Tallinn University
MSc Human-Computer Interaction

Improving Web Accessibility by Integrating Continuous Evaluation to Development Workflow

Master's Thesis

Author: Mariann Tapfer
Supervisor(s): Mustafa Can Özdemir, MSc
Mari-Ell Mets, MSc

Tallinn 2023

Notes (this will be removed from final paper)

Table of Contents

Introduction

1 Research goal and methodology

1.1 Research goal

Linters and tests are used in software development to ensure that all newly added code follows the quality and best practice standards that the company or developer has defined and that these changes don't break anything. A similar quality gate could be set up for ensuring conformance with accessibility standards. This would act as a gatekeeper to ensure that if there are any detectable problems in the added code then it can not be deployed until these have been fixed.

In this thesis, I intend to test if automated accessibility tests can help improve overall accessibility and awareness about it. Most testing tools also provide links and instructions about the issue that was detected. I think this could help improve the knowledge of the developers who use these tools. They can learn about common accessibility problems and how to fix them.

Many companies have design systems to help maintain consistency both in what the end users experience and how code is written. This seems like a good place to start with establishing a basic level of accessibility. Making changes there would have the biggest impact. Most accessibility checkers are intended to be used for the whole page or website. I want to run these tests on the individual components in a component library.

There are limitations to testing isolated components, but how I see it is similar to unit tests and end-to-end tests in software development. Unit tests are used to ensure that a small specific part of the code - a unit works as intended. End-to-end tests are run when all these parts of code are combined to make up the end product and they should check if all these small parts also work well in combination with each other. These tests are imitating what would happen when end users are using the product.

I see the accessibility tests being run on isolated components as being similar to unit tests - each one making sure that that particular element has no issues. Then when the components are used to make up a whole page there should already be fewer issues that need to be dealt with and the next steps could be taken to ensure a high level of accessibility.

Many studies have been done to compare different accessibility testing tools (**Alsaeedi2020**; **Ismailova2022**; **Sane2021**; **Vigo2013**; **RybinKoob2022**; **Duran2017**). I will take a look at the results of these studies and truly out enough tools on my own to find one that would be usable for my purposes, but I am not intending to compare them methodically as a part of this work. I will rely on the comparisons that have been made by others in the past.

My research goal is to test an automated accessibility tool in Pipedrive's React component library - Convention UI React (CUI). This will probably pose some limitations on the tool I can use. The selected tool should work with our current tooling and not disrupt the current

development flow.

1.2 Research questions

RQ1: How good is the knowledge about accessibility standards, tools and best practices in the company currently?

RQ2: What kind of errors can be caught by running automated accessibility tests on a component library?

RQ3: To what extent can integrating automated testing into a component library's development pipeline help improve its compliance with WCAG?

RQ4: What are the biggest problems of integrating automated accessibility testing into a component libraries' development workflow?

1.3 Methodology

Write about methodology

- Literature review
- Survey
- Case study
- Statistical analysis of survey and case study results
- User-centered research? (this was suggested when I presented my proposal. Does it make sense now?)

2 Background

2.1 Web accessibility

Accessibility is about providing equal access to goods and services to everyone regardless of age, gender or disabilities. We all have different abilities, some run faster than others, some see more clearly and some might not be able to hear sounds. WHO estimates that 1.3 billion people – or about 16% of the global population – experience a significant disability and this number is growing (**WHO2022**). Disability is a part of being human and persons with disabilities are different from each other. We can also talk about temporary disability - this might be a mother who can only use one hand because she is holding a child in the other or someone who broke an arm and can't use it until it heals. The limited ability might be very similar to someone who has a permanent condition. Everyone who designs the physical and virtual environment around us should consider these different limitations.

Accessibility principles can be applied to various fields. For example, in architecture, it could be designing buildings in a way that can be accessed by people who can walk on their own as well as the ones who need to use a wheelchair. In digital products it is more often related to the senses we use to consume content. Everything could be equally accessible whether I need or want to consume the information using my vision or ability to hear.

The biggest strength of the web is how easily available it is to everyone. Tim Berners-Lee, W3C Director and inventor of the World Wide Web said: "The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect" (**WWWC1997**). The virtual world has the potential to be much more inclusive than the physical world.

Web accessibility considers auditory, cognitive, neurological, physical, speech and visual disabilities - everything that might affect someone's ability to access the Web and people using different devices, who have reduced abilities because of aging, temporary disabilities, situational limitations or are using limited or slow internet (**Henry2022**).

2.1.1 Standards

The standards for considering people with different abilities and making web content accessible for different ways of consuming are defined in Web Content Accessibility Guidelines (WCAG) (**Kirkpatrick2018**). The guidelines are developed by Web Accessibility Initiative (WAI) which is part of World Wide Web Consortium (W3C).

The first version of WCAG (1.0) was published in 1998 (**Vanderheiden2023**). The current version is WCAG 2.1 and the next version 2.2 is scheduled to be finalized in May 2023 (**Henry2023**). All WCAG versions are backward compatible, meaning that all requirements that were in 2.0 are included in 2.1 and some new requirements are added. Version 2.2 is

currently in the draft stage, but the planned changes include some changes to the current guidelines. One current requirement will be removed and another one will change from level AA to level A.

