Link to the Balisage Proceedings Home Page at https://www.balisage.net/Proceedings
Link to this paper (entry page with links) at https://www.balisage.net/Proceedings/vol25/html/Vitali01/BalisageVol25-Vitali01.html

Balisage
The Markup Conference

Balisage Paper: Experiences from declarative markup to improve the accessibility of HTML

Vincenzo Rubano
PHD student
Department of Computer Science and Engineering, University of Bologna
<vincenzo.rubano@unibo.it>

Fabio Vitali
Professor
Department of Computer Science and Engineering, University of Bologna
<fabio.vitali@unibo.it>

Balisage: The Markup Conference 2020
July 27 - 31, 2020

How to cite this paper

Rubano, Vincenzo, and Fabio Vitali. “Experiences from declarative markup to improve the accessibility of HTML.” Presented at Balisage: The Markup Conference 2020, Washington, DC, July 27 - 31, 2020. In *Proceedings of Balisage: The Markup Conference 2020*. Balisage Series on Markup Technologies, vol. 25 (2020). https://doi.org/10.4242/BalisageVol25.Vitali01.

Abstract

Producing accessible content for the Web is a rather complex task. Standards, rules and principles that offer largely useful recommendations for accessible content do indeed exist, but they are not adequately enforced and supported by actual implementations. It is fairly frequent for content authors to produce material that ends up not being accessible without even noticing it, even when using additional tools and services.

Yet, most of the existing recommendations for accessible web resources center around the addition of reasonably simple markup with a clear declarative purpose in their design. How therefore is it possible that producing truly accessible content is such a rare occurrence?

In this paper, we posit that an important justification of this, in addition to well-known lack of interest and lack of awareness, is the difficulty of evaluating and perceiving the correctness or wrongness of the generated assistive markup by non-disabled content authors and tool designers. Designers have serious difficulties when evaluating the effectiveness and correctness of the accessibility of their works, and existing tools do little or nothing to reduce the "handicap".

Under these assumptions, we aim to describe an innovative approach based on declarative markup to improve the design and evaluation the accessibility of web pages. In particular, our strategy encompasses the combined usage of a declarative framework of accessible web components, capable of enforcing best-practices and conformance to accessibility standards, as well as automated tools to test for the accessibility of web content and, in addition, a new approach to manual tools to let developers and content creators examine visually the accessibility issues so that they can make sense of their impact on people with disabilities.

Introduction

Web Content Accessibility Guidelines (WCAG 2.1) is the reference international standard when it comes to digital content accessibility. They define the principles and guidelines that developers must adhere in order to produce accessible content, as well as success criteria and conformance level to assess the accessibility of their work and its conformance to the guidelines, settling a framework for evaluating the accessibility of digital content in a technology-agnostic way. Many support documents illustrating how to meet these guidelines that contain practical examples for specific technologies are also provided, even offering recommendations of best practices to follow, describing the most common failures and the most appropriate remediations for them. Yet, we argue that producing accessible content for the Web can still be considered a pretty complex task.

First off, it is technically possible to produce inaccessible content without even noticing it. Testing for accessibility, in fact, requires additional tools and/or services that are outside of the usual workflows adopted for creating web content. Such tools are provided in many different flavours (web services, browser extensions, automated testing frameworks, etc) and are sometimes built-in within the browser; however, they still are only optionally involved into developers' workflows, thus from a practical point of view their use can be skipped entirely. Additionally, understanding the output of such tools often requires specific knowledge about web accessibility, which may or may not be available to developers and that they may or may not be willing to acquire.

Considering this and the fact that web standards (i.e. HTML, CSS and JavaScript) allow to produce accessible content just as easily as to produce its inaccessible versions, we must conclude that accessibility support is nowadays a completely optional and opt-in feature in most web design processes, and that it is not adequately enforced and supported by actual implementations. Developers frequently end up producing inaccessible content without even being aware of it.

Lacking specific competencies and supporting tools, it is very hard for content authors and designers without disabilities to perceive what the effect of accessibility issues is, what aree the differences between inaccessible content and its accessible equivalent, and ultimately how accessible their product is. New tools must be provided and existing ones must be improved to better fill this gap.

Since the overwhelming majority of web designers and developers don't have disabilities, their full range of senses is preventing them to perceive the difficulties and the problems that their products generate on people with disabilities. Thus, they have to base their implementation decisions on third party reports, either from experts, testers or automatic tools. In a way, their sightedness acts as a handicap in the perception of the correctedness of their markup.

Interestingly, the key approach of assistive markup is to enrich the content of web documents with *declarative* annotations, so that specialized applications can render the content in a perceivable manner to specific users. Such declarative annotations do not deal with the visual representation of the content, but, rather, with the attribution of semantic and structural roles to specific fragments of the content, and, traditionally, XML designers are used to and well versed into the use of declarative markup that doe impact visual rendering.

Therefore, in this paper we describe a vision for the design of accessible web content based on the declarative markup of web pages aiding humans as well as automated and manual testing tools. We describe a declarative framework of accessible web components capable of enforcing best-practices and conformance to accessibility standards, so that developers can easily produce accessible markup without specific awareness. We then describe a tool to help sighted designers perceive visually the assistive markup without interferences of the "usual" visual rendering. Finally, we describe a testing approach based on a rule-based engine to identify and report on run-time accessibility issues that would be quite hard to catch statically on HTML markup.

Key in our approach is to provide deep integration of existing tools for accessibility design, implementation and testing, so that it can be carried out within a standard web development workflow without particular effort by the developers. Each potential accessibility issue must be taken care of automatically or, at worst, presented so that developers with no or basic competencies about accessibility can make sense of it, find the specific code fragment causing it, fix it, and be able to immediately verify whether the fix solved the issue or not. Our approach is thus based on enabling developers and authors to directly perceive (e.g., visually) the accessibility quality of their content and the impact of eventual accessibility issues.

