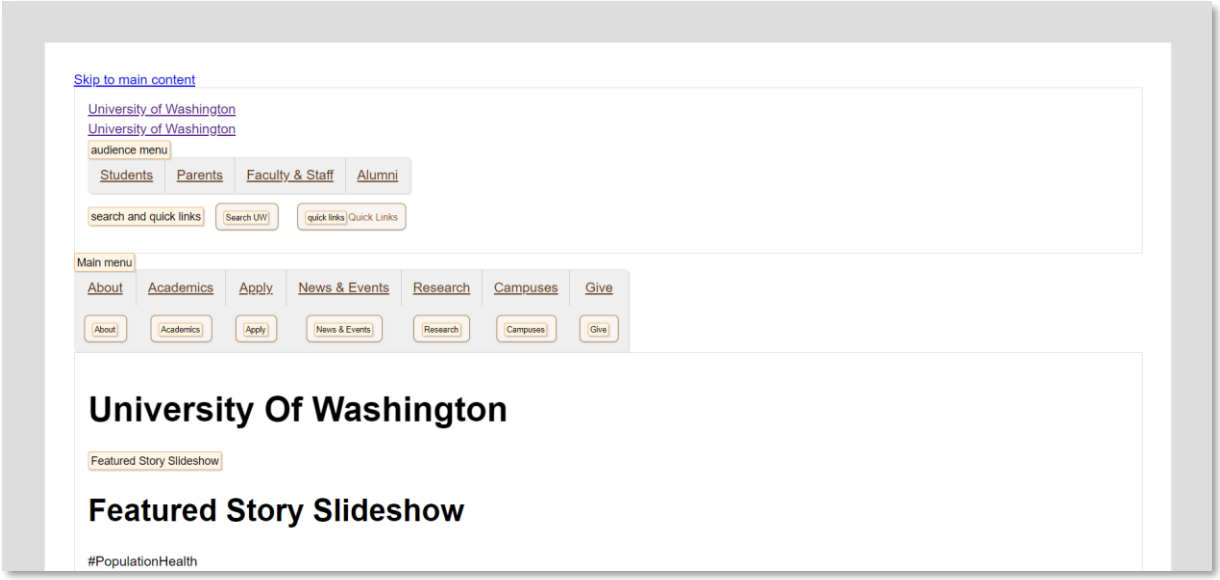


Figure 5.4. SahARIAn rendering (1) of the University of Washington Web site in "Min" mode.

SahARIAn includes style sheets that harmonize the style of all the elements that constitute the Web page and that can be as simple as buttons, navigation bars, lists, links, or as complex as trees, tablists and tab panels. These style sheets, their implementation, and their rules are further discussed in Chapter 6.6, which provides a more in-depth analysis of the visual representation of interface widgets and page structure. However, already in this example it can be observed that the colors of SahARIAn are intended to appear neutral, with white and gray nuances and hints of orange for some elements such as buttons or accessible names labels (discussed in Chapter 6.4).

Analyzing more in detail Figure 5.4, it can be noticed that the Web content comes enclosed in areas with a thin grey border: they are those that in ARIA notation are called Landmark roles (e.g. `main`, `banner`, `complementary`) and that outline the structure of the content with large containers. Two of these areas can be observed in the Figure:

- a `banner`, corresponding to a `<header>` element and containing the main logo, the title of the page, and a first navbar (in grey background);
- and an underlying `main` element, with the dominant content of the page.

Scrolling down the homepage (Figure 5.5), within the main content, another interesting element is encountered: an image.

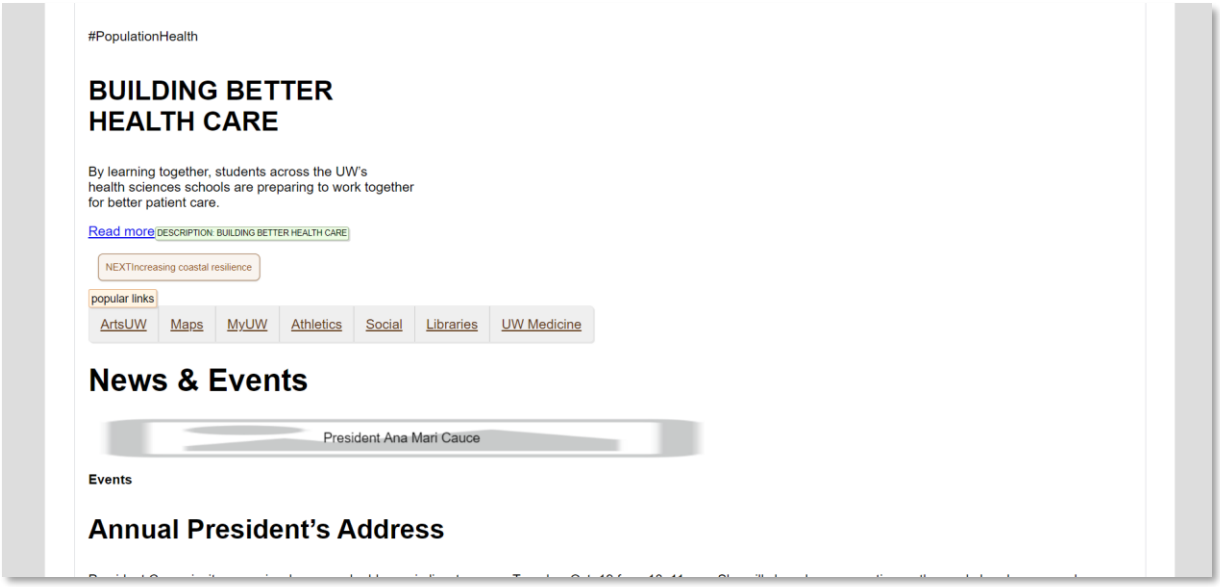


Figure 5.5. SahARIAn rendering (2) of the University of Washington Web site in "Min" mode.

Chapter 6.3 deals precisely with the subtle issue of image handling, but for now it is sufficient to know that all original images are removed and replaced by a standardized placeholder like the one shown in Figure 5.6, suitably stretched to superimpose the image's alternate text.

Figure 5.6. SahARIAn's default placeholder.

The "Max" version of SahARIAn software contains all the style rules of the "Min" mode, but also introduces some extra features and highlights critical areas of accessibility. As can be clearly observed in Figure 5.7, in which the University of Washington site is again taken as an interesting example, the Web content is now partitioned into a layout grid. In fact, the "Max" mode establishes that the element with the `banner` role is positioned at the top center of the page, with the `main` underneath it. All the items that are placed inside the body of the page but within generic `<div>` containers are highlighted with a salmon pink background, to emphasize the fact that they should be placed inside areas with well-defined landmark roles. This is to improve the accessibility of the site by facilitating the navigation of a screen reader or other assistive technology.

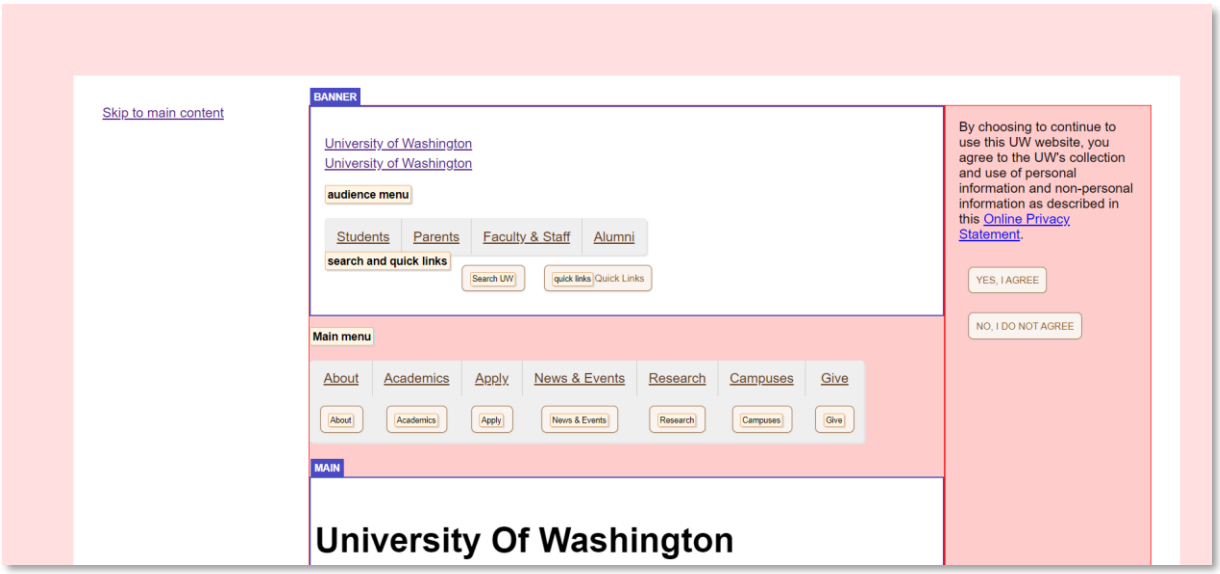


Figure 5.7. SahARIAn rendering of the University of Washington Web site in "Max" mode.

In addition, elements with ARIA landmark roles feature a convenient label that immediately identifies their role (in the image, in blue and capital letters: "banner" and "main") and a blue border that demarcates the area. Finally, in order to make the "Max" view immediately different from the "Min" display, the frame, which was previously gray, is turned pink.

The University of Washington Web site was perfect from an accessibility standpoint. In the case of a site with poorly implemented content, however, how would SahARIAn proceed?

To answer the question, let's consider the site of the Italian state railways, Trenitalia[69] (Figure 5.8). Not only does it have major structural problems at the accessibility level (with a very low score of 59 on Lighthouse), but it also exhibits usability issues, with crashes, actions that aren't completed in the way one would expect, and a global sub-optimal user experience.

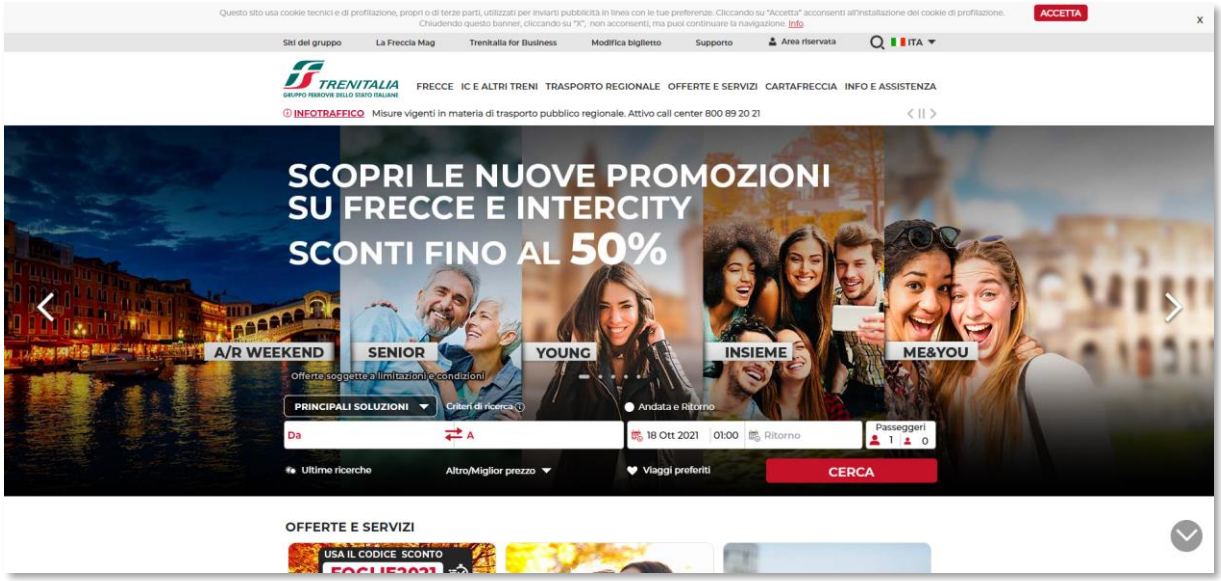


Figure 5.8. Trenitalia official Web site.

Everything that makes up the normal structure of a Web page in SahARIAn visualization (Figure 5.9) is completely annihilated. Trenitalia Web site does not make use not only of ARIA landmarks, but also of the semantic containers that HTML proposes instead, such as `<header>` and `<main>`. In their place, it employs general `<div>` elements with `id="main"` and `id="top-header"`, enriched with styles and colors through a specific CSS. The result? A catchy Web site, on a graphic level, a real disaster, on a semantic and structural level.

[69] Trenitalia Official Web site | Homepage. Online available at https://www.trenitalia.com/. Last accessed Oct. 17, 2021.