WAI gives an overview of the laws and policies around the world. 25 out of 40 listed on the page are based on WCAG 2.0 or WCAG 2.0 derivative (**Mueller2018**). Pipedrive operates in the US and European markets so the most notable and relevant ones for me are the accessibility standard in European Union EN 301 549 enforced by Web and Mobile Accessibility Directive (**MuellerEU2017**) and Section 508 of the US Rehabilitation Act of 1973, as amended in the United States (**MuellerUS2017**). WCAG 2.0 AA level conformance has been incorporated into the law in the US and the European standards' latest version follows WCAG 2.1 AA (**LevelAccess2021**). In this thesis, I will test against WCAG 2.1, because it is the latest published version of the most widely used standard.

What would be a better resource for this overview? This seems to be outdated because another resource said the EU standard is referencing WCAG 2.1

WCAG is intended for developers of web content, authoring tools or web accessibility evaluation tools and others who need a standard for accessibility. The 4 principles of accessibility addressed in the guidelines are:

1. Perceivable - information and user interface can't be invisible to all the user's senses,
2. Operable - user can interact with the interface,
3. Understandable - user should understand the information and how the user interface operates
4. Robust - content should be robust enough that it can be interpreted reliably by a wide variety of user agents.

Under each principle is a list of guidelines that address that principle (**AGWGWP2022**). Under each guideline, there are Success Criteria that describe specifically what conformance to it means. Each Success Criterion is a statement that will be either true or false for specific Web content.

2.1.2 Rules format

W3C Accessibility Guidelines Working Group has developed Accessibility Conformance Testing (ACT) Rules Format to provide developers of evaluation methodologies and testing tools a consistent interpretation of how to test for conformance with accessibility requirements like WCAG (**Fiers2019**). The format describes both manual and automated tests. The aim of this is to make accessibility tests transparent and results reproducible.

For example, accessibility testing tools check if the provided HTML meets the requirements

defined in WCAG. These requirements are different for each element, but they might be also combined and include more criteria. Each element needs a specific set of requirements to be checked.

ACT Rules include atomic rules that define an element to be tested for a single condition and composite rules that can combine multiple atomic rules to determine if a single test subject satisfies an accessibility requirement (**Fiers2019**). Each rule defines when it should be applied. For atomic rules, this could be an HTML tag name, computed role or distance between two elements for example and composite rules it is a union of the applicability of the atomic elements it combines.

2.2 Accessibility evaluation

Different evaluation methods are used to determine if a website is accessible. These include expert review, user testing, subjective assessment, screening techniques or barrier walkthrough. Each method has its pros and cons depending on the subject of the evaluation, available experts and other factors.

Expert review, also called, conformance, standards or guidelines review or manual inspection is most widely used (**Brajnik2008**). It means checking if a page satisfies a checklist of criteria. The results depend on the evaluator's opinions and depend on the chosen guidelines. Skillful evaluators are needed for this method to be effective.

Barrier walkthrough is a technique where the evaluator has to assess how seriously some predefined barriers impact the user to achieve their goal on the website (**Brajnik2008**). Accessibility barriers can be any condition that makes it difficult for people to achieve a goal. Which method is suitable depends on several factors including the availability of skilled auditors and resources.

Screening techniques consist of evaluating a website by using an interface in a way that some sensory, motor or cognitive capabilities of the user are artificially reduced (**Brajnik2008**). Subjective assessment is based on a panel of users exploring a website themselves and giving feedback on what worked and what did not. User testing means conducting usability tests with disabled people while adopting a think-aloud method to get feedback on their experience.

Evaluations with experts are very dependent on the knowledge and experience of the evaluators. Research into the effect of expertise on web accessibility evaluation conducted by **Brajnik2011** shows that when web pages are evaluated with non-experts we see a drop in reliability and validity (**Brajnik2011**). Even with experts, the result will vary a bit, but the results should even out when at least 3 experts are used. For the same level of reliability, at least 14 evaluators that are not considered experts in the field of web accessibility are needed.

2.2.1 Tools for accessibility testing

There are software programs and online services that help determine if web content meets accessibility guidelines. These tools may support various standards. They might be browser or authoring tool plugins, command line tools, code linters, open source application programming interfaces (API), desktop or mobile applications or online services (**AbouZahra2017**). Some tools are aimed at non-technical content creators and these are often built into the tools they use daily. Others are online services, where you can enter the URL of your website to be evaluated or services that regularly check your website and that is be hosted by the provider or in a company's internal network.

The results can be presented as a report in HTML, CVS, PDF, XML or other formats, as in-page feedback with temporary icons and markup or as a step-by-step evaluation where the user is prompted to assess the parts that can't be automated (**AbouZahra2017**). The tool might transform the page, show only text or take away all the colors, to help identify issues. The scope of what the tool evaluates at once might be a single page or a group of related pages. Some are capable of accessing password-protected content.

Browser extensions and online tools usually evaluate one page at a time. This might work well for small websites and one-time audits but can be quite ineffective to use as a continuous long-term solution. Command-line interface (CLI) tools need more setup, but can potentially be seamlessly integrated into the development workflow.

Authoring tool plugins could be plugins that check if the color contrast conforms with WCAG requirements in Figma or plugins for integrated development environments (IDE) that provide immediate feedback to a developer in their working environment when they miss something related to accessibility in their code.