The remaining of this paper is organized as follows. In section “Related work” we examine existing accessibility standards, justify the reasons why a declarative framework is critical in our vision for improving web accessibility and briefly analyze the most significant tools and services that can be related to our proposal. In section “Problems for the sighted developers” we provide a deeper look at our approach, with practical examples on the use of the proposed framework to illustrate why and how it can improve the current situation with regards to accessibility design, and describe the purpose of both the automated and manual accessibility testing tools we propose. The foundational principles and some technical details about the implementation of our approach will be discussed in section “Helping the sighted developers”. Finally, in section “Conclusions and future developments” some final considerations are made, and some ideas for future developments and possibly related research topics are proposed.

Related work

Accessibility standards

Currently, information on how to develop accessible websites and application and author accessible content is spread across various W3C recommendations and their related support documents. The Web Content Accessibility Guidelines (WCAG) 2.1 wcag21 are the basis of such documentation, as they settle the four main principles which lay the foundations for anyone to access and use Web content. Accessible Web means content that is:

Perceivable.	Information and user interface components must be presentable to users in ways they can perceive. This means that users must be able to perceive the information being presented (it can't be invisible to all of their senses).
Operable.	User interface components and navigation must be operable. This means that users must be able to operate the interface (the interface cannot require interaction that a user cannot perform).
Understandable.	Information and the operation of user interface must be understandable. This means that users be able to understand the information as well as the operation of the user interface (the content operation cannot be beyond their understanding).
Robust.	Content must be robust enough that it can be interpreted reliably by a wide variety of user ag including assistive technologies. This means that users must be able to access the content as technologies advance (as technologies and user agents evolve, the content should remain accessible).

other than the four accessibility foundational principles, WCAG 2.1 provides guidelines, i.e. abstract rules to follow in order to produce content respecting such principles. For each guideline, so called "success criteria" are provided, i.e. practically testable statements to check wether content conforms to it or not. This is clearly explained in "Understanding WCAG 2.1" understandingwcag21, an informative (thus non-normative) document produced by the Accessibility Working Group at W3C providing any additional information on how to interpret the WCAG recommendation.

Information on how to comply with such guidelines when using specific technologies, along with practical examples of how to do that, are illustrated in Techniques for WCAG 2.1" wcag21techniques, another informative document being constantly updated over time by the same group at W3C.

While such documents and attributes available in HTML where enough to make websites accessible, the advent of AJAX and complex desktop-like web applications introduced many new challenges for web accessibility. The WAI-ARIa specification aria11, now at version 1.1, was born to address such challenges, especially in case of dynamic content and advanced user interface controls. To support developers implementing this specification in their projects, the WAI-ARIA authoring practices aria11authoringpractices is provided, a W3C working group note that explains how to make the most commonly used design patterns accessible leveraging this specification and providing code examples to implement them.

A W3C recommendation on how WAI-ARIA support should be implemented by user agents is available, but it currently refers to version 1.0 of the WAI-ARIA specification; its equivalent for the latest version is still in draft aria11implementation. A more in depth discussion on how user agents should be made accessible, and what information assistive technologies could expect to be exposed through native platform accessibility APIs, is available in User Agent Accessibility Guidelines (UAAG) 2.0 uaag20, published as a W3C working group note.

Finally, another W3C recommendation has been specifically crafted with regards to authoring tools accessibility, i.e. any tool that allow to produce, edit or manipulate in any way content, including automatic conversions. Authoring Tools Accessibility Guidelines (ATAG) 2.0 atag20. This recommendation is divided in two parts: in part A principles, guidelines and testable success criteria are provided to ensure that such tools can be used by people with disabilities; in part B, the same is done with regards to the ability of such tools to allow and encourage end users, not limiting to those with disabilities, to produce

content that is accessible and, in any case, to not deteriorate its accessibility during automatic processes (i.e. format conversions). Guidance on how to meet such guidelines, perform conformance testing and in dept practical examples are provided in a separate document implementingatag20.

Declarative frameworks and the Web

Implementing a declarative framework on top of web technologies to make their use more effective efficient is not new, as other examples have been already proposed in the literature over time. Hanu Koschnicke erdeclarativewebprogramming, for example, describe a declarative framework to supp implementation of web-based systems to manipulate data stored in relational databases. Li et all echarts proposed ECharts, an open-source, web-based, cross-platform framework that supports the rapid construction of interactive visualization and is regarded as a leading visualization development tool.

The more general topic of improving web application development by leveraging declarative languages has also been discussed in the literature. In applicationembedding, Wild presents Application Embedding, a novel approach to application development which allows all aspects of a web application, including its business-logic, to be programmed declaratively. Lorenz & Rosenan declarativeweb20 argue that the contextual nature of Web 2.0 content needs a better representation, and that the same can also be used to better describe the rich interfaces for applications building on that contextual content, identifying in a declarative way of representing Web 2.0 data such representation.

While the usage of a declarative framework to improve both specific and general aspects of web programming has already been adopted in the past with various success degrees, it is legitimate to wonder why a declarative framework is necessary in the specific case of web accessibility. True, WCAG 2.1 wcag21 settle principles and guidelines that content should conform to in order to be considered accessible, as well as practically testable success criteria to assess its conformance to the standard according to determined levels (A, AA and AAA), and documents such as wcag21techniques provide practical examples and detailed resources on how to create such content leveraging specific technologies. Yet, web authors still have a hard time understanding and applying accessibility guidelines, as they are considered too technical, and not supporting adequately problem identification and solving personasaccessibilitytesting. This can be mitigated by providing better support for markup that is "accessible by default", i.e. without any specific intervention by the developer, and creating more advanced testing tools that do not require having previous competencies about web accessibility in order to detect issues and understand how to fix them.