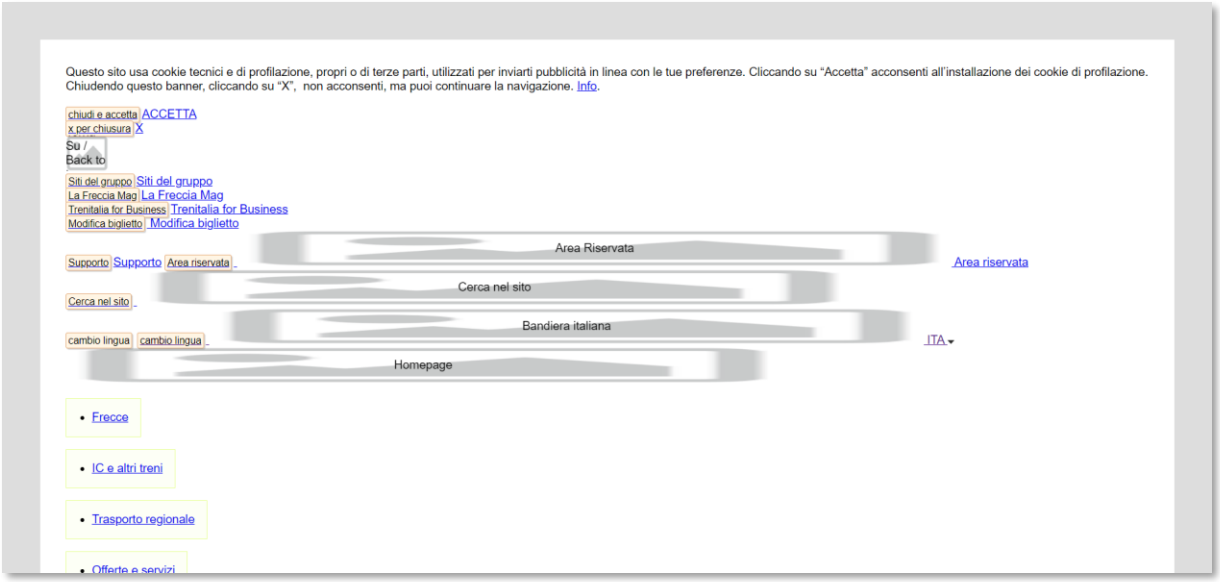


Figure 5.9. SahARIAn's rendering of the Trenitalia official Web site ("Min" mode).

Since the structural problems are already heavily present at the level of the "Min" view, the "Max" mode doesn't present substantial differences (Figure 5.10): it is immediately noticeable, however, that the entire content of the Web page is not at all partitioned into landmark regions due to the absence of a layout grid. Instead, it is simply organized in plain `<div>` elements that are all siblings to each other, within the `body`.

Figure 5.10. SahARIAn's rendering of the Trenitalia official Web site ("Max" mode).

To conclude, through the provided examples, it is therefore possible to state that SahARIAn is an extremely easy-to-use visualizer of accessibility issues, even when exploited by inexperienced developers. This Chapter has introduced it with a high-level presentation unburdened by technical details, whereas the next one investigates its implementation more deeply, to provide a step-by-step analysis of SahARIAn's functioning. Chapter 6 indeed further elaborates on some elements already briefly seen in the University of Washington and Trenitalia Web sites, such as accessible names, images, page structured and interactive interface widgets.

Chapter 6

SahARIAn architecture

After a high-level introduction of SahARIAn validation tool, the purpose of this Chapter is to go “down the rabbit hole”, digging deeper into the project’s implementation. SahARIAn was, and still is, an integrated effort between a domain expert and two implementers, two of whom are sighted. Together, several implementation choices have been made, which will be reported in the following paragraphs in order to provide a comprehensive conceptual and concrete structure of the application.

First of all, the five-step process of SahARIAn functioning is introduced, involving, in brief, deactivation of images and styles, transformation of mouse events into keyboard events, and injection of SahARIAn style sheets (Chapter 6.1). Afterwards it is demonstrated how it is feasible to toggle SahARIAn without any loss of information of the elements being processed (specifically, styles and images), delving into the delicate case of images in the following chapter. Chapter 6.5 provides detailed examples of the on-the-fly and dynamic visualization of SahARIAn, required to capture runtime accessibility issues. Last but not least, plenty of examples of ARIA interface widgets and page structures are offered in Chapter 6.6, which, divided into the five ARIA Role categories, shows the breadth of SahARIAn's coverage and visualization of accessibility errors.

6.1 Deep down the rabbit hole: the architecture

SahARIAn generates an autonomous and self-sufficient HTML document which, through the code injection technique, allows interaction with the designated page to be tested, with which a user dialogues through the pop-up interface (Figure 5.2).

Code injection is mostly known as a malicious process, but in SahARIAn and many other reliable browser extensions it is used to insert a piece of code into a Web page (or other

software). This strategy is rather simple: after waiting for an application to be loaded and executed, additional code is inserted to modify and influence its behaviour. As a result, the source code of the application to be injected is not irretrievably modified. Crackers can exploit this technique for cyber-attacks, such as installing malware on servers or accessing sensitive databases. But code injection is also used to introduce useful features not previously available on an application, as in the case of SahARIAn. Our program, in fact, loads specific style sheets for ARIA notations that help Web developers to test the accessibility of their pages. And to accomplish this, there is no other way than to inject code on the DOM of the page to be evaluated and change its HTML without modifying its source code.

SahARIAn's software is divided into two parts:

- the pop-up, with its little HTML document (displaying the tree buttons, “Off”, “Min” and “Max”), its CSS and some small JavaScript scripts;
- the very heart of the program, a long JavaScript script that is forced inside the host application when the browser extension is activated via one of the pop-up buttons. It contains all the styled ARIA roles, properties and states, as well as the functions to deactivate the native Web page style sheets and load the SahARIAn ones.

The very first function of SahARIAn’s main script is `activate(force)`. This function dialogues directly with the scripts contained in the pop-up. Indeed, if "Min" or "Max" are clicked, the `activate(force)` function, which receives as input a boolean parameter (`true` or `false`), triggers SahARIAn to activate the desired visualization. Likewise, hitting the "Off" button will revert it to the original version.

If `force` happens to be `true`, SahARIAn performs a five-step process (Figure 6.1):

1. *Image deactivation*. As sight-impaired users cannot visualize images, they are removed and replaced with a default placeholder with their alternative text superimposed (if present). This procedure is carried out every 200 milliseconds (0.2 seconds) to ensure that any images or advertisements that may appear after SahARIAn’s activation are replaced anyway.
2. *Individual style attributes deactivation*. Since SahARIAn places a strong emphasis on the structure of a page rather than its presentation, it completely deactivates style choices placed within the HTML code tags. The `display` and `visibility` attributes are instead ignored because they do not represent accessibility problems,

since the choice to make an element invisible is an implementation issue and not a way to hide structural problems in the page. This process is also performed every 200 milliseconds in order to prevent any styles from being reset by the code of the page being analyzed over time.

3. *Accessible name and description computation*. In order to interact with the elements of a Web page, assistive technologies rely on the accessible tree, an internal accessible representation of the page. Each of its nodes is an accessible object that must have an accessible name and can have an accessible description. SahARIAn calculates both (the second only if present) and shows them next to the element in question.
4. *Event bypass activation*. By means of this technique, all events performed with the mouse are transformed into keyboard events. The developer can therefore continue to use his mouse or his trackpad for convenience, but he is made aware if a keyboard event has been misconstrued, as he won't be able to execute the corresponding mouse-based action.
5. *Style sheets deactivation*. Not only SahARIAn disables the styles embedded in the HTML code of the page, but also its CSS style sheets. Moreover, since it must be able to retrieve this style information at the moment of deactivation (via the "Off" button), it stores it in copies, and then deactivates it. Again, `display` and `visibility` attributes are treated as exceptions.
6. *SahARIAn style sheets activation*. The two modes ("Min" and "Max") provide different style sheets according to their different functionalities. Chapter 6.4 presents an overview of the most used components on the Web, such as radio buttons and tabs, whose simplicity should not be misleading because it still leads to serious accessibility problems.

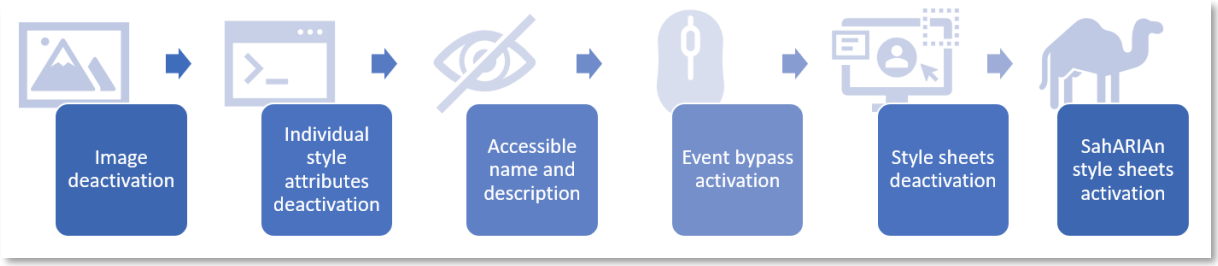


Figure 6.1. The five-step process of SahARIAn activation.

If `activate(force)` receives a `false` input, it deactivates all the processes described above (if they were previously activated) in the opposite way to how they have been listed. Therefore, SahARIAn removes its specialized style sheets, then reactivates the original style sheets, deactivates the mouse event bypass, makes the styles within the HTML tags operational again, and replaces the previously deactivated images in order to return the page to its initial state.

The New York City Police Department (NYPD) official Web site[70] offers a nice example of the end result of SahARIAn's five-step approach, in which the page is gradually despoiled of its style and images in favor of a structure visualization functional to the accessibility analysis.

[70] NYPD Official Web site | Homepage. Online available at https://www1.nyc.gov/site/nypd/index.page. Last accessed Oct 13, 2021.

Figure 6.2. The NYPD official Web site.

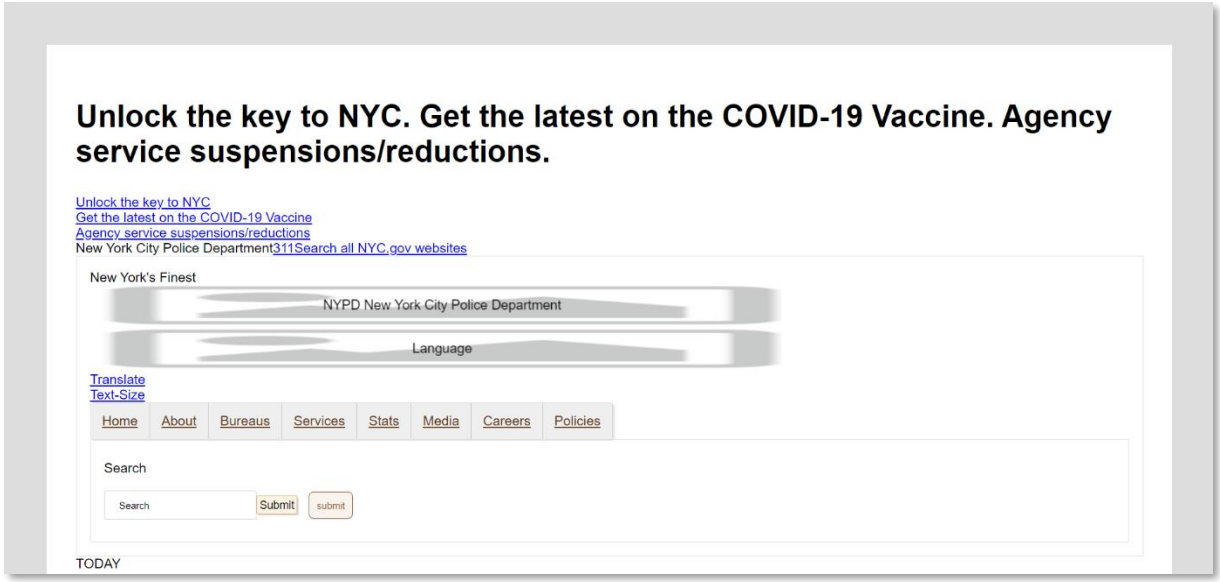


Figure 6.3. SahARIAn rendering of The NYPD official Web site.

To conclude, since it is not yet available on the Chrome store, in order to install SahARIAn it is now required to visit the browser's extensions panel, activate the developer mode, select the folder containing all the SahARIAn files and install it manually.

6.2 Toggling SahARIAn without information leakage

When SahARIAn encounters a noticeable element that requires processing, such as an image or a style attribute placed within the HTML code tags, in the first case it will replace it with a default image (more about it in Chapter 6.4), in the second it will directly remove it. In both scenarios, however, the information that is substituted or deleted must not be lost, since it has to be restored when SahARIAn is toggled via the "Off" button.

Because our software neither relies on a server nor on the user's specific machine, it employs specific functions (`sideline()`, `sidelined()`, `restore()`) to "set aside" and restore valuable information. This information is saved as value of attributes (starting with "`data-saharian-sideline-`") added to the elements themselves:

- for inline style attributes (via `<style>` placed within HTML code tags), the content is transformed into values of `data-saharian-sideline-style` attributes.
- for images (`<img>` or `<source>` elements, or `<div>` or `<span>` enriched with `role="img"`, as more extensively discussed in 6.4), the related information is preserved by means of:
    - `data-saharian-sideline-src`, that holds the original source, i.e. the URL, of the image;
    - `data-saharian-sideline-srcset`, in the case of a list of image file URLs, along with size descriptions;
    - `data-saharian-sideline-width`, as the name implies, used to set aside width information (e.g. "500px");
    - `data-saharian-sideline-height`, same for the height of the image;
    - `data-saharian-sideline-role`, in these cases always "`img`";
    - `data-saharian-sideline-play`, for elements such as gifs and iframes.

However, SahARIAn must also ensure that an element has not *already* been processed. This is especially true in the case of images, where an original image is replaced by a standard placeholder, i.e. actually another image element, visually different but with equal tags. To signal that an element has already been handled, SahARIAn computes a `hash`, a code in the form of a string of fixed size generated from an attribute of the element (e.g., the `src`). It is highly

improbable that two different sources generate the same code, so it can be considered unique.

For instance, in the case of images, to control whether the element is original or if it has already been processed, SahARIAn must check the `hash` of the `src`, which in the second case would have been inserted as a value of the newly generated `data-saharian-sideline-src-hash` attribute (a real-life case would be, for example, `data-saharian-sideline-role-hash="104387"`).