Some tools do a very specific task or work inside a very certain tool and ones that try to tackle accessibility as thoroughly as possible. In most cases using many different tools in combination will give you the best result.

2.2.2 Automated accessibility testing

In this thesis, a computer program that can be set up to perform checks automatically with minimal manual effort will be called automated accessibility testing. It is important to differentiate these kinds of tools from the wide range of other tools describes in the previous chapter. This would leave out browser extensions because these would need to take extra steps to run the checks every time. Automated tests should be triggered each time a change has been made. The earlier in the development process it can be triggered the better.

Mostly these are CLI tools that can be integrated to run together with other tests. The Document Object Model (DOM) is the data representation of the objects that comprise the

structure and content of a document on the web (**MDN2023**). These programs scan the rendered DOM of a website against accessibility standards like WCAG to find violations. These errors are reported back with a reference to the code that produced the error.

Continuous integration (CI) is a software development practice where members of the team integrate their work frequently and each integration is verified with an automated build that includes tests to detect errors as quickly as possible (**Fowler2006**). This is believed to help develop high-quality software more rapidly. One of the practices in CI is making your code Self-Testing - adding a suite of automated tests that can check a large part of the code base for bugs. These tests need to be easy to trigger and indicate any failures.

This last principle could be applied well to accessibility evaluation. The practice would follow modern software development principles. Tests will not catch everything, but they will catch enough to make it worthwhile. After the initial setup, they should not need a lot of maintenance and will be run every time someone contributes to the codebase. This can act as a very effective gatekeeper for any potential accessibility issues.

According to the survey conducted by Level Access about the state of digital accessibility 67% of organizations that practice continuous integration also include accessibility tests (**LevelAccess**). This has gone up from 56% reported in 2021. Organizations that have an accessibility program that is in the early stages (2-6 years) or who have IAAP-certified personnel are more likely than average to have implemented a CI testing process that includes accessibility.

It is important to mention that these kinds of tests can detect only a part of all the possible violations. The UK Government Digital Service Accessibility team compared 13 automated tools' performance in **GAT2018** on a page that had 142 deliberately introduced accessibility issues and found that they found 13-40% of issues. **Abbott2021** compared the two most popular accessibility tools that can be used in CI using the same deliberately inaccessible site in **Abbott2021** and reported that axe-core caught 27% and pa11y 20% (**Abbott2021**). **Vigo2013** compared 6 different tools in **Vigo2013** and found that they found 23-50% of WCAG 2.0 Success Criteria (**GAT2018**; **Abbott2021**; **Vigo2013**).

The tools and WCAG standards have evolved during this time, but the common conclusion to most of the similar comparisons is still that most tools will be able to detect errors on 20-30% or WCAG Success Criteria.

Find citation

Axe-core accessibility testing engine promises to find on average 57% issues, which is significantly higher than other tools on the market(**Deque2023**). They claim that the current statistics are founded on an inaccurate definition of accessibility coverage - the percentage of individual WCAG Success Criteria. In reality, some types of issues are found much more

frequently and the issues found by automated tests form a higher percentage of all issues compared to those discovered by manual detection. They suggest that the coverage should be the percentage of all the issues found on a site. They conducted a study where they compared 2000 audit results from testing pages with axe-core and manual testing methodology. The average percentage of the number of issues detected by axe-core for each data set is 57.38%.

They also say that looking at the results with current, in their opinion, inaccurate, definition coverage would be in the range of 20-30% (**DequeSystems2021report**). I did not find any studies about other tools that would use the same method for calculating the coverage so for this paper we will still compare the tools according to the most widely used coverage calculation method.

There is still a limit to what can be detected no matter how we define the coverage. More testing will always be required to ensure that the content is fully accessible. The main strength of these kinds of tests is that they can be set up to run automatically, and they provide measurable results. This means it is a good way to monitor compliance with WCAG rules consistently without much extra effort. This can help avoid unwanted changes and show easy fixes in your code.

2.3 Component libraries

Component Driven Development (CDD) is a development methodology that anchors the build process around components (**Coleman2017**). In **Coleman2017 Coleman2017** called CDD the biggest trend in user interface (UI) development.

A component is a well-defined and independent piece of UI, like a button, checkbox or card (**Ella2019**). This approach correlates well with other similar widespread principles that promote modularity in software development like atomic design and micro-front-ends. Designing each component as a standalone unit improves maintenance, reusability, and testing, shortens the learning curve and makes development faster.

The key to success here is the separation of concerns and isolating a logical piece so that it can be worked without distractions. It promotes concentrating on details and refining the element as much as possible. These pieces can be combined into more complex components if needed and when combined they can make up views that become the whole site or web page.

It is not very straightforward to preview one single separated component and if you need to set up a test or live environment for the whole webpage it would defeat the purpose of dividing the big problem into manageable chunks. In development tools called component explorers are often used to mitigate that.

Component Explorer is a separate application that showcases the components in various states (**Coleman2017**). Working on a single component by manipulating the entire web page or

app to a certain state can be difficult without a tool like that. This allows developers to test a given component in all the states that have been defined in isolation and make it easy to build one component at a time. It also makes it easier to go through all the possible states in one component and promotes the reusability of these elements.

Bit, Storybook and Styleguidist are popular tools used in component library development (**Ella2019**). Bit lets you pack the bundled and encapsulated components and share them on the cloud where your team can visually explore them. Storybook supports multiple frameworks and provides a rapid component development and test environment. The environment also allows you to present and document your library for better reusability. Styleguidist is useful for documentation and demoing different components.