Regarding the generation of accessible markup, HTML 5 can be considered a step in the right direction. This version (and the following ones) added to the language many features commonly used in web applications that, not being available in a standardized implementation, had to be implemented leveraging external solutions (think of audio and video playback) with a varying degree of accessibility. However, many commonly used interactive widgets have not natively become part of HTML and still require markup and code that may or may not be accessible. Other language features have been included in later versions, but are not supported properly by browsers and/or assistive technologies: for instance, according to the standard, autocomplete fields should be populated by a datalist element; yet, when using it, there is no indication that the field it is associated with supports autocompletion, and it looks exactly like a regular text field, so that screen readers treat the field exactly as a plain textbox.

Another commonly used widget that is often the source of subtle accessibility problems is the modal dialog, which, according to HTML 5, should be implemented by element *dialog*; however, many browsers still do not support this element at all (e.g., Apple Safari), others require its support to be enabled explicitly in their advanced preferences (e.g., Mozilla Firefox), and others implement it natively *but* with critical accessibility issues. Finally, many controls commonly used by web applications (for instance, tabs a associated tab panels, menu bars, toolbars, trees, to name a few) have still to find a good and agreed markup representation in HTML to build good accessibility support upon. In addition to this, even i best possible scenario, no guarantee is made about whether all these widgets will be implemented consistently across browsers, or that styling them will be supported in the same way.

Due to this, many controls required by complex web applications have to be implemented by leveraging generic HTML tags enhanced with JavaScript code. Such elements can be made accessible by using the WAI-ARIA specification aria11, which allows to enrich the semantics of HTML elements by adding markup (e.g., specific attributes, and in particular the *role* attribute) that defines the semantics of the element in terms of accessibility. However, in this scenario the developer is responsible for manually implementing the exact behavior expected by assistive technology users for each element: marking an element as having a certain ARIA role is a promise, but the developer is responsible for fulfilling it: differently than HTML elements, ARIA roles do not directly cause browsers to provide keyboard behaviours or styling aria11authoringpractices.

Accessibility testing tools

With regards to accessibility testing, there is no automated tool that can compete with a human in terms of quality and depth of the analysis. Unfortunately, manual accessibility testing is a costly process in time and money, and requires specific and non-trivial competence. Thus, most projects in the real world are unwilling or stingy in spending about this. in truth, many automated tools have been proposed over time to facilitate web developers and content authors in identifying accessibility issues and determining appropriate fixes for them. Such tools come in different forms, and often the same tool is provided in different flavours to suit specific needs in different web development workflows. While they serve the exact same purpose, of course each tool can have characteristics and features that differentiates it from its competitors.

Automated accessibility testing tools are available as web services, both free to use achecker and commercial tenon.io, that allow checking a page for accessibility issues by url or by requiring its source code to be uploaded directly; some even support crawling an entire website starting from a certain page, so as to generate a single report for all accessibility issues contained in a whole site or subsite. Other tools (lighthouse, wave) are available as browser extensions, and allow to quickly test the page currently viewed in the browser for accessibility issues, sometimes highlighting where the errors generate and suggesting solutions for fixing them. More recently, automated testing tools have been made available as frameworks to support test driven development and command line tools axe exist for development workflows in which continuous integration (and therefore automated testing) play a critical role.

Finally, there is another critical aspect to consider when evaluating available automated accessibility testing tools. Thanks to JavaScript, more and more web pages are nowadays built and/or updated dynamically client-side, causing the runtime Document Object Model (DOM) to be vastly different from

the one found in the static source code. Therefore, it is critical for automated accessibility testing tools to be able to work with the actual DOM of the page, and not just the original source code, as otherwise it would potentially miss many accessibility issues.

Problems for the sighted developers

> *Blinded by his sight*
> *Wrapped up in misuse*
> *Another scripter in the night*
>
> — (with apologies to Bruce Springsteen)

Our approach to web accessibility encompasses the combined use of a declarative framework implemented on top of existing web technologies, a deep integration of automated accessibility testing and the creation of innovative tools to let developers, designers and content authors manually test web pages for accessibility issues and directly *perceive* their impact on people with disabilities. We believe that such a combination can help target users produce accessible content.

First we point out that, given the overwhelming presence of people without disabilities among web content developers, there is no direct experience of accessibility issues in most web projects. Non-disabled people cannot perceive the content of their work in the way disabled people would, and cannot perceive personally and precisely the issues they have allowed to arise. The usual edit-reload-watch cycle of most developments efforts does not work for accessibility, because developers cannot directly "watch" the effect of the latest edit cycle, but have to rely on indirect witnesses, be they people with disabilities enrolled as testers, validation tools, or third-party experts. Additionally, the more indirect is this witness, the more difficult it is to fix the issues that were found, since accessibility validation is either blocking all other development activities (and therefore very expensive in the context of a usually late project) or performed in parallel with other activities (which therefore keep on modifying the code base that is being reviewed, making the review itself either pointless, outdated or unaware of additional accessibility issues being introduced in the meantime).

Our approach arises from a different point of view, that is that of

1. abstracting away from developers and designers the burden of determining whether the generated code is accessible or not
2. allowing developers and designers to directly perceive accessibility issues in the generated code
3. bringing accessibility validation closer to, and tighltly integrated with, development frameworks used by developers and designers.

For example, consider the very simple case of representing a plain text field to collect a person's name in a registration form. From a pure HTML point of view it is perfectly legal to ignore accessibility-related markup altogether, creating an inaccessible representation of the field, such as the following.