In addition, SahARIAn also adds “hash” attributes that signal that the other element features have been successfully implemented. They map the same attributes enumerated in the list above, with a `done` value: for instance,

- `data-saharian-sideline-style-hash=”done”`,
- `data-saharian-sideline-src-hash=”done”`,
- `data-saharian-sideline-width-hash=”done”`, etc.

Eventually, when SahARIAn software is disabled, the `restore()` function reinstates the shelved information temporarily saved in `data-saharian-sideline-` attributes. More precisely:

- it removes the `data-saharian-sideline-hash-` elements; and
- it fetches all the values stored inside the `data-saharian-sideline-` attributes, placing them in their original locations.

In this way, the developer can activate and deactivate SahARIAn without any image or style information ever being lost.

6.3 The delicate handling of images

Nowadays, Web sites deeply rely on pictures and animation to convey their content. Users with disabilities, particularly visual and cognitive impairments, who employ screen readers do not visualize images. Accordingly, in its Success Criterion 1.4.5 "Images of Text", WCAG 2.1 states that Web pages should avoid text images except when essential to a particular visual presentation, such as a company logo.

When a screen reader encounters an image, it seeks to read a text alternative that offers the same information displayed in the image. The most common form of encoding a text alternative is accomplished by the HTML `alt` attribute, but it can also be achieved with ARIA notation (`aria-label` and `aria-labelledby`). If an image does not feature any of these elements, a screen reader may announce the image's `src` attribute instead.

Web sites also make use of decorative images: images with no function or information content that are hidden by assistive technologies. This can be done in various ways:

- using an empty alt attribute (`alt=""`);
- using ARIA landmark `role="presentation`;
- using a CSS background image.

Image deactivation comes as the very first step in the SahARIAn process and is run every 0.2 seconds to re-block any image or advertisement that may appear on the page. The process is carried out by the asynchronous function `deactivateImages()`, which is triggered whenever it encounters an HTML `<img>` or `<source>` element, or a `<div>` or a `<span>` with `role="img"` attribute. First of all, it checks that the element in question has not already been processed by SahARIAn itself, meaning that it is an original image. More precisely, as more exhaustively outlined in the previous chapter (6.3), it controls that there are no `data-saharian-sideline` attributes that contain the original image information. If they are not present, and thus that element is a real new image ready to be handled, the function seeks its text alternative:

- if there is an `alt`, this will be the preferred option;
- otherwise, the value of the `aria-label` attribute will be chosen;
- if there is none, the result will be "No alt text".

At this point, the function creates a svg image that displays a standardized placeholder icon, stretches it purposely to contain the text, and substitutes it to the original image. Over this placeholder, then, it overlays a second image containing only the calculated text alternative. Both images are then given forced dimensions based on those of the real image (Figure 6.6).

Figure 6.6. The NASA logo from the official Web site[71] and its SahARIAn rendering, with the text alternative ("Nasa") superimposed on the generic placeholder.

If the `<img>`, the `<div>` or the `<span>` did not feature an `alt` or an `aria-label` attribute, the algorithm will impose a red background-color to alert the developer that the element presents a strong accessibility issue that needs to be fixed (Figure 6.7).

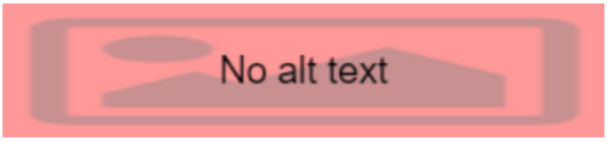


Figure 6.7. A SahARIAn image with no alternative text, in an alerting red background.

In particular, the image containing the placeholder and the alternate text are encoded in two different ways depending on the original element:

- if it was a `<div>` or a `<span>` with `role="img"` and therefore the image was not encoded in a `src` attribute but was placed inside the CSS `background-image` attribute, then the source element is maintained and its `background-image` is replaced with the placeholder. In other words, a `<div>` will remain a `<div>` and a `<span>` will remain a `<span>`, with a different background;
- if the element was an `<img>` with a `src` attribute, the function inserts the placeholder with its text alternative as the value of the `src` attribute using the data URI encoding.

[71] NASA.gov | Homepage. Online available at https://www.nasa.gov/. Last accessed 3 Oct., 2021.

The data URI scheme provides a way to embed small items of data into a URL. Since SahARIAn software doesn't rely on a server, it leverages this procedure to encode and set aside image information. These will be needed once SahARIAn is deactivated with the `reactivateImages()` function that is launched when the user clicks on the "Off" button displayed on the plug-in interface. This function takes advantage of the `restore()` function already described in the previous chapter (6.3) , which was intended for images and inline style attributes inside HTML code tags. Here, it recovers the image values stored inside the `data-saharian-sideline` attributes (`src`, `srcset`, `width`, `height`, etc.) and reinstates them in their respective places, thus restoring the images to their original appearance.

6.4 Accessible name and accessible description

When a Web page is interpreted by assistive technologies, the elements it contains are arranged in a parallel structure, known as the accessibility tree, which is an internal accessible representation of the page itself. This structure is made up of accessible objects (HTML markup elements and their attributes), which provide information about their roles, states and properties. For assistive technologies to identify an object, they must read its accessible name (AN), a short label that provides information about its purpose. An object may also feature additional information in the form of an accessible description (AD), not necessary if the name is a sufficient explanation of the object's role, but required if the element features an `aria-describedby` property.

In 2014, W3C published the "WAI-ARIA 1.0 User Agent Implementation Guide"[72] document, superseded in 2018 by the "Accessible Name and Description Computation 1.1"[73] specification. The latter provides a non-normative algorithm that applies across multiple content technologies. Its starting point is a DOM element, and its output is a flat string that can be a single word or several space-separated tokens.

---

[72] J. Scheuhammer and M. Cooper. (2014). *WAI-ARIA 1.0 User Agent Implementation Guide*, W3C, W3C Recommendation. Online available at https://www.w3.org/TR/2014/REC-wai-aria-implementation-20140320/. Last accessed Oct. 4, 2021.

[73] J. Diggs, B. Garaventa and M. Cooper. (2018). *Accessible Name and Description Computation 1.1*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/accname-1.1. Last accessed Oct. 4, 2021

The algorithm is written in pseudocode, a plain language description of steps that omits language-specific code and is augmented with natural language description details. This process, intended for human reading rather than machine one, is well suited to the purpose of the W3C specification, which gives precise and clean instructions on how to calculate the accessible name and description in various programming languages (JavaScript, Java, C++, etc.). The only problem was that, in fact, the instructions were neither precise nor clean at all, and interpreting the pseudocode took longer than it should have.

At the moment, the JavaScript algorithm implemented in SahARIAn is activated only for elements that specify one of the following three attributes: `aria-label`, `aria-labelledby` or `aria-describedby`. Instead, it relies on the normal display for the remaining cases, i.e. HTML `<label>` elements or other elements with an `alt`, a `title`, or a `role` (e.g. `button`, `cell`, `checkbox`, `heading`, etc.) that is not `presentation` or `none`. The function, `generateAccessibleNames()`, generates two attributes: `data-saharian-an` and `data-saharian-ad`. More precisely:

- `data-saharian-an` contains the text of the accessible name of the element in question. Specifically:
  - in the case of elements with an `aria-label` attribute, the accessible name is made by the content of `aria-label`;
  - for elements with `aria-labelledby`, the accessible name is generated for each idref present.
- `data-saharian-ad` contains the text of the accessible description of the element in question. This accessible description is created when an element has the `aria-describedby` attribute and is composed of the text of its corresponding idrefs.

If the element is hidden (via `display="none"`, `role="presentation"` or `visibility="hidden"`), even if it has `aria-label`, `aria-describedby` or `aria-labelledby` attributes, its accessible name and/or description will not be displayed.

For instance, let's take the example of an interesting table (Figure 6.8) displaying the names of some of the best antagonists featured in Christopher Nolan's films. The table is structured as follows:

- a table (`<table>` or `<div role=”table”>`) with two attributes:
  - `aria-label=”Table of antagonists”`;

- `aria-describedby="myDescription"`.

- and a table caption, i.e. a generic `<div>` with a `id="myDescription"` attribute containing the description "*Best antagonists in Christopher Nolan's movies (in alphabetical order)*";

Best antagonists in Christopher Nolan's movies (in alphabetical order)

Name	Opponent	Movie	Release date
Andrei Sator	the Protagonist	Tenet	2020
Bane	Batman	The Dark Knight Rises	2012
Joker	Batman	The Dark Knight	2008
Mann	Cooper	Interstellar	2012

Figure 6.8. A simple HTML and CSS table example.

Through its algorithm, SahARIAn takes the `aria-label` of the table and interprets it as an accessible name, providing the attribute `data-saharian-an="Table of antagonists"` as output. Visually (Figure 6.9), it will be displayed as a small label placed before the element it refers to, by means of a CSS pseudo-element (`::before`). It will feature a smaller font and a light orange background.

Moreover, the algorithm also computes the accessible description of the table by tracking the string related to the element with `id="myDescription"`, i.e. the table caption `<div>`. Thus, the output will be the attribute `data-saharian-ad="Best antagonists in Christopher Nolan's movies (in alphabetical order)"`. The accessible description will be placed after the table, via another CSS pseudo-element (`::after`) and will be visualized with an even smaller font and a light green background.

Table of antagonists Best antagonists in Christopher Nolan's movies (in alphabetical order)

Name	Opponent	Movie	Release date
Andrei Sator	the Protagonist	Tenet	2020
Bane	Batman	The Dark Knight Rises	2012
Joker	Batman	The Dark Knight	2008
Mann	Cooper	Interstellar	2012

DESCRIPTION: Best antagonists in Christopher Nolan's movies (in alphabetical order)

Figure 6.9. SahARAIn rendering of a simple table with accessible name (orange label at the top) and accessible description (green label at the bottom) before and after the table element.

Therefore, the aforementioned procedure allows a sighted developer immediately to perceive the accessible name and/or description of an element, without having to test the Web page with a proper screen reader software.

6.5 In the blink of an eye: a dynamic visualization

One of the most innovative features of SahARIAn is its on-the-fly visualization. As already discussed in 3.1, existing software already displays invisible ARIA notations with icons (WAVE) or labels (Visual ARIA, ARC Toolkit) superimposed on the Web page, together with suggestions and recommendations for correct usage. However, their testing is not dynamic at all: it starts and ends with the loading of the page in the browser. Conversely, as been already mentioned, SahARIAn execution runs every 200 milliseconds on the actual DOM corresponding to the generated markup, in order to visualize runtime issues.

For example, in Chapter 3.1, Figure 3.12 (here reported as Figure 6.3) already demonstrated the failed WAVE update of the `aria-checked` state of a selected `role="checkbox"` element, that still displays a `false` value even if the checkbox is indeed checked.

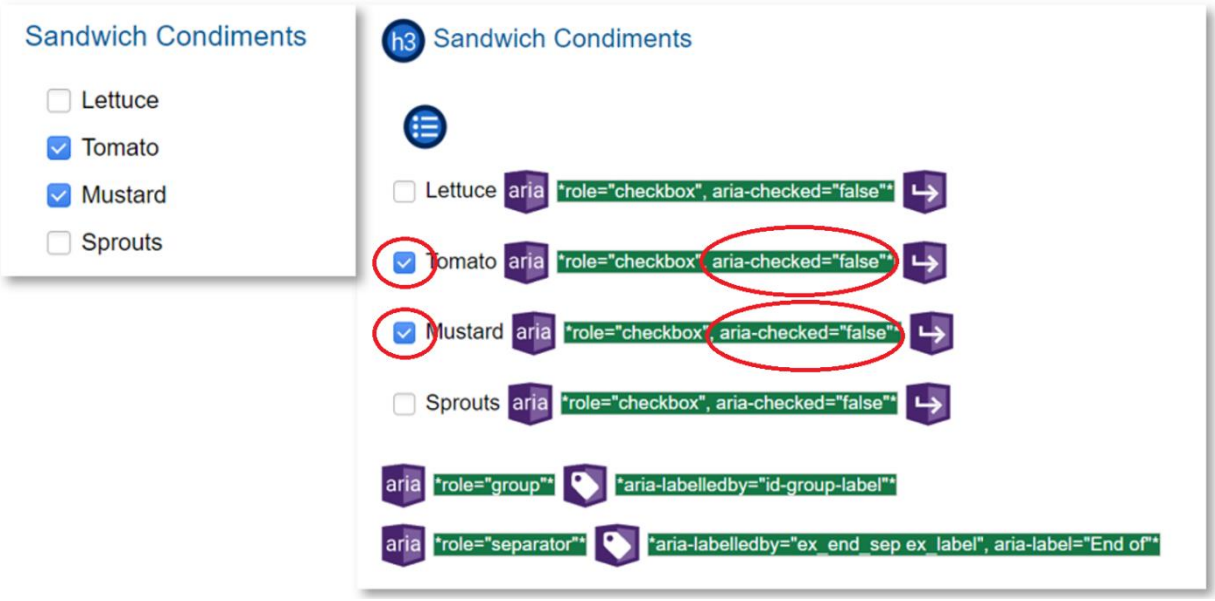


Figure 6.3. WAVE failed update of `aria-checked` state on two checked checkboxes.

SahARIAn, on the other hands, continuously interfaces with the Web content and its markup. This can be nicely observed in the example of the checkbox taken from the same "WAI-ARIA Authoring Practices 1.1" document provided above. Figure 6.4 shows its SahARIAn rendering ("Min" modality), before and after the selection of the two checkboxes. As can be easily witnessed, the update of `aria-checked` from `false` to `true` is successfully reported within the elements, and not with inconvenient labels.