A collection of multiple components that are shared to be reused is called a component library (**Ramotion2022**). Design systems include values and principles along, with branding, guidelines and all the building blocks and design patterns to create a successful product or service. It governs the design process in the organization. A component library or UI kit, UI library, and UI component library are one of the building blocks of a design system.

Most big companies have a design system that includes a component library. Often these might be published with an open source code, making it possible to reuse and extend them. Some of the more known ones include Material-UI (**MUS**), Adobe Spectrum (**Adobe**) and Bootstrap (**Collings**). They help keep things consistent across multiple projects by providing small building blocks that can be used to create complex designs. It can be built with HTML or by using a framework like React or Vue.

Using a component library provides consistency and better quality. These components can be polished over time to provide the best user experience to end users when they are integrated into the final product. The effort that can be put into developing a component that will be used in multiple projects is bigger than what would be reasonable for a single use case.

3 Research

The research conducted for this thesis can be divided into 6 steps:

1. Researching automated testing tools
2. Pre-case study questionnaire
3. Setting up automated testing tools in our company's component library
4. Manual accessibility audit
5. Comparing manual and automated testing results
6. Post-case study questionnaire

The first thing was to look at different automated accessibility tools that are available, test them out and see what would fit our purpose the best. The tool needed to be easy to use for everyone and not block development. The next step was to send out a questionnaire to better understand the current approaches toward accessibility among the people who work on it the most. After that, an automated accessibility tool was integrated and announced in relevant channels to raise awareness about it.

In parallel to observing how the tool is being used we performed a manual accessibility audit in the same component library. These results were then compared with the automated testing results that we got from the same version of the library. As the last step, I sent out a questionnaire to gather information about how the tool has been adopted. To get more details I also did some interviews with developers and designers I knew had used the tool during that period.

Pipedrive has been developing a sales CRM using mostly typescript and React. Accessibility has never been a high priority and at this point, it is not very easy to get started. We have a design system and a React-based component library to keep the look and UX consistent. This seems like a good place to start with solving accessibility issues. The library is used widely in the company and developers from different teams contribute to it. If a button in the reusable library gets fixed most of the buttons in the web app that our customers use should be improved.

This should not be taken as a way to solve all accessibility problems but as a good first step to getting started. Making changes in a reusable UI library should have a wide impact on the product's overall accessibility and without reaching some basic level there first it could be hard to start testing views in the final product.

3.1 Current state of awareness about accessibility in Pipedrive

First I sent out a survey in our company Slack channels (see figure 1) to understand what is the knowledge and general approach to web accessibility in the company. It was shared in 4 channels to reach people who are most likely to be dealing with accessibility, our component library and who are invested in accessibility (see more details in table 1).

Channel members or theme	number of members
Front end developers	212
Dedicated to our component library	124
Designers	73
Accessibility channel	31

Table 1: List of all the channels the survey was shared in.

Some people might also be on more than one of these channels. The aim was to reach people in the company who would be most likely to be using these tools and who would be likely contributors to the library. There were 7 questions and some of them also included a field for free text to give more details on the subject if they wanted.

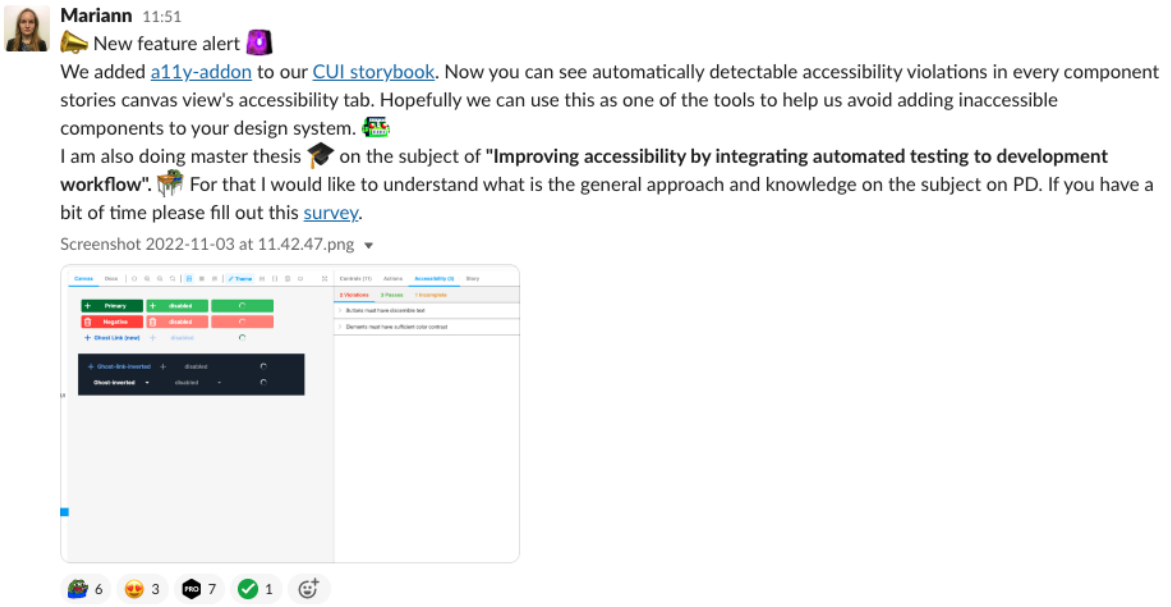

Figure 1: Message in company Slack announcing adding accessibility tool and inviting people to reply to the survey.