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

When performing general markup validation, and when checking the rendering on a normal user agent, this representation is perfectly fine. Hence accessibility is not enforced at the implementation level, and specific efforts by the developer are required to check for markup accessibility and to identify the best

approach to improve it. For instance, a blind user needs non-spatial guidance to associate the input field to the text describing its nature and purpose, and simple visual closeness is not meaningful.

In HTML, there are at least five different ways to represent our example in a way that is accessible:

1. replacing the "span" element with a "label" element so that it wraps also the "input" field, e.g.

```
<label>Name: <input type="text"></label>
```

2. replacing the "span" element with a "label" element, but specifying the relationship between "input" field and its label by means of attributes "id" in the input element and "for" in the lab

```
<label for="name-input">Name:</label>
<input type="text" id="name-input">
```

3. Specifying an accessible name for the "input" field by means of the "title" attribute, e.g.

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

4. Using the "aria-label" attribute from the WAI-ARIA specification, e.g.

```
<span>Name:</span>
<input type="text" aria-label="Name:">
```

5. Using the "aria-labelledby" attribute from the WAI-ARIA specification, e.g.

```
<span id="name-input-label">Name:</span>
<input type="text" aria-labelledby="name-input-label">
```

While having all those representations makes it possible to create an accessible text field in different contexts, this introduces a cognitive effort for the developer to understand which one to choose, and the reason why one is preferrable to the other in general and in this specific context. One may legitimately argue that this flexibility is required in order to support a multitude of features, such as allowing for better positioning and styling of both the field and its label, but this richness comes at a cost that in many cases is not acceptable. Better yet, are we sure we really need so much flexibility? Couldn't we achieve astonishing designs differently?

In fact, HTML, when used correctly and precisely, is already mostly accessible. Assistive technologies are available to provide accessible representations of HTML elements as they were originally designed to be used. The problem is given by the number of possible semantical characterizations of HTML elements that are not and cannot be reflected in the actual syntax.

For instance, when `<span>` is used to mark actual spans, and `<button>` is used to mark actual buttons, syntax and semantics coincide and this gives no problem for assistive technologies. But if we write, for instance, `<span onclick="doSomething()"> ... </span>`, the markup is syntactically a *span* but semantically a *button*. This is where accessibility problems arise: unless the author of the markup signals (for instance leveraging the ARIA specification `role="button"`) that there is a conceptual similarity between this span and a button, the assistive technology cannot convey a meaningful accessible representation of the element. Consider also that there are many different of ways to turn a syntactical `span` into a semantic button, such as:

1. plain HTML with inline Javascript:

```
<span onclick="doSomething()"> ... </span>
```

2. plain HTML with separated Javascript:

```
<span class="myClass"> ... </span>
...
document.getElementsByClassName("myClass").onclick = doSomething;
```

3. plain HTML with JQuery:

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

or:

```
<span class="myClass"> ... </span>
...
$(".myClass").on("click", doSomething);
```

4. Angular, React or Vue:

```
<span (click)="doSomething()"> ... </span>     (Angular)
<span ng-click="doSomething()"> ... </span>    (AngularJS)
<span onclick={doSomething()}> ... </span>     (React)
<span v-on:click="doSomething()"> ... </span>  (Vue)
```

5. plain HTML with JQuery and delegation:

```
<span class="myClass"> ... </span>
...
$(document).on("click", ".myClass", doSomething);
```

… and the list could go on.

The last example is particularly vicious, yet extremely common and frequent on the web: rather than binding a callback function to the click event on spans of class "myClass", this code delegates the handling of the click event to the document root node, *but only if* the click happens on an element of class "myClass". This is very frequent and common because it allows the programmer to bind callbacks to elements that do not exist yet, and maybe will be created after some user's actions or loading additional content through an Ajax call: since at binding time (usually before the page is shown to the user) the destination of the callback binding is not in the DOM, the developer binds the callback to a different node (as long that it exists and will end up containing the correct element, e.g., a container or the root node), and delegates to it the task of calling the callback function when the event fires within the intended target element. Thus any element of the containment chain between the root node and the target callback can be chosen as the destination of the binding that transforms a plain `<span>` into a *bona fide* button.

As seen, allowing assistive technologies to help disabled users rely on its ability to identify the correct role, purpose and behavior of the elements in the document, yet the HTML language is neither sufficiently prescriptive to prevent abuses of the semantic characterization of its elements, nor sufficiently descriptive to support features that are common and expected in many web applications, and for which there is no specific markup: from more traditional controls like tabs, collapsible elements, dropdown menubars, modal panels, all the way to more exoteric carousels, accordions, etc., the HTML language is much less expressive than the functions that a little CSS and a little Javascript let browsers provide.

Helping the sighted developers

Having discussed some existing accessibility-related tools and the potential of our approach for improving the current situation, we can now highlight some of the key implementation principles that will be followed in the implementation phase as well as the reasons why they are important.

Declarative markup to the rescue.

According to the original authoritative sources on the topic (sgml),

> Generalized markup is based on two postulates:
>
> 1. Markup should describe a document's structure and other attributes rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.
> 2. Markup should be rigorous so that the techniques available for rigorously-defined objects like programs and data bases can be used for processing documents as well.

A declarative style of markup in web design is therefore the specification of permanent logical, structural and semantic characteristics of all parts and fragments of a web page or application, rather than of their transient and task-specific characteristics such as presentation, in-browser behavior, etc. In our vision, this is exactly what is needed to help assistive tools make web pages and applications accessible and usable by people with disabilities: designers and authors are not expected to provide special services, but just to describe the content and features of the page/application in a sufficiently precise way to allow (existing) assistive technologies to perform their job in the right way and at the right time.