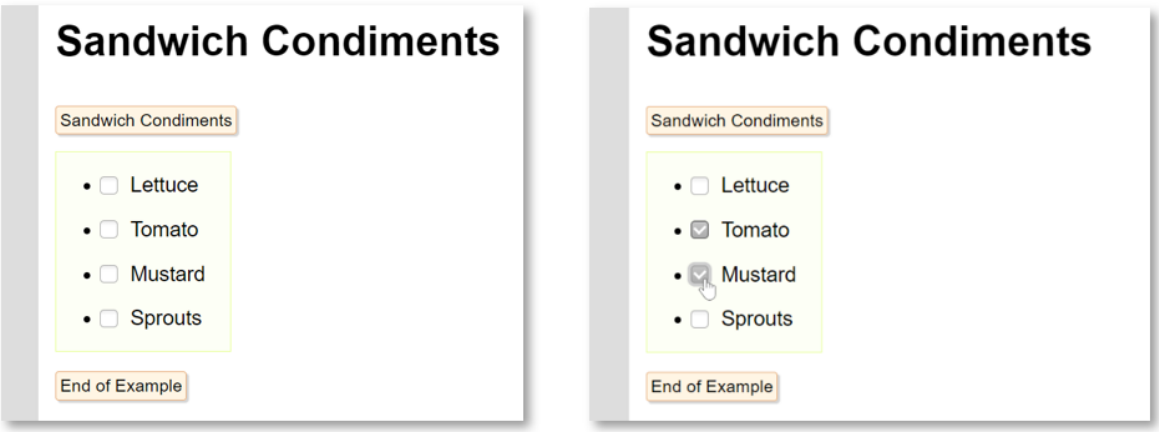


Figure 6.4. SahARIAn successful rendering of the updated `aria-checked` state of two checkboxes.

Let's now explore what would have resulted if the checkboxes had been implemented with accessibility issues. The correct example of "WAI-ARIA Authoring Practices 1.1" was realized with the following HTML snippet:

```
<li> <div role="checkbox" aria-checked="false" tabindex="0">
Lettuce </div> </li>
```

By means of some live editing on the page (via DevTools, Ctrl+Shit+C or F12), let's remove the `tabindex` global attribute from the first element. Now "Lettuce" can't be focused and no longer participates in the sequential keyboard navigation (Tab order). Let's also modify the second element, by deleting `role="checkbox"`: although it still contains some ARIA markup (the `aria-checked` property), now "Tomato" is no longer an ARIA checkbox, but just a plain `<div>`. SahARIAn will act accordingly, instantly changing its appearance as shown in Figure 6.5.

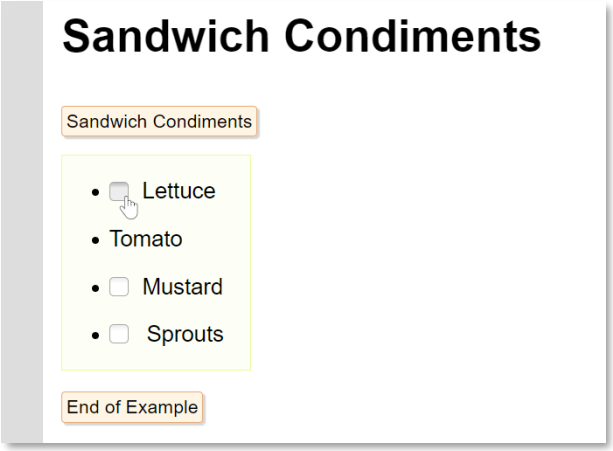


Figure 6.5. SahARIAn visualization with accessibility issues in the first two checkboxes.

In the first case ("Lettuce"), as the checkbox cannot be activated by the keyboard, SahARIAn ensures that it will not be triggered with the mouse either: the sighted developer can keep on using the pointing device but will be forced to operate as a visually impaired user. In this particular case, the checkbox square will remain empty and without a grey check mark. In the second example ("Tomato"), as the checkbox is completely gone, the element is treated as if it were a `<div>` containing plain text.

Therefore, SahARIAn works simultaneously with the developer, allowing live editing directly on the HTML injected into the page. By continuously updating, it also incorporates and transforms styles and images from ads or other sources, capturing runtime issues as they occur.

6.6 SahARIAn interface widgets and page structure

SahARIAn style sheets ("Min" and "Max") contain the CSS information for designing and displaying the elements of the tested Web page. My contribution to the project has primarily been located here, in the creation of rules in the form of `[role="radio"][aria-checked="true"]{ *rules* }`. The goal was to misrepresent visual experiences on the basis of an incorrect ARIA markup and, conversely, to provide an optimal interaction whenever the non-sighted users' experience was already well defined. As already mentioned in this dissertation, in fact, for screen reader users, ARIA controls are in charge of rendering the non-visual experience.

Web pages are rich, highly structured documents that make extensive use of navigation elements and widgets. The latter are simple or complex interactive components that present information to users. Whether they are simple buttons and checkboxes, or complicated accordions and treegrids, they must be implemented as accessible. To achieve this, in 2009 W3C published "WAI-ARIA Authoring Practices 1.0", replaced in 2019 by "WAI-ARIA Authoring Practices 1.1". This already mentioned specification recommends approaches to create an accessible Rich Internet Application using correct implementations of WAI-ARIA roles, states, and properties. It provides guidance and sample codes to make widgets, navigation, and behaviors accessible, while also explaining concepts behind them. This guide was required for two main reasons:

- the languages used to create rich, dynamic Web sites (HTML, JavaScript, CSS, etc.) do not natively include all the features necessary to make them usable by users with disabilities who rely on assistive technologies;
- assistive technologies, in turn, must understand the meaning and purpose of the user interface elements they encounter.

This is the strength of ARIA: to describe elements so that they can be recognized and interpreted by assistive technologies. As a matter of fact, ARIA roles are nothing more than semantic information about the purpose of an element. The semantic conflict described in

Chapter 4.2, thus, pops up here once again. Let's take as an example a tree widget, i.e. a hierarchical list with child items to be expanded or collapsed. A developer can create a perfect and functional tree widget in HTML, CSS and JavaScript. However, HTML does not provide any semantic tree element. What appears to non-disabled users as a functional collapsible tree widget, will not be perceived by, or operable by, a user with disability, because his assistive technology will not recognize its role.

"WAI-ARIA Authoring Practices 1.1" seeks to prevent these accessibility issues by teaching how to make accessible widgets and RIA patterns through ARIA markup and keyboard support. Although the intent is noble and the structure is excellent, however, some problems have been found in the samples, such as the excessive use of CSS rules for classes instead of ARIA roles and properties.

Chapter 1.4 listed the six categories of ARIA roles, divided by interface widgets and page structures: Abstract Roles, Widget Roles, Document Structure Roles, Landmark Roles, Live Region Roles, and Window Roles. Selected examples of SahARIAn are now proposed for each of the categories, with the exception of the first one, since Abstract Roles are the foundation upon which all other ARIA roles are built and must not be used by content authors.

6.6.1 SahARIAn Document Structure roles

Document Structure roles give information about non-interactive, static sections of the Web page and outline the structure of the content. Some of them map onto existing HTML tags (e.g. `<form>`, `<article>`) and are only meant for cases when using the native tag is impractical, or to fill gaps in support. The most common include:

- `article` and `feed`;
- `group`;
- `heading`;
- `figure` and `img`;
- `list` and `listitem`;
- `row`, `rowgroup` and `rowheader`;
- `table`.

The list also features `cell`, `columnheader`, `definition`, `directory`, `document`, `math`, `note`, `presentation`, `region`, `separator`, `toolbar`, `tooltip`.

Figure 6.10 presents an extract from "Authoring Practices 1.1"[74] with two interesting Document Structure roles: a feed (`role="feed"`) made of articles (`role="article"`) with independent, self-contained content. As can be seen, the (very straightforward) structure is emphasized at the detriment of presentation, while the information is perfectly preserved. Notice the accessible name and description and the handling of the images, the first of which has a declared alt (`alt="5 stars"`), both described in the previous chapters.

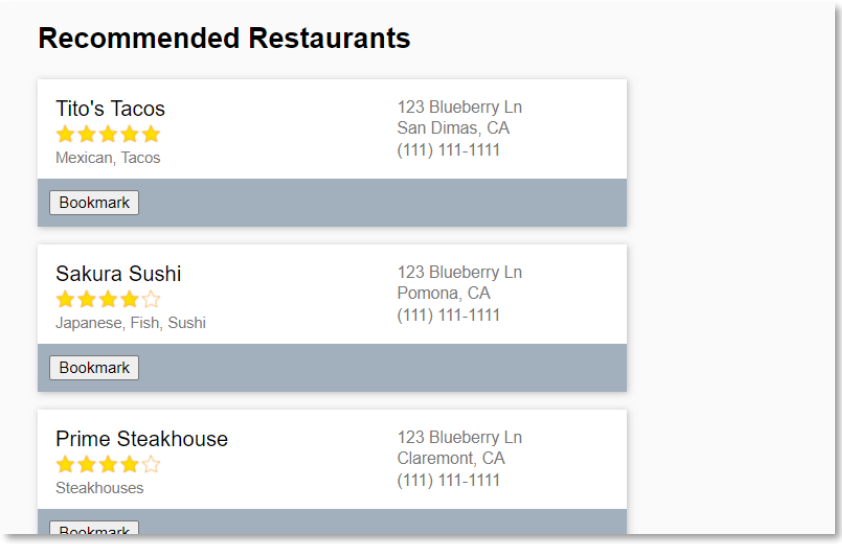


Figure 6.10. The "Authoring Practices 1.1" feed sample, with `role="feed"` and `role="article"`

[74] *WAI-ARIA Authoring Practices 1.1* (2019) | WAI-ARIA Feed role example. Online available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/feed/feedDisplay.html. Last accessed Oct. 7, 2021.

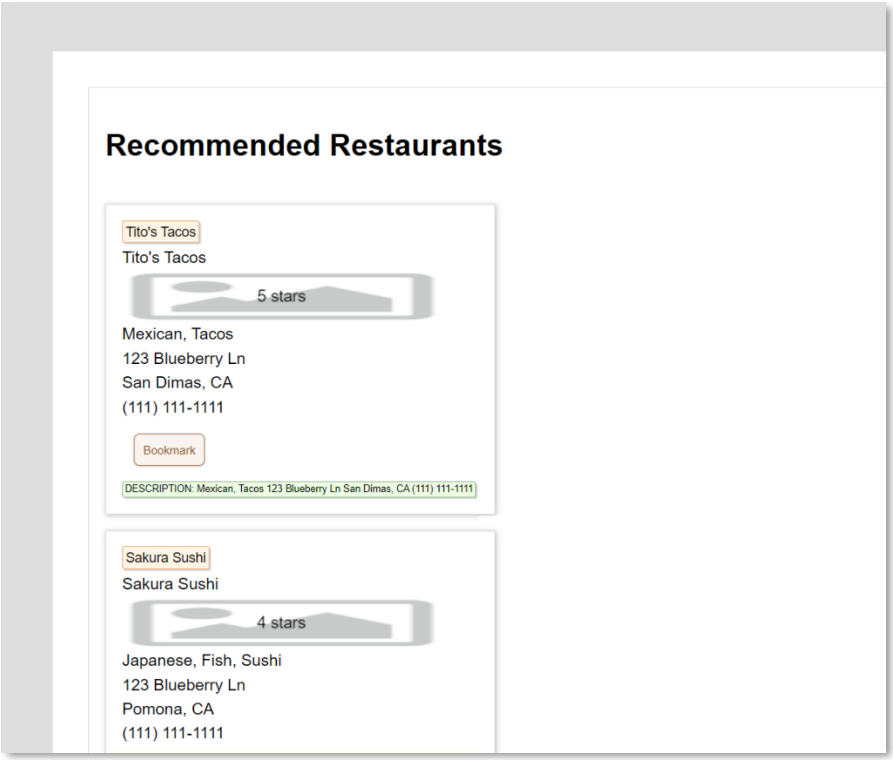


Figure 6.11. SahARIAn rendering of the feed sample from "Authoring Practices 1.1"

Another common example of a Document Structure role is the table featuring the best antagonists in Christopher Nolan's movies (Figure 6.8, here reported at the top left of Figure 6.12). This `<div role="table">` contains, nested, the following elements:

- `<div role="rowgroup">`, which replaces HTML `<thead>` and `<tbody>` and identifies the elements that serve as containers for the table header and table body rows (as ARIA notation does not include separates roles);
- `<div role="row">`, in place of the HTML element `<tr>`;
- `<span role="columnheader">` (`<th>`), which identifies the cells containing column labels (the header);
- `<span role="cell">` (`<td>`) for the remaining rows, i.e. the individual cells.