In total 20 people replied to the survey - 6 designers and 14 developers, including one engineering manager. This does not give a full overview of the company, but it should give a good insight into what general opinions regarding accessibility might be. Likely, developers and designers that are more involved with our component library and/or interested in accessibility were more likely to respond.

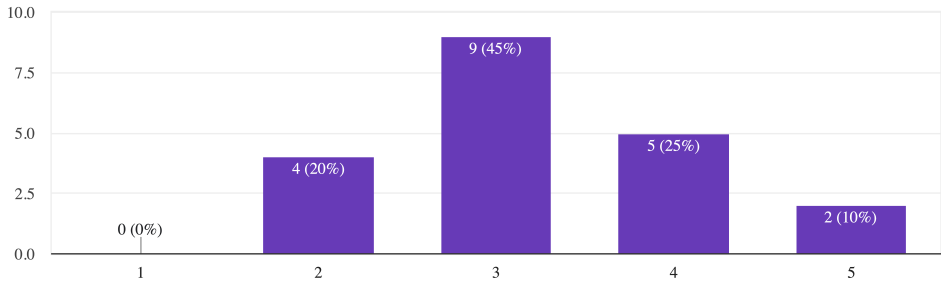

Figure 2: Results to pre-case study survey question: How much do you know about accessibility (standards, best practices, importance, testing)? 1- "Almost no knowledge" to 5 - "I have very good knowledge"

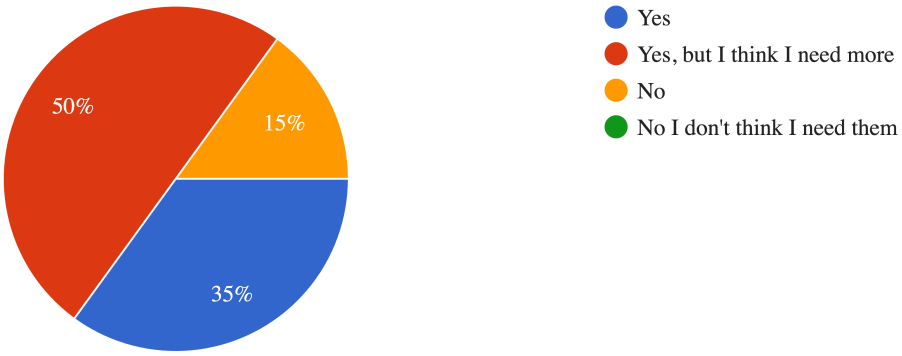

Figure 3: Results to pre-case study survey question: Do you know where to find resources about accessibility standards?

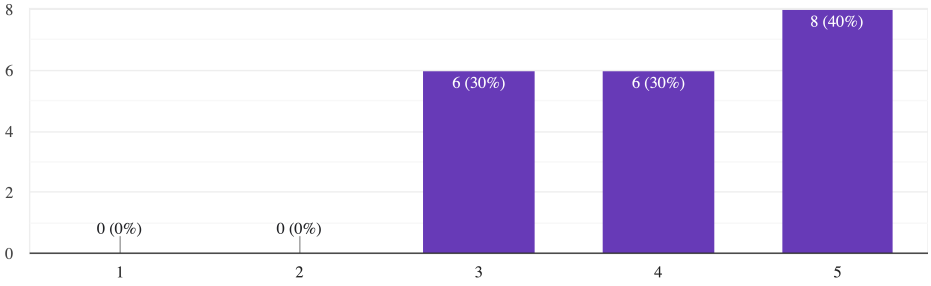

Figure 4: Results to pre-case study survey question: Do you think we should prioritize following accessibility standards (like WCAG, EN 301 549) in our product? 1- "No I think they are irrelevant for Pipedrive customers" to 5 - "Yes I think following accessibility standards should be a high priority"

The results show that 10% of people who responded think their knowledge of accessibility is very good, while most think that their knowledge level is average (see figure 2). 35% of responders know where to find resources about accessibility standards, 15% don't know and 50% know, but think they need more (see figure 3). They don't see that accessibility as a high

priority in the company currently, but at the same time, 40% of the people who responded think that following accessibility standards should be prioritized (see figure 4). There were several comments in the free text sections expressing saying they appreciate this subject being opened.

3.2 Adding automated accessibility tests to component library

The next step was adding some automated tests to our component library development workflow and observing their usefulness. The aim was to find something that we can integrate easily and that does not have a steep learning curve or add unnecessary complexity.

We use Storybook as the component explorer for our reusable component libraries development. It is an open-source software for UI development tool that allows you to work on one component at a time (**storybook**). It allows us to render isolated UI components without integrating them into the final product right away. Our developers use it to test out the components they are developing locally. We also have a version of Storybook available for anyone in the company to see with all the components. If this is the first place where elements will be rendered it seems logical to try to find a way to start testing them there.

To show a component you need to write a story - this means use it the way you would use it in the real world, and it will be rendered in a browser inside Storybook UI. It also has a sidebar for navigating between different examples, and controls for additional tools and documentation. This means we can't use a browser extension for accessibility tests because that would also test the UI around the actual relevant example.