As such, scholars of declarative markup styles learned in the ways of SGML and XML would immediately see the problem and the way to address the problem. They are *the* standard ways to use markup in this world:

1. Create a rich and expressive language that describes the permanent logical, structural and semantic characteristics of the page and application, rather than forcing and stretching the interpretation of generic building blocks originally meant only for presentational purpose.
2. Allow for transient and tasks-specific characteristics (e.g., visual rendering) to be toggled on/off at will and easily replaced with different ones, so as to verify directly the generality and universality of the chosen markup by comparing the effectiveness of different presentations.
3. Validate the result by creating a rule system that can be applied to the final markup to identify violations to best practices or expectations.

Our proposal therefore is threefold:

1. *guarantee that the generated markup is always accessible*. This is achieved by extending the HTML markup language through ready-to-use fragments called *components* that are specific to the logical, structural and semantic characteristics of their intended use, and whose markup is fully accessible by construction.
2. *represent visually the markup for accessibility*. Sighted developers can be made to perceive directly the accessibility markup by replacing the normal presentation of the page with a special visual representation based only on the accessibility markup: styling and positioning choices are deactivated and replaced with ones totally and completely based on the accessibility information conveyed to assistive technologies, and the usual interactive behaviours allowed by the browser are mapped onto the corresponding actions that the page allows to perform via any assistive technology.

The end result is that the developer keeps on using mouse and keyboard and eyes to test the web application that is being designed, but in a different visual context that is completely based on the accessibility markup, one that makes the page understandable and usable proportionally to the correctness of the accessibility markup only.

3. *provide in-browser automated testing*. Including automated tests has two significant advantages First, they can act as a barrier for developers who may intentionally or unintentionally alter th markup generated by our framework, since unfortunately preventing this is technically impos Second, automated tests are performed on the actual DOM corresponding to the generated m so that they can catch runtime issues that could not be taken care of by the framework itself: example,visual issues that could be introduced by styling the markup, such as color contrast sizes problems.

   While there is evidence that only up to 50% of accessibility issues can be caught by fully automated tests automatedaccessibilitytoolsbenchmark, we believe there is potential for a combination of such testing techniques with a declarative framework to increase this number, provided that the framework is designed and implemented appropriately. Yet, manual accessibility testing will always be required, and thus must be part of our approach to improve the current situation.

In order to facilitate their adoption, our tools are designed to be used even without our declarative framework. The automated testing integration provided by the declarative framework enables a developer to test even parts of a web page that are not generated by the framework itself, and our manual accessibility visualizer, Saharian, is useable on any website, without any specific additional requirement other than installing the tool itself.

A framework for sighted developers

The implementation of a declarative framework of accessible web components as required by our approach is a challenging process, and providing a full featured solution is likely to require more iterations over time as well as gathering feedback from the community and acting consequently. The necessity of creating a highly extensible and maintainable solution naturally arises from these simple considerations. We also believe that good documentation is key to the success of such a framework, therefore significant efforts are being dedicated to documenting its components (the public API) as well as its internals, in order to provide web developers and content authors all they need to use it at the maximum of its potential.

By definition, the framework should enforce the generation of accessible markup as much as possible. The nature of a declarative solution helps with this, as the correctness of hierarchical relationships (i.e. prevent using an input of type "radio" outside of a fieldset) can be easily enforced as required in order to generate accessible markup. Not only that, but specifying required parameters when instantiating components (take the case of form control labels as an example) can be enforced as well. When such conditions are not met, the framework should not render the offending component or make it de-facto unusable, rendering the error in an appropriate way and providing instructions on how to fix it instead.

Our idea is based on the extension of the markup language through the use of *components*. Introduced and shamelessly promoted by all three of the major web development libraries currently in favour (Angular, React and Vue, and recently even standardized by W3C), components are small, autonomous modules containing markup, styling and executable code that can be aggregated and composed to build full web applications with reliable and sophisticated functionalities. The HTML language is therefore replaced by an open set of elements each of which is mapped onto a complete component providing for its deployment,

including the markup to make it presented on screen. A framework can then become responsible for ensuring the accessibility of the generated markup, and determining the most suitable HTML markup representation of the many that are possible, shifting this burden away from the developers. By doing so, it also shifts away from the developer the responsibility of looking at guidelines and techniques for implementing that component in a way that is accessible, and most of the effort to determine which solution is the most appropriate to each case.

In contrast to the scenarios described above, consider instantiating the input field to become be as s as writing:

```
<textfield label="Name:">
```

This is clearly not an HTML tag, but a markup placeholder for a *textfield* component handled by our framework. Upon rendering the page on the browser, or through a compilation process, the above markup is automatically converted into a combination of markup, styling and code, whose markup contribution automatically includes accessibility specifications (chosen from any of the above-mentioned approaches): the developer is not faced with the task of studying and choosing solutions, yet the final result is perfectly accessible for disabled users.

As is often the case for newly introduced frameworks, it is very important for our solution to be able to coexist with parts of a web page that do not use it. This would allow developers and content authors to gradually adopt the framework, as well as letting consumers use it since its early development phases, even if it does not include every component they need. We argue that this could significantly increase its adoption rate since the early stages, and allow us to gather feedback even in the earliest development stages. For the same reason, we need to be compatible with older browsers even if adopting the latest and greatest modern web development practices and language features; in this context, however, supported browsers need to be determined by keeping into account the degree of support offered for the essential accessibility features leveraged by the generated markup.

Another important point to note is that our framework is not meant as a replacement for very well established Javascript libraries designed to facilitate web application development, like JQuery, Angular, React or Vue, to name a few. Instead, we want to design it to be low-level enough to be used *in combination with* such libraries.