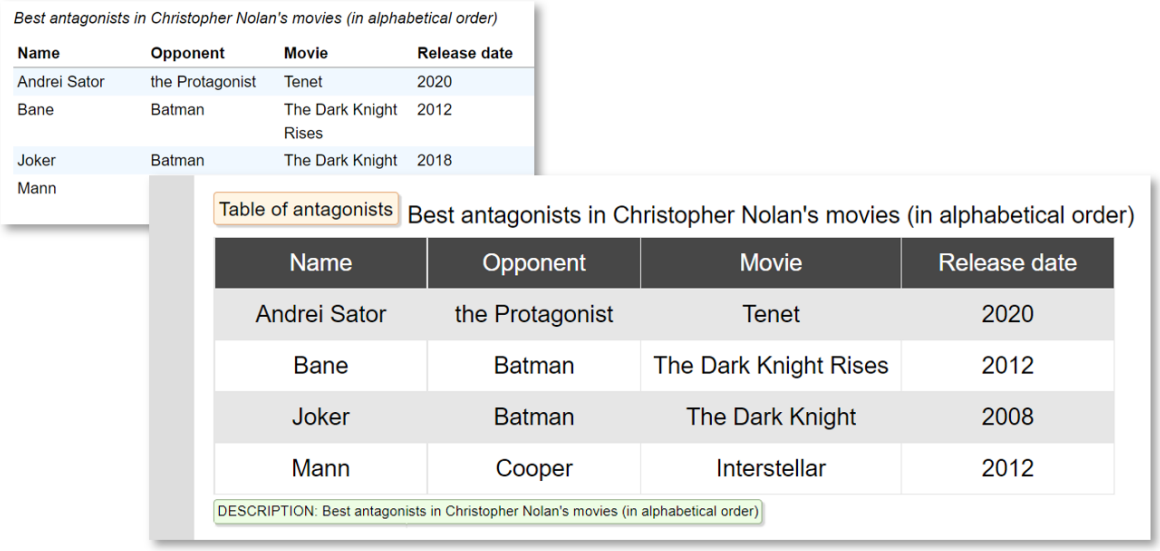


Figure 6.12 Comparison between a HTML and CSS table with optimal ARIA notation (at the top left) and its perfect SahARIAn rendering.

Let's see what happens by introducing issues in these ARIA attributes (Figure 6.13). Since adding or removing ARIA notation does not change sighted users' visualization, the HTML and CSS table on the top left appear to be unmodified, even if it conceals serious accessibility issues. However, enabling SahARIAn will make them immediately visible. Indeed, now the table is utterly disfigured. The first cell of the header ("Name") no longer presents a `role="columnheader"`: thus, it has become a simple span within a `<div role="row">` parent. The other cells of the first column have also lost their `role="cell"` attribute, being now simple spans. Finally, the table body (the second `<div=role="rowgroup">`) is not exposed as a row group to assistive technology and this results in the whole table being white, without alternating between gray rows.

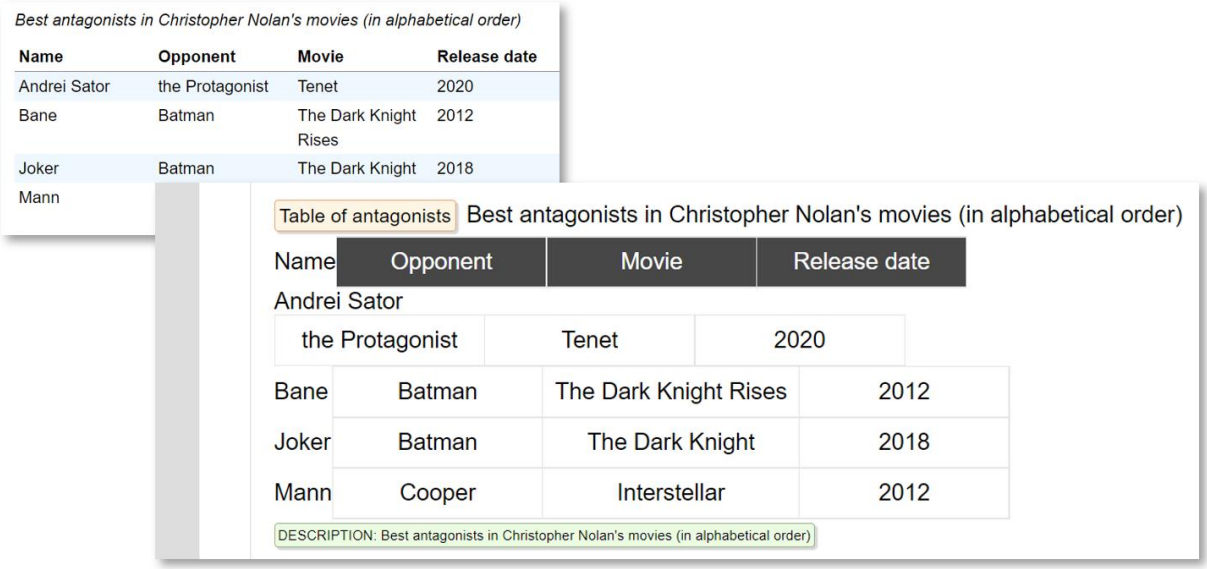


Figure 6.13 Comparison between the same table (at the top left) but with accessibility issues well displayed in the SahARIAn rendering

6.6.2 SahARIAn Window roles

ARIA Window roles act as windows within the primary Web page. Currently, they only consist of `alertdialog` and `dialog`.

The `alertdialog` role is a lightweight version of a `dialog`. It presents only informational issues, with limited complex information, and must always be given an accessible name. What differentiates it from the plain `alert` role, however, is that the latter contains only static content and has no interactive focusable elements, while `alertdialog` must always present at least one focusable element through the Tab.

For example, let's consider the `alertdialog` displayed in "Authoring Practices 1.1"[75] (Figure 6.14). Once SahARIAn is activated, the modal must maintain the same accessibility requirements that apply to sighted users (Figure 6.15).

[75] *WAI-ARIA Authoring Practices 1.1*. (2019) | WAI-ARIA Alertdialog Role example. Available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/dialog-modal/dialog.html. Last accessed Oct. 7, 2021.

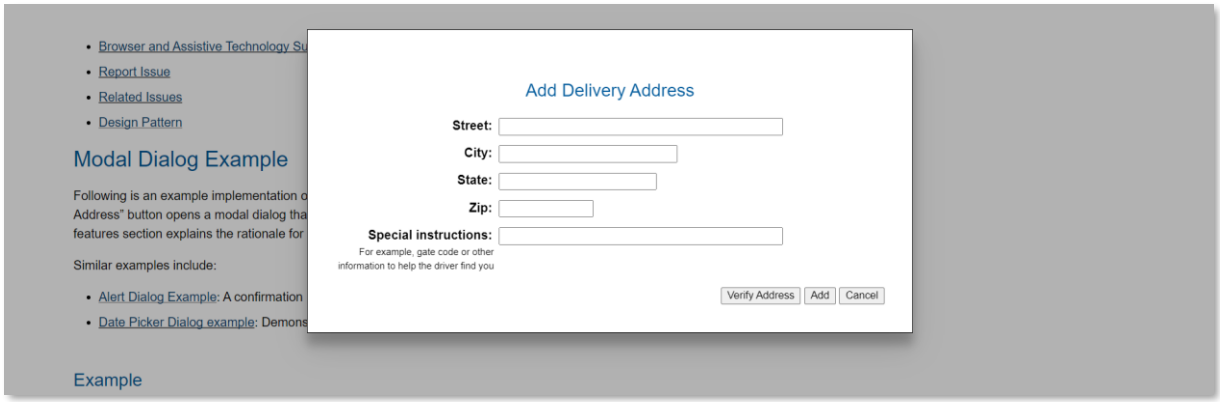


Figure 6.14 `alertdialog` *example taken from "Authoring Practices 1.1" documentation*

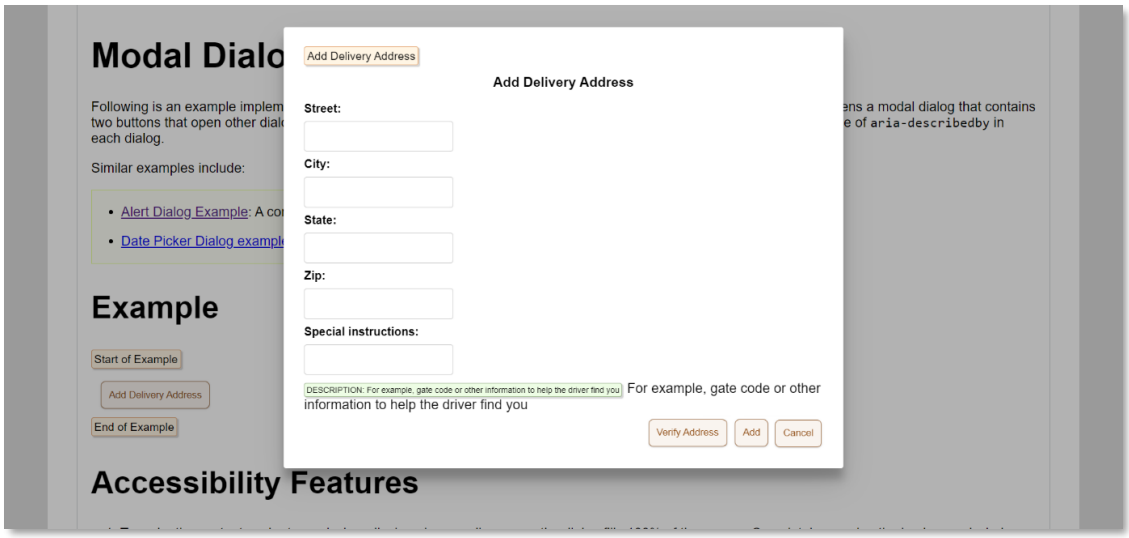


Figure 6.15 SahARIAn rendering of the "Authoring Practices 1.1" `alertdialog` *example*

If, for example, the `role="alertdialog"` was removed, the result would be disastrous: the content would no longer be recognized as a pop-up window and it would be dropped at the very end of the Web page.

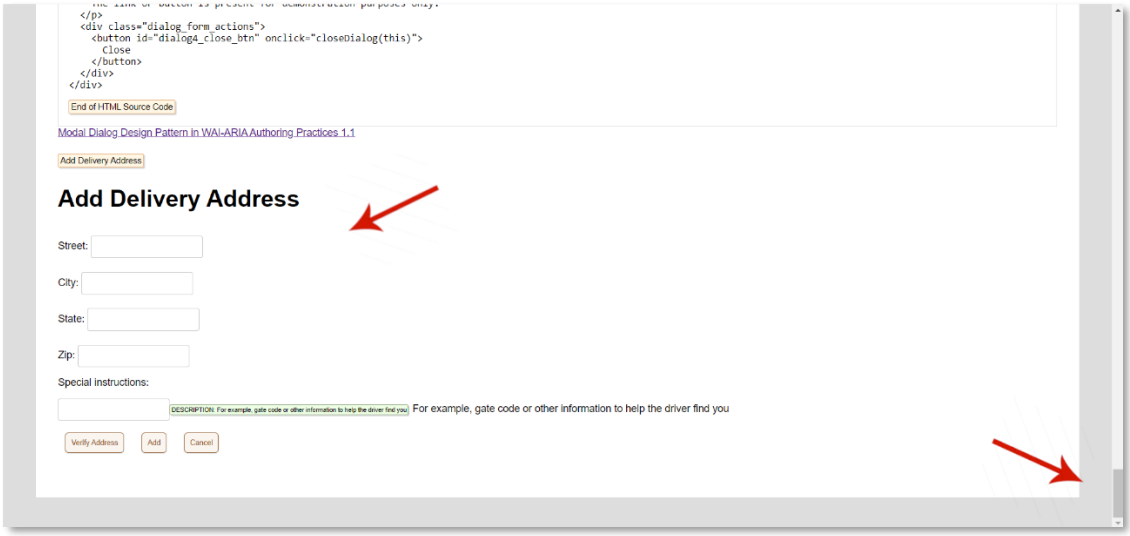


Figure 6.16 `alertdialog` *example with accessibility issues, dropped at the bottom of the Web page*

6.6.3 SahARIAn Live Regions roles

ARIA Live Region are roles used to notify screen readers that a content has been updated on the Web page. This proves to be most beneficial in Rich Internet Applications, which make extensive use of asynchronous technologies such as Ajax to update their content. Since screen readers can only focus on one part of the page at a time, they won't notice what is being updated. Live Regions roles (`alert`, `log`, `marquee`, `status` and `timer`) and their attributes (`aria-live`) are designed to prevent this issue.

SahARIAn style sheets supply these roles with a yellow background, to distinguish them from the remaining content and to emphasize their warning function. Figure 6.17 and Figure 6.18 show the dynamic example of a `role="timer"`: once a user clicks on the "Start countdown" button, the `<span role="timer">` next to it will change without having to refresh the page, showing a countdown from 3 to 0 seconds.

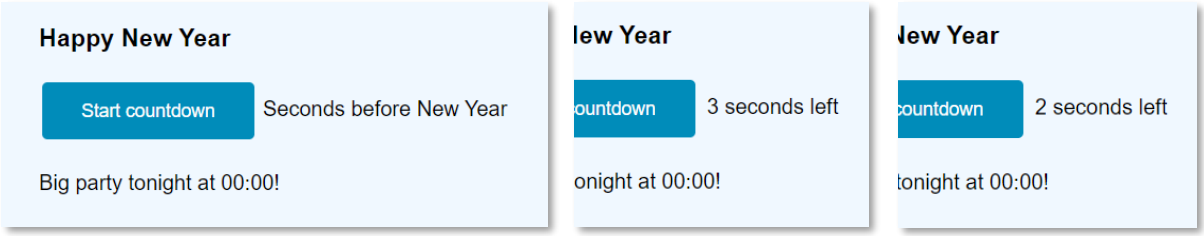


Figure 6.17 `role="timer"` *example with HTML and CSS classes.*

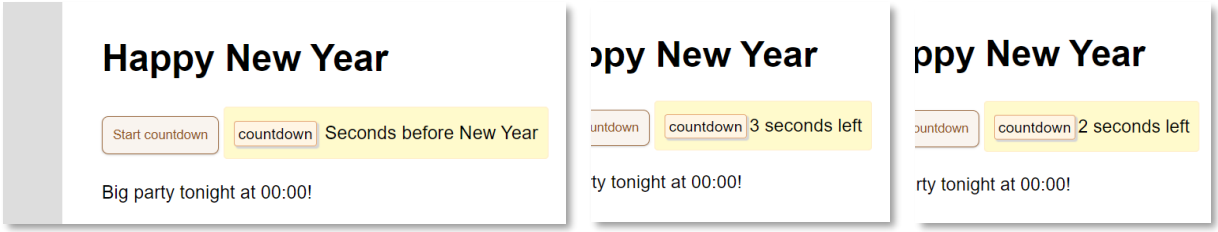


Figure 6.18 `role="timer"` *example within SahARIAn style sheets.*

6.6.4 SahARIAn Widget roles

As already mentioned in Chapter 6.5, widgets are interactive component of a graphical user interface, with which the user can interact: they describe common interactive patterns that lack semantic analogues in HTML. "Authoring Practices 1.1" divides them in two groups:

- *standalone widgets*, which may be part of larger, composite widgets: `button`, `checkbox`, `gridcell`, `link`, `menuitem`, `menuitemcheckbox`, `menuitemradio`, `option`, `progressbar`, `radio`, `scrollbar`, `searchbox`, `separator`, `slider`, `spinbutton`, `switch`, `tab`, `tabpanel`, `textbox`, `treeitem`;
- *composite widgets*, i.e. containers that manage other standalone components: `combobox`, `grid`, `listbox`, `menu`, `menubar`, `radiogroup`, `tablist`, `tree`, `treegrid`.