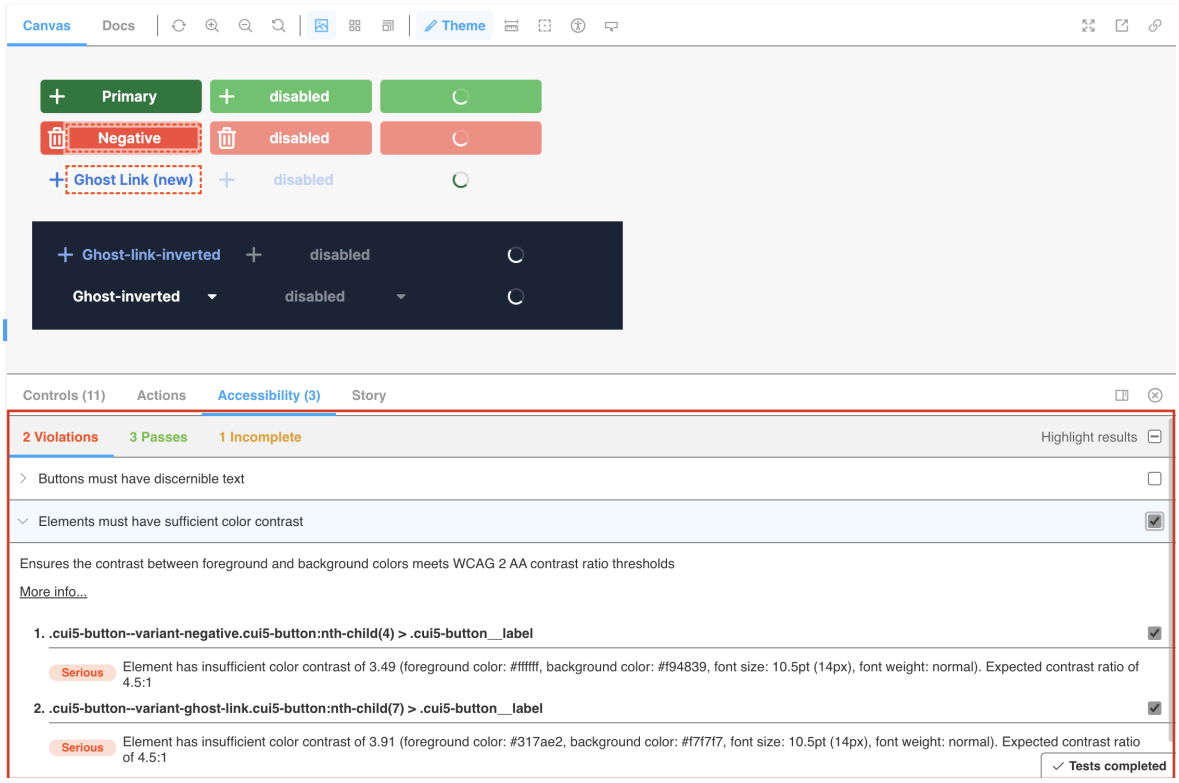

Figure 5: Accessibility add-on for Storybook (addon-a11y)

Storybook has a wide ecosystem of add-ons and one of them is addon-a11y (see figure 5). It uses Deque's axe-core to audit the HTML rendered for an isolated component (**addon-a11y**).

Axe-core is an accessibility testing engine for websites and other HTML-based user interfaces (**Deque2023**). It includes WCAG 2.0 and 2.1 on levels A and AA and it has a coverage of about 20-30% if you calculate it in a commonly used way, but they claim it to be 57% according to their new and improved calculation method. This has been explained in more detail in the Automated accessibility testing chapter.

This seemed like the best solution in our situation so addon-a11y was added to our component library's storybook. The tool is visible in the sidebar of every example. It shows all the checks that it has passed, all the violations that were found and any issues that could not be checked and might need manual testing. Every rule listed there also has reference to the HTML node that had the violation, explanation and links to the Deque webpage with examples. This should be very useful in understanding and fixing these issues.

It does not generate a report of all the issues found across the whole library for this another tool was used that will go through all the examples and generate a summary of all the violations found.

The rules format that axe-core uses is developed by Deque Systems and is an adoption of the ACT rules format developed by WAI (**Fiers2017**). They have a set WCAG that can be

evaluated in a fully automated way. They are divided by WCAG standards version (2.0, 2.1, 2.2) and Level (A & AA, AAA) and they also have some rules for best practices in the industry that improve the user experience but might not conform to WCAG success criterion (**Fiers2023**).

I used another tool to generate a report of all the violations in the whole component library. It goes through all the examples and runs the same tests as addon-a11y, but it combines them in one document. The final report links back to each component story where you can see the example and addon panel with all the information I mentioned before.

To get some data that could be compared with the results from manual testing I went through all the examples and extracted the following info for each component:

- How many occurrences of the component are there in the accessibility violations report? Might contain the same issue multiple times.
- How many unique issues does the component have? Most components have more than one story. Opened the example and went through all the violations in all the stories and counted unique ones.
- How many passed checks does the component have? This together with violations will show how many things were tested for each component. Same method as in the previous item - all unique passed checks in all the stories for that component.
- How many of these checks are valid? Looked at the example and only counted passed issues that are valid and relevant to the component that the example is about.