One might argue that in such a situation a developer could easily mess up with the framework internals, thus vanishing the original efforts in guaranteeing the accessibility of its generated markup. In order to minimize this risk, UI state management (operations such as enabling a checkbox) is built-in into the framework, so that a developer does not need to manually change and/or alter the markup generated by the framework. Rather, the ability to provide callbacks for being notified and act upon significant events is provided at the framework level, so as to minimize unwanted side effects caused by their custom implementation. Whenever necessary to guarantee the accessibility of a certain component, handling of significant events (for example support of specific keyboard shortcuts) is built-in into those components.

Finally, in order to provide the accessible equivalent of a majority of components whose use is nowadays widespread in web development, a strong, possibly controversial principle has been adopted: *making an opinionated decision is better than not making a decision at all*. Application of this principle should be restricted to the minimum, so that the framework does not condition unduly the developer, yet its adoption

is critical in order to provide working components even in situations in which multiple solutions may be acceptable but would need a conscious implementation strategy. For instance, there are many different ways to implement accessible date pickers in HTML, each of which would require a different markup approach. We are choosing just ONE of such approaches to the detriment of all others that may have been preferred by some developers.

The Saharian browser's extension

> *And you may tell yourself*
> *This is not my beautiful page!*
> *And you may tell yourself*
> *This is not my beautiful style!*
> *And you may ask yourself*
> *Am I right? Am I wrong?*
> *And you may say yourself*
> *"My God! What have I done?"*
>
> — (with apologies to Talking Heads)

Even if there is some margin for possible improvements of automated accessibility testing, checking a page manually for accessibility issues is nowadays required. We believe that tools to help developers perceive accessible issues as they arise should be provided, so as to let them perceive their impact on people with disabilities. While negative effects of accessibility issues are often documented, in fact, we believe that mapping their effects to concepts that developers and content authors are more familiar with could make them more perceivable, thus help users recognize their gravity. It's our intention to develop a set of tools to implement this philosophy, the main of which is "Saharian".

Figure 1: The main interface of Saharian

The main interface of Saharian, allowing to activate, deactivate and switch between document and application modes.

Saharian is a browser extension (currently working on Chrome and Firefox) aimed at letting developers perceive the effects of ARIA annotations (roles, states and properties) used to enrich a certain web page, but in an innovative way. Unlike existing solutions (e.g. visualaria), Saharian does not limit its features to visualizing aria annotations and offering recommendations to implement the correct behaviours to support them in JavaScript, but rather uses the existing annotations and their supporting behaviours as implemented by the author to create a *visual* and *alternative* representation of the generated page.

For example, things like incomplete or inappropriate ARIA annotations will result in inappropriate

visualizations of the corresponding elements; incorrect keyboard support will be translated into incorrect behaviours of those elements for *mouse users* (for instance, if an element cannot be focused or activated via the keyboard, the user won't be able to focus or activate it by using the mouse), etc.. Saharian is the first in a series of tools that will be developed over time to complement our approach.

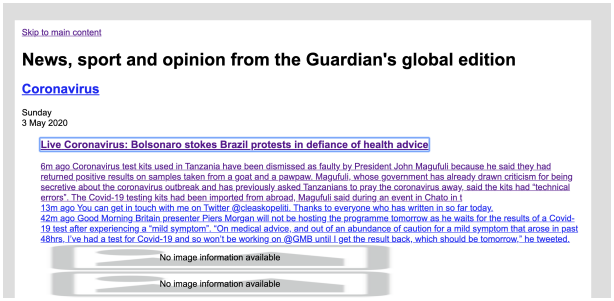
Figure 2: The usual visual display of a web page

A normal web page, as it is shown visually by a browser.

SAHARIAN (which stands for "*Sighted Architect's Helper for ARIA Notation*") performs the above-mentioned purpose by

1. deactivating the usual CSS and inner styling choices of the page and replacing them with default ones.
2. replacing all multimedia items with the default images with the alternative text in full sight
3. rerouting all interactive callbacks to mouse event handlers to corresponding keyboard ones
4. routinely verifying the update and modification of the DOM in order to capture and reorganize the new content in a similar fashion as the rest

Figure 3: The Saharian visual rendering of the same web page



The same web page, shown on the same browser, with visual styles replaced by Saharian.

As a result, the sighted developer is still able to interact and check visually with all the features of the application or content being developed. Yet, these interactions and visual checks are done on a page that is on purpose limited to only the visual styles and the behaviors that are allowed by the ARIA markup, and is as usable and comprehensible visually by the developer as much as it is usable and comprehensible in a non-visual way by a blind user.

In this perspective, a sighted developer can easily and rapidly verify the impact on the accessibility of the page of a well thought out design choice or a rushed last minute edit: by activating the SAHARIAN tool, developers are forced to rely only on the ARIA notation to make sense of the page and interact with its items, and, even if sighted, they will be able to carry out tests and activities on the page *only if* the ARIA

notation is correct and adequate.

Testing tools

Testing the end result of the design process is always a complicated process, and, as we know, there is never a *last* bug. Some considerations need to be made regarding the testing tools that are part of ou proposal so as to illustrate the significant role they play in our vison.

Testing in our approach is not performed through an homemade tool, but by extending and customi existing ones. We believe reinventing the wheel is not a good idea, especially when dealing with co topics such as web accessibility testing: if valuable and open-source solutions exist, it is better to base our work on them instead of building everything from scratch.

For this reason we make use of *axe-core* axecore, a rule-based automated accessibility testing engine. The fact that Axe is rule-based makes it easily extensible, as the only requirement for implementing new automated accessibility tests is to implement a few new rules, and flexible, as we can decide which tests should be run at any given time and how results should be presented. In addition, this tool is highly popular in the accessibility community, the company behind its development is authoritative and reliable and many professional solutions by tech giants like Google and Microsoft rely on it.