Chapter 6.2 already presented an optimal example of one of the most common widgets, the `checkbox` (Figure 6.3 and Figure 6.4). It compared WAVE and SahARIAn and showed that the latter, by running every 200 milliseconds, is also dynamic in displaying changes in the code

that lead to accessibility issues. A similar example can also apply to a sibling of the checkbox, the `radio`. Checkboxes and radio buttons are both controls which can be clicked to select one option, but in the latter case only one option in the group can be checked at a time.

For instance, a simple survey template from the popular online survey provider SurveyMonkey[76] is a fine example of Saharian's behaviour with `"role=radio"` elements (Figure 6.19).

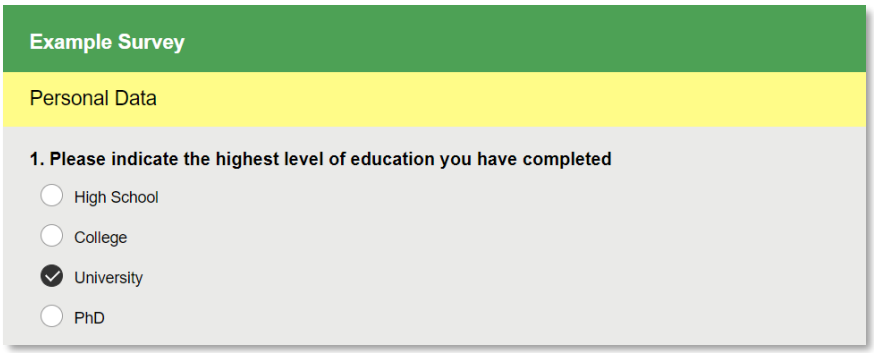


Figure 6.19 A survey template from SurveyMonkey depicting `role="radio"` elements

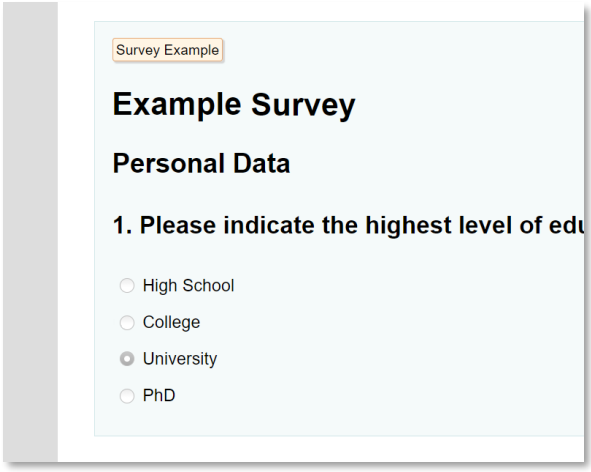


Figure 6.20 SahARIAn rendering of a survey template from SurveyMonkey with `role="radio"` items

[76] Survey Monkey.com | Survey Monkey templates. Online available at https://www.surveymonkey.com/mp/survey-templates/. Last accessed Oct. 9, 2021

As may be observed, in SahARIAn the element with `role="radiogroup"`, which identifies a `div` as a container for radio buttons, has a light blue background. Within it, the radio buttons (in the form of `<span role="radio">`) must feature an `aria-checked` property, `true` if the control is clicked and `false` if it is not. Based on this structure, let's place a few accessibility issues. In Figure 6.21, for example, the `radiogroup` element has been removed: the grouping of radio buttons is no longer properly exposed to assistive technologies. In addition, `aria-checked` has been replaced by `aria-selected`, a property applied to select multiple items in a group of controls (e.g. checkboxes). Finally, the second option ("College") has been semantically disassociated from its syntax, as the `role="radio"` has been removed. Now, assistive technologies can no longer identify the element as a radio button but as a simple `<span>`.

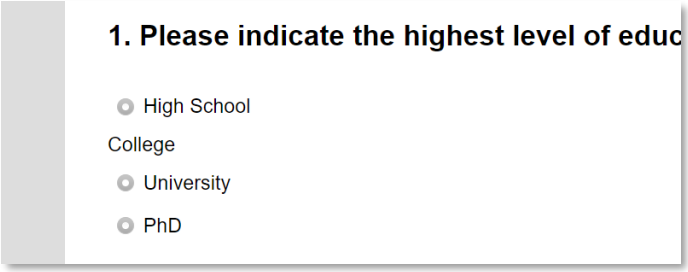


Figure 6.21 The `role="radio"` example depicted in Figure 6.18, but with accessibility issues

A bad example of `role="radio"` can be found on Ryanair's Web site[77], where two radio buttons at the beginning of the homepage ("Return trip" and "One way") have been implemented as buttons without proper radio semantics (Figure 6.22). They do have an `aria-label` attribute, but they are structured as simple `<button>` elements. This error is difficult to notice because the CSS rules of the Web page transform them into perfect radio buttons. Once SahARIAn is activated, however, the problem becomes apparent: the two elements will lose the standard round shape of radio buttons that was imposed on them by the CSS visual layout and will show their real structure, i.e. simple buttons.

[77] Ryanair Web site | Homepage. Available at https://www.ryanair.com/. Last accessed Oct 10, 2021.

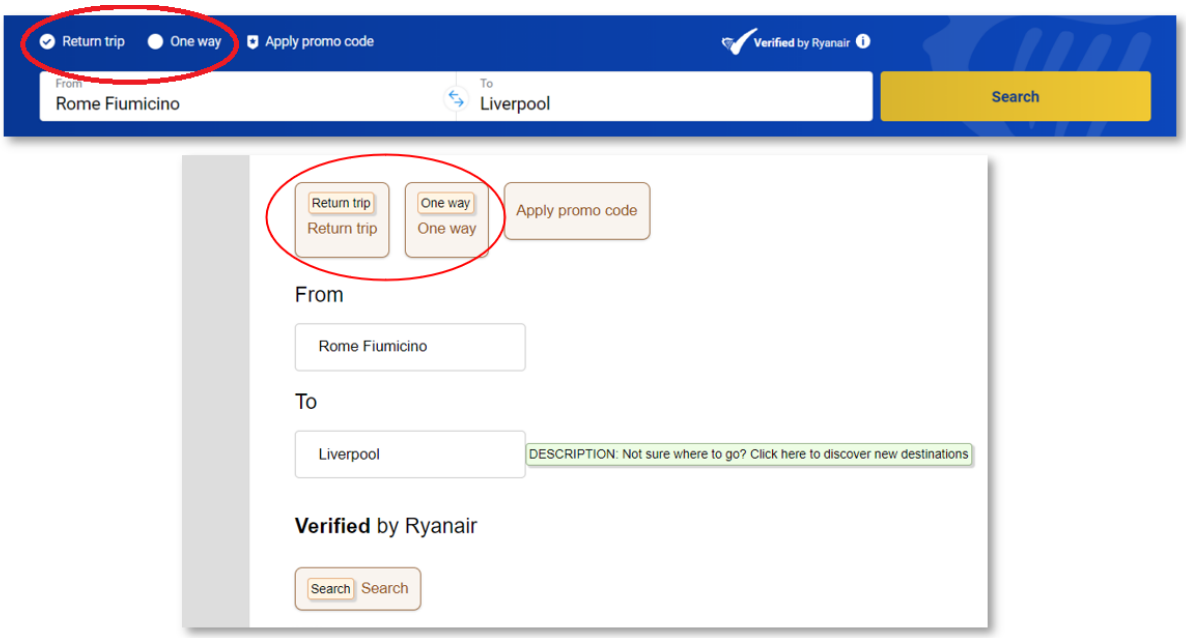


Figure 6.22 A bad example of radio buttons implementation. At the top, an extract from the homepage of the Ryanair Web site. At the bottom, its SahARIAn rendering.

SahARIAn is also equipped to manage composite widgets, such as tablists, tabs, and their related tab panels. The role of this interactive structure is to display one panel of content at a time by clicking on a list of tab elements that activates the panel, commonly placed on the top border. HTML does not offer native elements to represent `role="tablist"` and its components, which are therefore rendered with generic `<div>`, `<span>` and `<buttons>`. In its dedicated page[78], "Authoring Practices 1.1" provides a useful widget that implements the design pattern for tabs (Figure 6.23).

[78] "WAI-ARIA Authoring Practices 1.1", 2019 | Example of Tabs with Manual Activation. Available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/tabs/tabs-2/tabs.html. Last accessed Oct. 9, 2021

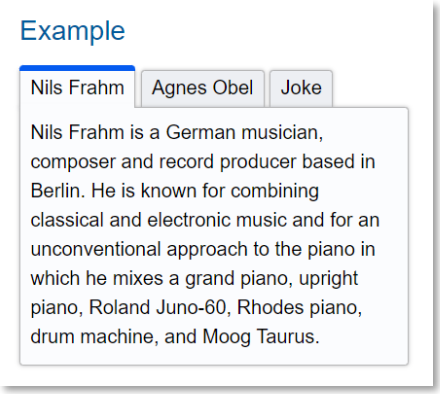


Figure 6.23 "Authoring Practices 1.1" Tabs example.

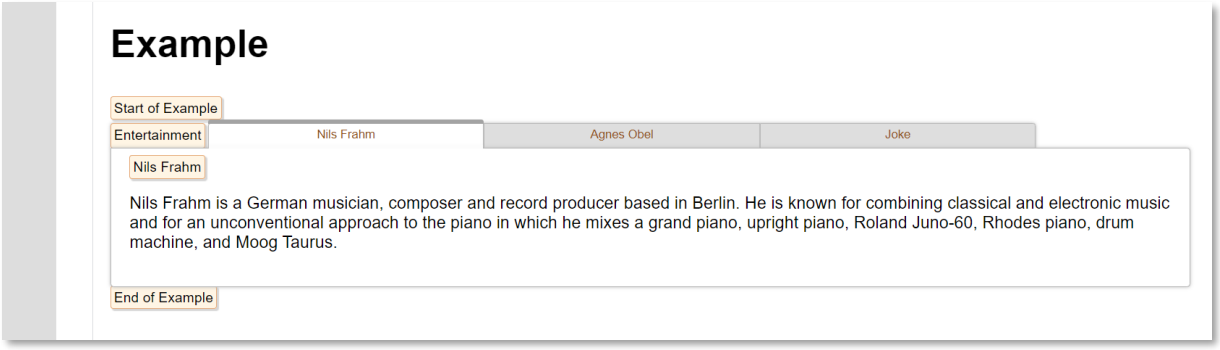


Figure 6.24 SahARIAn rendering of "Authoring Practices 1.1" Tabs example.

As in the case of radio buttons, tabs also rely on the `aria-selected` property, which indicates whether the tab control is activated and whether its associated panel is shown. The first element of this `role="tablist"` sample (`<button role="tab">`, "Nils Frahm") is automatically shown with an `aria-selected="true"`, while the other two elements, "Agnes Obel" and "Joke" feature a `tabindex="-1"`. This is due to the fact that a non-sighted user has to access the other buttons not with the Tab key but with the directional arrows. The `role="tabpanel"`, on the other hand, must be included in the page Tab sequence especially if it does not contain a focusable element, and for this reason it must have a `tabindex="0"`. A well implemented JavaScript code is essential to make this widget accessible. Figure 6.25 illustrates how the widget would look if the tab interface semantics were not provided.

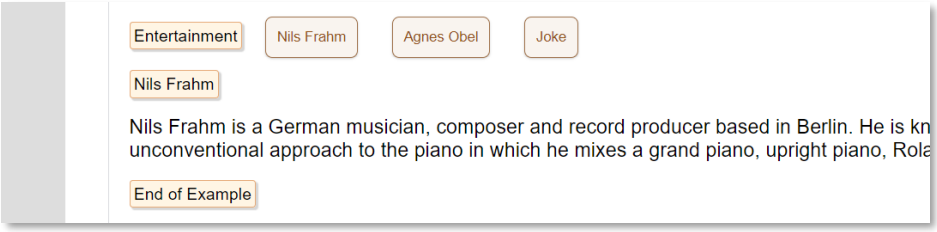


Figure 6.25 SahARIAn rendering of Tabs example without ARIA tabs semantics.

As may be observed, the interface now features three simple `<button>` elements and their related generic `<div>` containers.

6.6.5 SahARIAn Landmark roles

While Document Structure roles (Chapter 6.5.1) outlined the structure of the content, Landmark roles (or simply landmarks) describe the structure of the whole page by identifying large content areas. They are used by screen readers for navigation because they provide information about users' orientation and how to easily move between sections. Ideally, all its content should be placed within a landmark.