3.3 Manual accessibility audit

The second part is conducting a manual accessibility audit in the same library to get a better overview of all the issues. This will also allow me to compare the results with the automatically generated test report results to determine what are its strengths and weaknesses and if testing isolated components pose any limitations.

3.3.1 Methodology for manual audit

We used Storybook to render an isolated preview of each component. The accessibility add-on had already been installed, and we looked at the violations reported there too. We worked in a team of 4 people - 3 designers and 1 developer. We created a task for each component - 53 tasks in total.

Everyone worked individually and each morning we had a quick meeting to discuss any questionable issues. Each evaluator checked violations in the accessibility add-on panel, tried using only a keyboard to navigate, and tried using a screen reader to navigate. Each

component had 1 or more examples. As many as was relevant to get the whole picture where looked at.

At the beginning of the audit, we tested an add-on in Storybook for mocking a screen reader(**Lara**). It had the option to show the output a normal screen reader would play as audio as text. Initially, it seemed like a convenient solution with rather reliable results, but further investigation revealed that the output was very different from what an actual screen reader.

For the rest of the audit, we used VoiceOver - the MacBook built-in screen reader because our work computers are Mackbooks and it was easily available to us. There was an initial learning curve, but after that, it went quite smoothly. All the components that had already been tested with the faulty add-on were looked over again using VoiceOver.

The audit results were documented in a table. We used the barrier walkthrough method where the barriers we focused on were defined by how the user interacts with the webpage and looked at what issues different types of users might encounter. We separated them into 3 sections:

1. Mouse user issues - using a mouse to interact with the page
2. Keyboard user issues - the user can only use a keyboard to do the same things that you could do using a mouse
3. Screen reader user issues - trying to do the same things and getting the same information while only using a screenreader

We see this separation as a good way to prioritize fixing the issues in the future. The mouse user is the user we are considering in all of our development currently. The issues they would encounter should be the most critical. This category includes a lot of visual, color contrast and image text issues.

The second type of user would encounter all the issues from the first category plus everything that is unreachable to them by using a keyboard. We looked at what functionality is or should be working when you use a mouse and tried to do the same things by only using a keyboard.

The third user was imitated by using a screen reader. The prerequisite for this was keyboard usability - if it was not possible to navigate using a keyboard then most likely it would not be very usable by a screen reader.

In real life, these users might not be so clearly separated and many issues would affect all types of users, but as the intention was to come out of the audit with an actionable list we needed to prioritize the issues while we found them in order of severity and current customer impact. These categories also mostly depend on each other, so it would make sense in most cases to start by solving mouse user issues, then keyboard user issues and then screen reader user issues.

4 Results

4.1 Comparing results from manual audit and automated report

To get the whole list including all components a report was created that included the violations caught in each example of each component. This report was generated at the beginning of the manual audit, so the results obtained from both methods are based on the same source code.

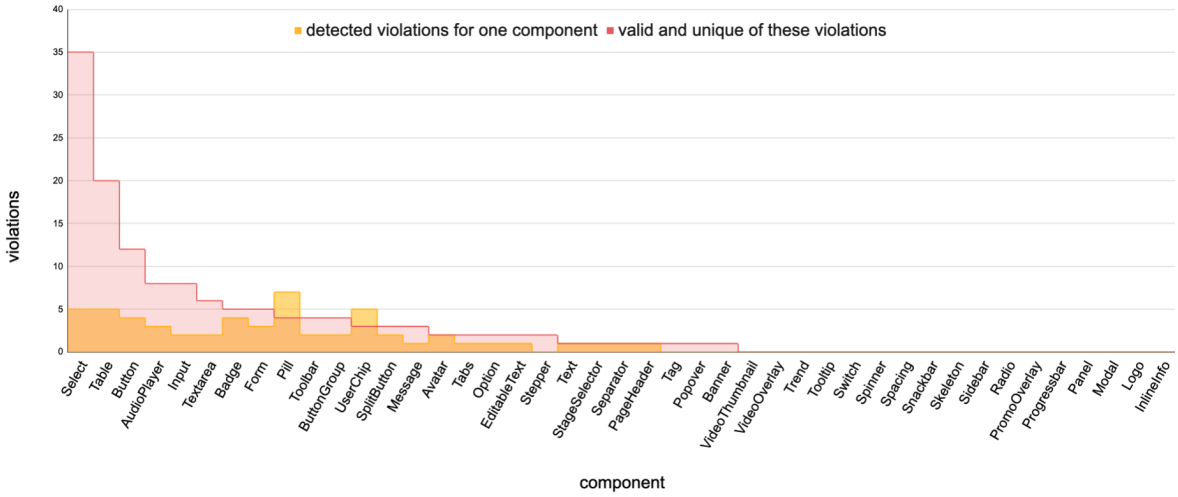

Figure 6: All violations found by addon-a11y and how many of them are valid.

I prepared a comparison table from both results. For automated accessibility tests, I recorded the number of examples that included violations, the number of different violations and the number of passed checks for each component. In most cases, there were more examples with violations because the same thing was reported in more than one example (see figure 6). Addon-a11y did not report any issues for 27 components out of 53 components.