Let's consider a simple example. WCAG 2.1 states that a color contrast of at least 4.5:1 for small text or 3:1 for large text is appropriate, even when the text is part of an image, to ensure it is readable by users with low vision or color blindness (*success criterion 1.4.3*). Enforcing conformance to this rule within the framework, i.e. during the generation of the markup, would be easy, yet pointless: many factors will affect the color scheme of an element, including the loading of external resources (e.g., images), conditional styling, and even browser defaults. The perfect time to perform such checks is therefore not statically on the markup, but at runtime, after the page is loaded in the browser, all CSS styling has been applied and all external resources have been fully loaded. Static testing of the markup in the fully dynamic world of modern web design is basically futile.

The Axe library already provides this rule and many others out of the box, but it is the runtime of our framework that is responsible for running it at the most appropriate moment within the page/app lifecycle (e.g., after the loading process is completed, after new content is inserted, or existing content is deleted, or replaced, etc) and translate its results in a form suitable for our examination. In particular, the default test results from Axe include a selector pointing to the DOM element that failed it, but do not highlight it in any way: it is the responsibility of our framework to parse these results and show the violations reported using the same mode and styles of accessibility issues ascertained statically in the markup generation phase (e.g., highlighting the visual rendering of the component to facilitate the developer understanding the issue and how to act upon it). This offers a consistent experience for the developers, and maximizes the usability of our design tools.

Another important concern for our testing tools is the so called "*zero-false-positives principle*": if something is reported as an accessibility issue, it **must** be an accessibility issue. There are times when something that looks like an accessibility issue (for instance, an image with an empty alt text) is not actually an accessibility issue (the image is decorative), thus automated testing in this regard may bring uncertainty on the table. In order to be reliable and trusted by designers, our tools should never report false

positives. While automated testing on a bare HTML page is not able to discriminate such situations, our framework is in a good position to do so: in a truly "declarative fashion", decorative and content images are represented by different components, in order to have all the information necessary to disambiguate the situation at runtime. The *img* element is a good example of an HTML tag that is overloaded with many possible semantics, an evidence that the cooperation between a declarative framework and an autom runtime testing tool makes the overall result more reliable and less prone to ambiguity.

Eventually, whenever an accessibility error is detected (regardless of whether it was done by the components of the declarative framework during the markup generation phase, the automated testi or the manual accessibility visualizer), it should be reported prominently to the designer; the final goal of our system is to make it impossible for clear accessibility issues to end up unnoticed.

Finally, there is another aspect about accessibility testing tools that is worth discussing. More often than not, such tools report the line number of the source code that contain the error. We believe this is not the most effective representation to let non-disabled people perceive the impact of accessibility issues on people with disabilities; not only the source for runtime problems may be the result of several independent and apparently harmless bits of code spread in the HTML, CSS, Javascript and any of the various libraries being imported: the real need for a sighted designer is being informed of the impact on disabled users that the problem is causing in a manner that make these issues easier to grasp (such as the visual appearance of the page) without looking up additional documentation or external resources. This is the reason why components provided by our framework are designed so as to "visualize" accessibility issues to sighted developers: whenever a code fragment causes an accessibility issue, its visual rendering is altered to let the developer know what the issue is about and perceive its impact on people with disabilities. Once again, this is possible thanks to the abstraction provided by leveraging declarative markup to let the developer describe his/her intentions semantically.

Conclusions and future developments

After trying to explain the main problems that developers and content authors have to face in order to produce accessible content, as well as highlighting the most significant support resources and tools available to assist them in such a complex job, we have described our approach to improve the current situation and facilitate a more widespread creation of accessible content by means of a declarative framework built on top of the existing web technologies, automated accessibility testing and innovative tools to let developers perceive the impact of accessibility issues on people with disabilities in ways they can understand without reading any technical documentation.

We strongly believe that this approach has a great potential in facilitating a more widespread production of accessible web content, as it offers tools to alleviate some of the most significant difficulties that developers, designers and content authors have to face in order to do that with the tools available today. The tools we propose are being implemented with an iterative process to refine and improve them over time, gathering feedback from the community and taking it into account to maximize their impact.

But their development opens up many possibilities for further research and provides important questions which as of today are not as easy to answer as they should be. The main purpose of our declarative framework, for example, is to provide components commonly used when developing websites and applications that are accessible by default. But which components should be included in such a set to

consider it complete? What are the most commonly used components across web pages? Finding an answer to this question can maximize the impact of our framework, as development may be prioritized by the popularity of (i.e. how necessary are) certain components over the others.

We have described how we intend to offer a tight integration existing automated accessibility tools, so as to provide more accurate tests and facilitate their adoption to ceck for accessibility issues both the mar generated by our declarative framework and parts of a web page or application that are not impleme using it. Currently such tools have technical limitations that influence their efficacy, but could they improved to open up new horizons for automated accessibility testing? With artificial intelligence a natural language processing techniques, which are promisingly arising in recent years, we believe there is a potential for such improvements that deserves being explored.

Finally, Saharian might be the first of a new generation of manual accessibility testing tools, specifically designed to help developers perceive the impact of accessibility issues on people with disabilities in a more comprehensive way than simply visualizing and suggesting appropriate fixes for them. Similar tools could be provided in a more generalized form, that supports making accessibility perceivable not only for the WAI-ARIA specific annotations but for any HTML element and attribute that influences how a page is conveyed to assistive technology users.