There are eight ARIA landmarks, some of which map to native HTML5 elements, listed in parentheses: `banner` (`<header>`), `navigation` (`<nav>`), `main` (`<main>`), `complementary` (`<aside>`), `contentinfo` (`<footer>`), `search`, `form` (`<form>`), `application`. SahARIAn style sheets represent these large content areas with a thin light gray border (1px) and a `display="block"` property.

The NASA Web site[79] (Figure 6.26) provides an improvable example of proper use of ARIA landmarks. The content of its homepage is divides into two generic `<div>` elements, identified only by their `id` attributes:

- the first one, `id="navigation"`, exhibits the same characteristics of a banner: it contains the logo, the name of the organization, a search field and a navbar.

[79] NASA.gov | Homepage. Online available at https://www.nasa.gov/. Last accessed Oct. 10, 2021.

- here, the search field has been implemented with a `role="search"`. Indeed, in the SahARIAn visualization (Figure 6.27), it is the only one that is rendered with a thin gray frame.

- the second one, `id="main"`, as the name implies, represents the dominant content of the Web page.

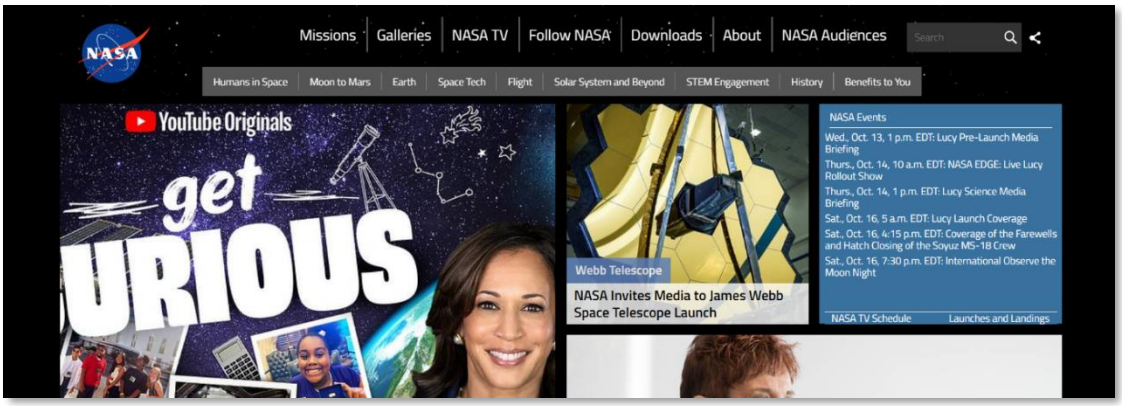


Figure 6.26 NASA.gov homepage.

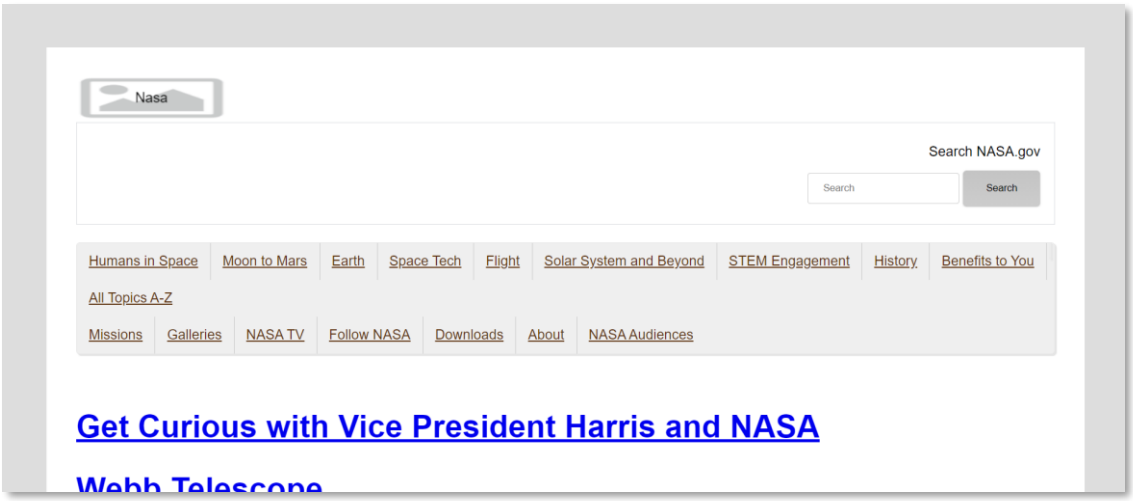


Figure 6.27 SahARIAn rendering of NASA.gov homepage.

On the other hand, Figure 6.28 shows how the same page would look in SahARIAn mode if the two site-specific `id` attributes were replaced with universal ARIA landmarks, namely `role="banner"` and `role="main"`. These two roles added to improve the accessibility of

the page are here framed in red for didactic purposes, but would actually be rendered in SahARIAn's CSS with a gray frame as all other ARIA landmarks.

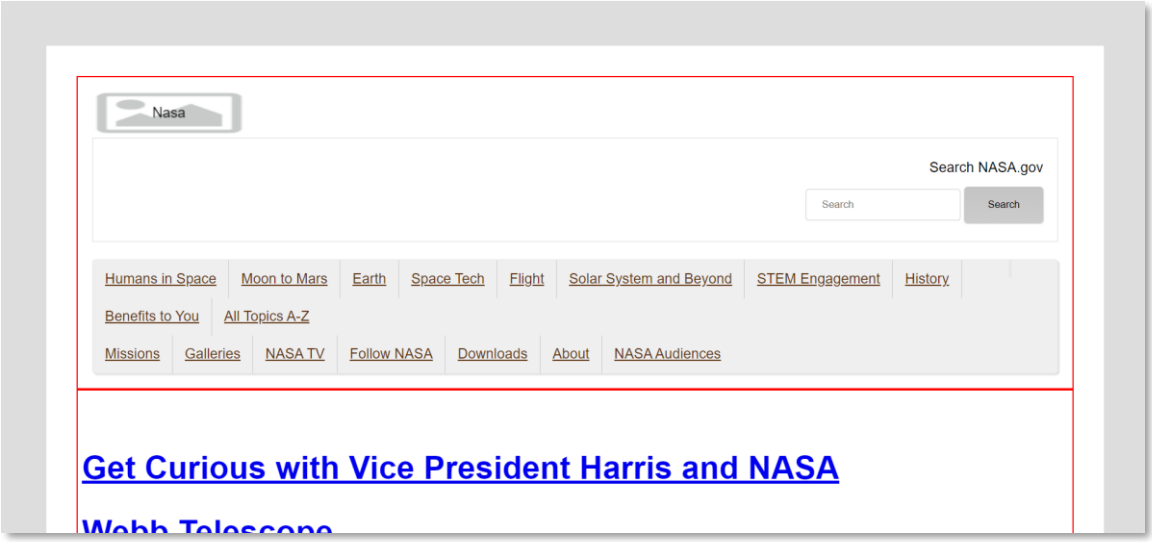


Figure 6.28 SahARIAn rendering of NASA.gov homepage.

The `main` role is significant because it represents an alternative to "skip to main content" links. Thus, its inclusion is recommended, even in the form of the native HTML5 `<main>` element.

In conclusion, by classifying and labeling large sections of a Web page, structural information conveyed visually through SahARIAn layout is represented programmatically. The sighted programmer can therefore understand on the fly how the keyboard navigation of a screen reader proceeds and how sections are nested or aligned with each other.

Conclusion

The comprehensive proceedings of this dissertation have been redacted with the intention of shedding new light on the pivotal issue of technological accessibility for users with diverse disabilities. Even today (2021), indeed, the digital divide separating the possibilities available to non-disabled users from those with various disabilities are still wide. The ongoing massive use of Ajax and asynchronous technologies that have created an increasingly dynamic Web with Rich Internet Applications has further complicated things and led the World Wide Web Consortium (W3C) and its WAI initiative to remedy the situation by pioneering WAI-ARIA. Even with the introduction of ARIA notation, this benefit disparity unfortunately persists although there is a wide dedicated digital library promoted by the W3C itself and legislation in favor of web accessibility has been promoted at national (Australia, Canada, France, Germany, Italy, USA, etc.) and international (European Union) levels.

To investigate the potential reasons for this constant global Web impartiality, a number of scientific studies have been reviewed that have brought to light several issues:

- difficulties in interpreting accessibility guidelines;
- too high number of guidelines themselves, leading to an online paper maze of documents that refer to one another;
- inadequacy in covering certain real errors that are not even included in the specifications;
- difficult balance between obtaining a perfectly accessible site and the desire to deliver a glitzy, dynamic graphical design;
- little importance given to this delicate issue by clients and stakeholders;
- need for webmasters and developers who have a very technical and detailed knowledge of accessibility issues and how to solve them;
- a strong and expensive increase in required time.

For such a purpose, it has been investigated which could be a useful contribution to this problem, realizable in the arc of a second cycle degree programme. Therefore, first this

dissertation listed a review of the accessibility testing tools available on the market in all their various forms, strengths and weaknesses. Secondly, after having empirically observed the lack of a software that would allow sighted developers to instantly grasp the accessibility issues of their Web sites, SahARIAn was chosen and improved. One of the reasons behind this is that it is a software that can be conveniently used as a browser extension of Chrome and Firefox and that does not even rely on a remote server, thus allowing even editing on delicate projects. This plug-in can be used, and indeed is recommended, from the very first drafts of Web pages, and exploits the usual edit-reload-watch cycle of standard Web developments. Developers relying on SahARIAn, in fact, can directly observe the effects of the latest edit cycle without having to depend on indirect witnesses, whether they are users with disabilities employed as testers, or senior accessibility experts. Development activities would not be blocked and there wouldn't be loss of time or costs.

SAhARIAn is a software still under development but includes fully functional algorithms, such as the image handling, the deactivation and activation of styles and CSS style sheets, and the calculation of the accessibility name and description, as well as almost all the standalone widgets (checkbox, radio, switch, tabpanel, tab, etc.), several composite widgets (combobox, tablist, tree, treegrid, etc.) and other ARIA roles such as landmarks and live regions.

At the end of this project, it can be affirmed with certainty that the realized system allows an easy and immediate analysis of the accessibility of a Web site, that doesn't rely on a frozen list of errors placed side by side to an out of context line of code, but that visualizes them directly and dynamically on the Web page: all the elements correctly implemented, in fact, will be made appealing and interactive, while those that present accessibility issues for assistive technologies, will display unpleasant and bizarre layouts and will not be interactive even with the mouse.

However, SahARIAn also suffers from limitations, since it lacks advice and assistance during development and error fixing. As WAVE and ARC Toolkit operated instead, our software lacks labels or icons positioned in critical points of the Web page giving advice on how to improve them, and perhaps even attaching links (as Google Lighthouse does) to guidelines and useful resources with the best solutions to solve them. Secondly, it would be useful to develop in a more detailed way the difference between document and application modes, both used by screen readers to navigate and/or interact with Web pages. In fact, the first one allows the user to move freely within the Web page even on contents that cannot be selected

from the keyboard (e.g. text areas), but does not allow any interaction. The second one, on the other hand, is necessary when interactive operations are to be performed on the page, such as filling in a form or clicking a link. By introducing the distinction between these two reading modes, the developer could test in a more precise manner how his or her Web page adapts when one or the other mode is activated.

To conclude, it is still necessary to refine the software by inserting the ARIA rules of the missing widgets and by taking into account some little features such as, for instance, the display of the alternative text in images when it is given by the attribute `aria-labelledby`, and not only by `alt` or `aria-label` (as it currently functions).

In spite of its strengths and also of its limitations, SahARIAn can represent a new powerful source of inspiration for a generation of manual accessibility testing tools focusing on the structure of Web pages and not on their presentation and style. If, therefore, the development and refinement of our software would open several possibilities for further research and innovation, this would certainly make us proud and delighted.

References

A. Alsaeedi. (2020). *Comparing Web Accessibility Evaluation Tools and Evaluating the Accessibility of Webpages: Proposed Frameworks*, in Information 11, no. 1.

A. Campbell, M. Cooper, A. Kirkpatrick, and J. O. Connor. (2018). *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/2018/REC-WCAG21-20180605/. Last accessed Sept. 28, 2021.

A. Champbell, M. Cooper, and A. Kirkpatrick. (2020). *Techniques for WCAG 2.1*. W3C, W3C Note. Online available at https://www.w3.org/WAI/WCAG21/Techniques/. Last accessed Sept. 28, 2021.

A. Henka, and G. Zimmermann. (2014). *Persona Based Accessibility Testing*. In Stephanidis C. (eds) HCI International 2014 - Posters' Extended Abstracts. HCI 2014. Communications in Computer and Information Science, vol 435. Springer, Cham.

A. P. Freire, C. M. Russo, and R. P. M. Fortes. (2008). *A survey on the accessibility awareness of people involved in web development projects in Brazil*. In Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A) (W4A '08). Association for Computing Machinery, New York, NY, USA, 87–96.

ADA.gov | *Americans with Disabilities Act*. (2009). Online available at https://www.ada.gov/pubs/adastatute08.htm. Last accessed Oct. 9, 2021.

Australian Government Federal Register of Legislation | *Disability Discrimination Act*. (1992). Online available at https://www.legislation.gov.au/Details/C2018C00125. Last accessed Oct. 9, 2021.

B. Kelly, D. Sloan, L. Phipps, H. Petrie, and F. Hamilton (2005). *Forcing Standardization or Accommodating Diversity?: A Framework for Applying the WCAG in the Real World*. In Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A), New York, NY, USA, pp. 46–54.

B. Sierkowski. (2002). *Achieving web accessibility*. In Proceedings of the 30th annual ACM SIGUCCS conference on User services (pp. 288-291).