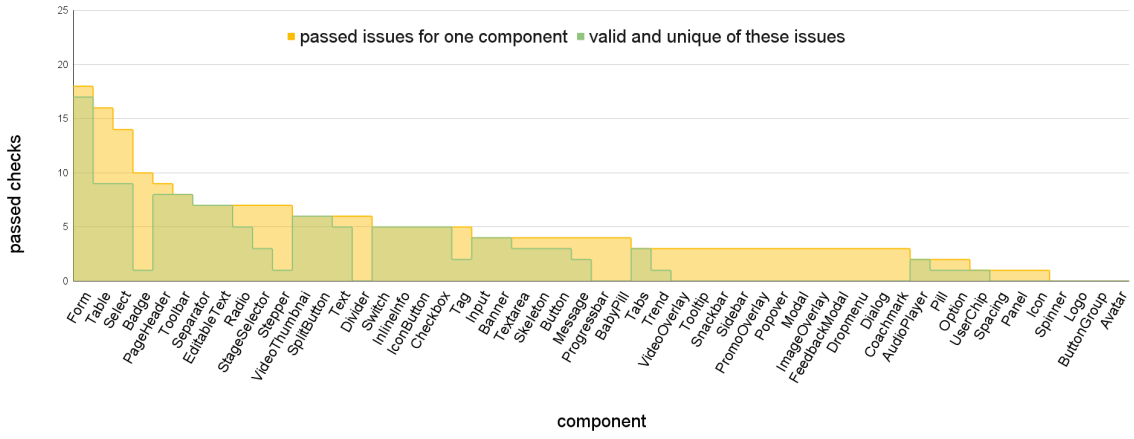

Figure 7: All passed checks reported by addon-a11y and how many of them are valid.

Passed issues were looked over to determine how many were valid (see figure 7). 4 components

did not have any passed checks and 22 did not have any valid passed issues.

Looking at all the fails and passes gives an overview of what was checked for each component. Out of 53 components, only 2 did not get checked by addon-a11y at all. In addition, components that become visible only when triggered by another element, like modals and popups that are currently in our library displayed with a button as a trigger, don't get tested. These components are seen in figure 7 starting from *VideoOverlay* and ending with *Coachmark* - 27 overall. This means 29 components were not tested but this tool and the rest of the components had passed or failed issues, but often not both.

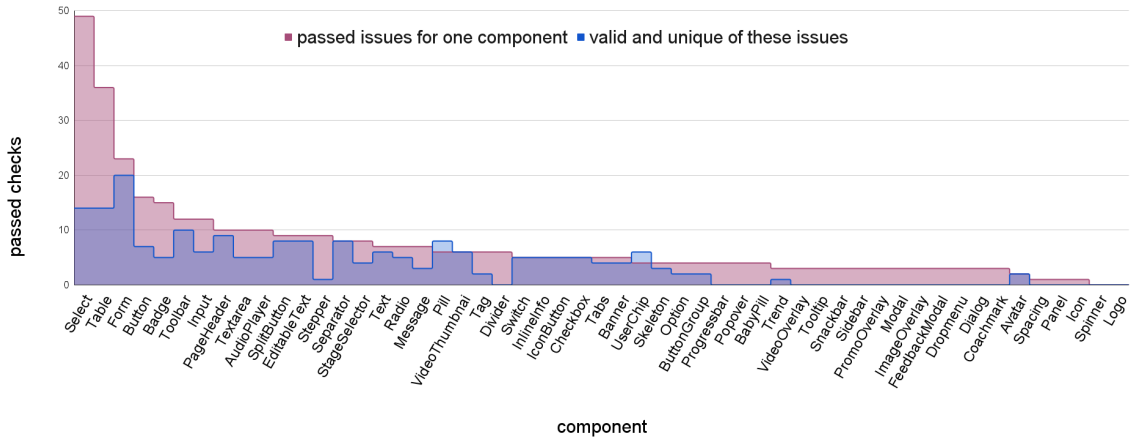

Figure 8: All issues that were tested by addon-a11y. This means both passed and failed issues combined.

In some cases, there were 0 violations detected, and 0 valid checks passed – this means that the automated testing was not effective (

Make chart or table for this

).

5 Discussion

5.1 Limitations of the study

5.1.1 Limitations Storybooks addon-a11y

The accessibility add-on in Storybook analyzes the examples that have been made for the component and unsuitable examples can cause false results. Like Components triggered by a button described before. This is because the initial HTML that the accessibility tests are being run on only has the button and the tests are not being run again after triggering the element. This could potentially be remedied with better examples.

The biggest limitation of this tool currently is that it can only be viewed in Storybook. To see the number of passes and fails you need to open the accessibility tab for each component. I looked into ways of automating this so that the same checks could be run on every change to the library and added to the continuous integration (CI) workflow. In the current version of Storybook, there is no easy way to do this, but it will become much easier in the next major version.

Upgrading our component library to that version would need some extra work to make it compatible, but I have tested out this solution on a test library, and it seems like it would be an improvement. Running the tests in CI would ensure that they are run every time someone makes a change and not only when we choose to. We could also block changes that don't pass the required accessibility checks.

Reasons for automated check not being effective

5.2 Future research

1. Testing storybook 7 and its test runner.
2. Components that only show a button - play function + test-runner? Would it work?
3. Adding accessibility tests as a part of library setup - would that have a bigger impact

Conclusion

Appendices

List of Figures

List of Tables