Lastly, as our approach leverages the usage of a declarative framework on top of existing web technologies, intriguing opportunities arise by this choice; for instance, the framework could be exploited so as to allow easily creating accessible multi-modal applications for whichthe web is only one of the means to be accessed by. With the population aging phenomenon currently going on, the known difficulties of elderly people when it comes to dealing with modern technologies and the fact that many countries still lack access to fast internet connections, this might become a critical aspect to take care about in the future. We believe that by definition our approach can help with these topics as well, thus there's another win for our approach!

References

[axecore] DequeLabs. Axe-core: Accessibility engine for automated Web UI testing. Online available at https://github.com/dequelabs/axe-core. Last accessed April 15, 2020.

[axe] Deque Systems, INC. Axe: accessibility testing for development teams. Online available at https://www.deque.com/axe/. Last accessed April 10, 2020.

[achecker] Gay, G., and Li, C. Q. (2010, April). “AChecker: open, interactive, customizable, web accessibility checking.” In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)* (pp. 1-2). doi:https://doi.org/10.1145/1805986.1806019.

[lighthouse] Google INC. Lighthouse | Tools for Web Developers. Online available at https://developers.google.com/web/tools/lighthouse. Last accessed April 12, 2020.

[erdeclarativewebprogramming] Hanus, M., & Koschnicke, S. (2010, January). “An ER-based framework for declarative web programming.” In *International Symposium on Practical Aspects of Declarative Languages* (pp. 201-216). Springer, Berlin, Heidelberg. doi:https://doi.org/10.1017/S1471068412000385.

[personasaccessibilitytesting] Henka, A., & Zimmermann, G. (2014, June). “Persona based accessibility testing.” In *International Conference on Human-Computer Interaction* (pp. 226-231). Springer, Cham. doi:https://doi.org/10.1007/978-3-319-07854-0_40.

[echarts] Li, D., Mei, H., Shen, Y., Su, S., Zhang, W., Wang, J., ... & Chen, W. (2018). “ECharts: A declarative framework for rapid construction of web-based visualization.” *Visual Informatics*, 2(2), 136-146. doi:https://doi.org/10.1016/j.visinf.2018.04.011.

[applicationembedding] Lorenz, D. H., & Rosenan, B. (2017). “Application Embedding: A Language Approach to Declarative Web Programming.” arXiv preprint arXiv:1701.08119. doi:https://doi.org/10.22152/programming-journal.org/2017/1/2.

[tenon.io] Tenon. HomePage | tenon.io. Online available at https://tenon.io. Last accessed April 11,

[automatedaccessibilitytoolsbenchmark] Vigo, M., Brown, J., & Conway, V. (2013, May). “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* (pp. 1-10). doi:https://doi.org/10.1145/2461121.2461124.

[wave] WebAIM. Wave - Web Accessibility Evaluation Tool. Online available at https://wave.webaim.org. Last accessed April 11, 2020.

[visualaria] WhatSock. The Visual ARIA Bookmarklet. Online available at http://whatsock.com/training/matrices/visual-aria.htm. Last accessed April 10, 2020.

[declarativeweb20] Wilde, E. (2007, August). Declarative Web 2.0. In *2007 IEEE international conference on information reuse and integration* (pp. 612-617). IEEE. doi:https://doi.org/10.1109/IRI.2007.4296688.

[atag20] World Wide Web Consortium (W3C) (2015, September). Authoring Tools Accessibility Guidelines (ATAG 2.0) 2.0. W3C Recommendation 24 September 2015. Online available at https://www.w3.org/TR/ATAG20/.

[aria11] World Wide Web Consortium (W3C), 2017. Accessible Rich Internet Applications (WAI-ARIA) 1.1. W3C Recommendation 14 December 2017. Online available at https://www.w3.org/TR/wai-aria-1.1/.

[aria11authoringpractices] World Wide Web Consortium (W3C), 2019. WAI-ARIA Authoring Practices 1.1. W3C Working Group Note 14 August 2019. Online available at https://www.w3.org/TR/wai-aria-practices-1.1/.

[aria11implementation] World Wide Web Consortium (W3C), 2014. WAI-ARIA 1.1 User Agent Implementation Guide. W3C Editors' Draft 25 March 2014. Online available at https://www.w3.org/WAI/PF/aria-implementation-1.1/.

[implementingatag20] World Wide Web Consortium (W3C) (2015, September). Implementing ATAG 2.0: A guide to understanding and implementing Authoring Tool Accessibility Guidelines 2.0. W3C Working Group Note 24 September 2015. Online available at https://www.w3.org/TR/2015/NOTE-IMPLEMENTING-ATAG20-20150924/.

[wcag21techniques] World Wide Web Consortium (W3C). Techniques for WCAG 2.1. Online available at https://www.w3.org/WAI/WCAG21/Techniques/. Last accessed April 12, 2020.

[understandingwcag21] World Wide Web Consortium (W3C). Understanding WCAG 2.1. Online available at https://www.w3.org/WAI/WCAG21/Understanding/. Last accessed April 12, 2020.

[uaag20] World Wide Web Consortium (W3C), 2015. User Agent Accessibility Guidelines. W3C Working Group Note 15 December 2015. Online available at https://www.w3.org/TR/UAAG20/.

[wcag21] World Wide Web Consortium (W3C) (2018, June). Web Content Accessibility Guidelines (WCAG) 2.1. W3C Recommendation 05 June 2018. Online available at https://www.w3.org/TR/WCAG21/.

[sgml] ISO (1986). Introduction to Generalized Markup, Annex A of ISO 8879:1986 Information

processing — Text and office systems — Standard Generalized Markup Language (SGML). Online available at http://www.sgmlsource.com/history/AnnexA.htm.

Author's keywords for this paper: Accessibility; Declarative markup; HTML

Balisage Series on Markup Technologies