C. Colwell, and H. Petrie (2001). *Evaluation of guidelines for designing accessible Web content*. SIGCAPH Comput. Phys. Handicap., 70, 11-13.

C. Power, A. Freire, H. Petrie, and D. Swallow. (2012). *Guidelines are only half of the story: accessibility problems encountered by blind users on the web*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12). Association for Computing Machinery, New York, NY, USA, 433–442.

Chrome Developers | *Manifest V3*. (2021). Online available at https://developer.chrome.com/docs/extensions/mv3/overview/. Last accessed Oct. 3, 2021.

Cisco Security | *Cisco 2016 Annual Security Report*. (2016). Online available at http://mkto.cisco.com/rs/564-whv-323/images/cisco-asr-2016.pdf. Last accessed Oct. 3, 2021.

D. Rømen, and D. Svanæs. (2008). *Evaluating web site accessibility: validating the WAI guidelines through usability testing with disabled users*. In Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges (NordiCHI '08). ACM, New York, NY, USA, 535- 538

D. Rømen, and D. Svanæs. (2012). *Validating WCAG versions 1.0 and 2.0 through usability testing with disabled users.* In Universal Access in the Information Society, 11(4), 375-385

D. Sato, H. Takagi, M. Kobayashi, S. Kawanaka, and C. Asakawa. (2010). *Exploratory Analysis of Collaborative Web Accessibility Improvement*, in ACM Trans. Access. Comput. 3, 2, Article 5.

D. Sloan, A. Heath, F. Hamilton, B. Kelly, H. Petrie, and L. Phipps. (2006). *Contextual web accessibility - maximizing the benefit of accessibility guidelines*. In Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A): Building the mobile web: rediscovering accessibility? (W4A '06). Association for Computing Machinery, New York, NY, USA, 121–131.

D. Sloan, P. Gregor, M. Rowan, and P. Booth. (2000). *Accessible accessibility*. In Proceedings on the 2000 conference on Universal Usability (CUU '00). Association for Computing Machinery, New York, NY, USA, 96–101.

EUR-lex | *Web Accessibility Directive*. (2012). Online available at https://eur-lex.europa.eu/eli/dir/2016/2102/oj. Last accessed Oct. 9, 2021.

F. Alonso, J. L. Fuertes, Á. L. González, and L. Martínez. (2010). *On the testability of WCAG 2.0 for beginners*. In Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A).

G. Brajnik. (2004). *Comparing accessibility evaluation tools: a method for tool effectiveness*. In Universal Access in the Information Society, 3:252{263.

G. Brajnik. (2009). *Validity and reliability of web accessibility guidelines*. In Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility (Assets '09). Association for Computing Machinery, New York, NY, USA, 131–138.

G. Goggin, and C. Newell. (2007). *The business of digital disability*. In The Information Society, 23(3), 159-168.

Gazzetta Ufficiale della Repubblica Italiana | *Disposizioni per favorire l'accesso dei soggetti disabili agli strumenti informatici*. (2004). Online available at https://www.gazzettaufficiale.it/eli/id/2004/01/17/004G0015/sg . Last accessed Oct. 9, 2021.

Gov.UK report. (2020). Online available at https://technology.blog.gov.uk/2020/02/24/why-the-gov-uk-design-system-team-changed-the-input-type-for-numbers/. Last accessed Sept. 29, 2021.

J. Craig, M. Cooper, S. McCarron, R. Schwerdtfeger, and J. Diggs. (2017). *Accessible Rich Internet Applications (WAI-ARIA) 1.1*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/2017/REC-waiaria-1.1-20171214/. Last accessed Sept. 30, 2021.

J. Diggs, B. Garaventa and M. Cooper. (2018). *Accessible Name and Description Computation 1.1*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/accname-1.1. Last accessed Sept. 30, 2021.

J. F. Spellman, J. Richards, and J. Treviranus. (2015). *Implementing ATAG 2.0*. W3C, W3C Note. Online available at https://www.w3.org/TR/2015/NOTEIMPLEMENTING-ATAG20-20150924/. Last accessed Sept. 30, 2021.

J. Hailpern, L. Guarino-Reid, R. Boardman, and S. Annam. (2009). *Web 2.0: blind to an accessible new world*. In Proceedings of the 18th international conference on World wide web (WWW '09). Association for Computing Machinery, New York, NY, USA, 821–830.

J. J. Garrett (2005). *Ajax: A new approach to web applications*. Adaptive Path Inc, 30.

J. Lazar, A. Dudley-Sponaugle, and K. Greenidge. (2004). *Improving Web Accessibility: A Study of Webmaster Perceptions*. Computers and Human Behavior, 20(2):269{288, 2004.

J. Richards, J. F. Spellman, and J. Treviranus. (2015). *Authoring Tool Accessibility Guidelines (ATAG) 2.0*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/2015/REC-ATAG20-20150924/. Last accessed Sept. 30, 2021.

J. Scheuhammer and M. Cooper. (2014). *WAI-ARIA 1.0 User Agent Implementation Guide*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/2014/REC-wai-aria-implementation-20140320/. Last accessed Sept. 30, 2021.

M. Ballantyne, A. Jha, A. Jacobsen, J. Scott Hawker, and Y. N. El-Glaly. (2018). *Study of Accessibility Guidelines of Mobile Applications*. In Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia (MUM 2018). Association for Computing Machinery, New York, NY, USA, 305–315.

M. Butkiewicz, H. V. Madhyastha, and Vyas Sekar. (2011). *Understanding website complexity: measurements, metrics, and implications*. In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC '11). Association for Computing Machinery, New York, NY, USA, 313–328.

M. Cooper, J. O. Connor, and A. Kirkpatrick. (2016). *Understanding WCAG 2.0*. World Wide Web Consortium (W3C), W3C Working Group Note. Online available at https://www.w3.org/TR/UNDERSTANDING-WCAG20/Overview.html. Last accessed Sept. 30, 2021.

M. Cooper, J. O. Connor, and A. Kirkpatrick. (2021). *Understanding WCAG 2.1*. World Wide Web Consortium (W3C), W3C Note. Online available at https://www.w3.org/WAI/WCAG21/Understanding/. Last accessed Sept. 30, 2021.

M. Greeff, and P. Kotzé. (2009). *A lightweight methodology to improve web accessibility*. In Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists, New York, NY, USA, pp. 30–39.

M. Vigo, J. Brown, and V. Conway. (2013). *Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests*, in Proceedings of the 10th

International Cross-Disciplinary Conference on Web Accessibility (W4A '13). Association for Computing Machinery, New York, NY, USA, Article 1, pp. 1–10.

Nucleus Research Note: The Internet is Unavailable (2019). Online available at https://nucleusresearch.com/research/single/the-internet-is-unavailable/. Last accessed Sept. 30, 2021.

O. Gaggi, G. Quadrio and A. Bujari. (2019). *Accessibility for the Visually Impaired: State of the Art and Open Issues*. In 16th IEEE Annual Consumer Communications & Networking Conference (CCNC), pp. 1-6, doi: 10.1109/CCNC.2019.8651676.

R. Calvo, F. Seyedarabi, and A. Savva. (2016). *Beyond Web Content Accessibility Guidelines: Expert Accessibility Reviews*. In Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (DSAI 2016). Association for Computing Machinery, New York, NY, USA, 77–84.

R. Lopes, D. Gomes, and L. Carriço. (2010). *Web not for all: a large scale study of web accessibility*. In Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A). Association for Computing Machinery, New York, NY, USA, Article 10, 1–4.

R. Schwerdtfeger, M. King, J. Scheuhammer, L. Pappas, Z. Bijl, M. Cooper, J. Ku, and J. Nurthen. (2019). *WAI-ARIA Authoring Practices 1.1*. W3C, W3C Note. Online available at https://www.w3.org/TR/2019/NOTE-waiaria-practices-1.1-20190814/. Last accessed Sept. 30, 2021.

R. Williams, and S. Brownlow (2020). *The Click-Away Pound Report 2019*. Online available at https://clickawaypound.com/downloads/cap19final0502.pdf. Last accessed Sept. 30, 2021.

S. B. L. Ferreira, R. C. dos Santos, and D. S. da Silveira. (2007). *Panorama of Brazilian Web Accessibility*. In Proceedings of the XXXI ANPAD Meeting-EnANPAD.

S. K. Kane, J. A. Shulman, T. J. Shockley, and R. E. Ladner. (2007). *A web accessibility report card for top international university web sites*. In Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A) (W4A '07). Association for Computing Machinery, New York, NY, USA, 148–156.

T. Berners-Lee, R. Cailliau, J. Groff, and B. Pollermann. (1992). *World-Wide Web: The Information Universe*. In Internet Research, Vol. 2 No. 1, pp. 52-58.

T. Burchardt. (2004). *Capabilities and disability: the capabilities framework and the social model of disability*. In Disability & society, 19(7), 735-751.

T. Schulz, and K. Skeide Fuglerud. (2012). *Creating personas with disabilities*. In Miesenberger, K., Karshmer, A., Penaz, P., Zagler, W. (eds.) ICCHP 2012, Part II. LNCS, vol. 7383, pp. 145–152. Springer, Heidelberg.

T. Sullivan, and R. Matson. (2000). *Barriers to use: Usability and content accessibility on the web's most popular sites.* In Proceedings of the 2000 Conference on Universal Usability (CUU '00), Arlington, VA, USA, 16–17 November 2000; ACM: New York, NY, USA, pp. 139–144.

U.S. Access Board | *Section 508*. (1998). Online available at https://www.access-board.gov/ict/. Last accessed Oct. 9, 2021.

V. Rubano, and F. Vitali. (2020). *Experiences from declarative markup to improve the accessibility of HTML*. In Balisage: The Markup Conference.

W. Chisholm, G. Vanderheiden, and I. Jacobs. (1999). *Web Content Accessibility Guidelines 1.0*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/WAI-WEBCONTENT/. Last accessed Sept. 28, 2021.

W. Chisholm, G. Vanderheiden, and I. Jacobs. (2000). *Techniques for Web Content Accessibility Guidelines 1.0*. W3C, W3C Note. Online available at https://www.w3.org/TR/WCAG10-TECHS/. Last accessed Sept. 28, 2021.

W. Chisholm, J. Slatin, and J. White (2008). *Web Content Accessibility Guidelines (WCAG) 2.0*. W3C, W3C Recommendation. Online available at https://www.w3.org/TR/WCAG20/. Last accessed Sept. 28, 2021.

Y. Puzis, Y. Borodin, A. Soviak, V. Melnyk, and I. V. Ramakrishnan. (2015). *Affordable web accessibility: a case for cheaper ARIA*. In Proceedings of the 12th International Web for All Conference (W4A '15). Association for Computing Machinery, New York, NY, USA, Article 32, 1–4.

Zaidi, and T. Burchardt. (2005). *Comparing incomes when needs differ: equivalization for the extra costs of disability in the UK*. In Review of income and wealth, 51(1), 89-114.

Accessibility Testing Checkers and other Tools:

AChecker. (2011). Online available at https://achecker.achecks.ca/checker/index.php. Last accessed Sept. 26, 2021.

Axe. (2020). Online available at https://www.deque.com/axe/. Last accessed Sept. 26, 2021.

ChromeLens. (2016). Online available at https://github.com/chromelens/chromelens. Last accessed Sept. 26, 2021.

Funkify. (2020). Online available at https://www.funkify.org/. Last accessed Setp. 29, 2021.

HeadingsMap. (2018). Online available at https://github.com/dzc34/headingsMap. Last accessed Sept. 29, 2021.

Lighthouse. (2018). Online available at https://developers.google.com/web/tools/lighthouse. Last accessed Sept. 26, 2021.

MSF&W | Color Contrast Ratio Calculator. Online available at https://www.msfw.com/Services/ContrastRatioCalculator. Last accessed Sept. 29, 2021.

NerdeFocus. (2019). Online available at https://github.com/wizzyfx/nerdeFocusPlugIn. Last accessed Sept. 29, 2021.

Tenon. (2017). Online available at https://tenon.io. Last accessed Sept. 26, 2021.

TPGi | Color Contrast Analyser. Online available at https://www.tpgi.com/color-contrast-checker/. Last accessed Sept. 29, 2021.

WAVE. (2019). Online available at https://wave.webaim.org/. Last accessed Sept. 26, 2021.

Web sites used to showcase SahARIAn's interface:

"WAI-ARIA Authoring Practices 1.1", 2019 | Example of Tabs with Manual Activation. Available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/tabs/tabs-2/tabs.html. Last accessed Oct. 9, 2021

"WAI-ARIA Authoring Practices 1.1", 2019 | Modal Dialog example. Available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/dialog-modal/dialog.html. Last accessed Oct. 7, 2021.

"WAI-ARIA Authoring Practices 1.1", 2019 | Feed example. Online available at https://www.w3.org/TR/wai-aria-practices-1.1/examples/feed/feedDisplay.html. Last accessed Oct. 7, 2021.

NASA.gov | Homepage. Online available at https://www.nasa.gov/. Last accessed Oct. 10, 2021.

NYPD Official Web site | Homepage. Online available at https://www1.nyc.gov/site/nypd/index.page. Last accessed Oct 13, 2021.

Ryanair Web site | Homepage. Available at https://www.ryanair.com/. Last accessed Oct 10, 2021

Trenitalia Official Web site | Homepage. Online available at https://www.trenitalia.com/. Last accessed Oct. 17, 2021.

University of Bologna | Homepage. Online available at https://www.unibo.it/en. Last accessed Sept. 26, 2021.

University of Washington Web site | Homepage. Online available at http://www.washington.edu/. Last accessed Oct. 15, 2021.