DIGDIR Exercise 1: UU-extension

Brage Asperanden Navarsete
Institutt for informatikk
Kristiania
Bergen, Norway
Brage.Asperanden.Navarsete@digdir.no

Eiril Solveig Ugulen
Institutt for datateknologi og informatikk (IDI)
NTNU
Trondheim, Norway
Eiril.Solveig.Ugulen@digdir.no

Audun Kristian Oklevik
Institutt for informatikk
Universitetet i Bergen
Bergen, Norway
Audun.Kristian.Oklevik@digdir.no

Andreas Conradi Nitschke
Institutt for Ingeniør- og teknologiutdanning
Høgskulen på Vestlandet
Bergen, Norway
Andreas.conradi.nitschke@digdir.no

Kristoffer Svedal
Institutt for datateknologi og informatikk (IDI)
NTNU
Trondheim, Norway
kristoffer.svedal@digdir.no

Thea Ueland
Institutt for informatikk (IFI)
Norges arktiske universitet (UiT)
Tromsø, Norway
tue001@uit.no

Kae Saito
Institutt for Teknologi og Innovasjon
Høyskolen Kristiania
Oslo, Norway
kae.saito@digdir.no

Kristian Gullhaug Birkeli
Institutt for data- og elektroteknologi
Universitetet i Stavanger
Stavanger, Norway
kristian.birkeli@digdir.no

***Abstract*—A short summary of the results from the project**

***Index Terms*—important keywords**

I. Introduction

A. Task description

This task is provided by UU-tilsynet who is working with universal design. Characteristics of any universal design (UD) product or environment is that it is accessible, usable, and inclusive.

The purpose of this project is to create a tool for semi-automatic testing which UU-tilsynet can use to automate some of the testing they are currently doing manually.

B. What should be tested?

Create a chrome extension responsible for testing specific WCAG requirements. The current solution is a tool for semi-automatic testing of buttons, to ensure that the buttons are compliant with universal design.

C. Concepts/ technologies used in this solution

1) Universal design:
2) WCAG:
3) chrome extension:

II. Related work

A. axe DevTools - Web Accessibility testing

III. UU-extension: architecture

A. Functionality

When you click on our extension, a side panel opens with buttons for dark mode and scanning the webpage (Fig. 1). Clicking ”Scan Page” analyzes the webpage for buttons,

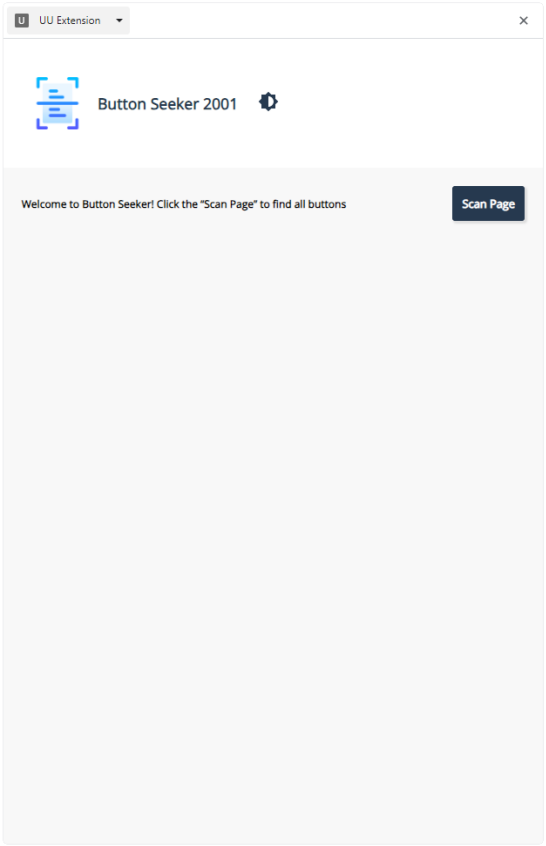


Fig. 1. Extension front page

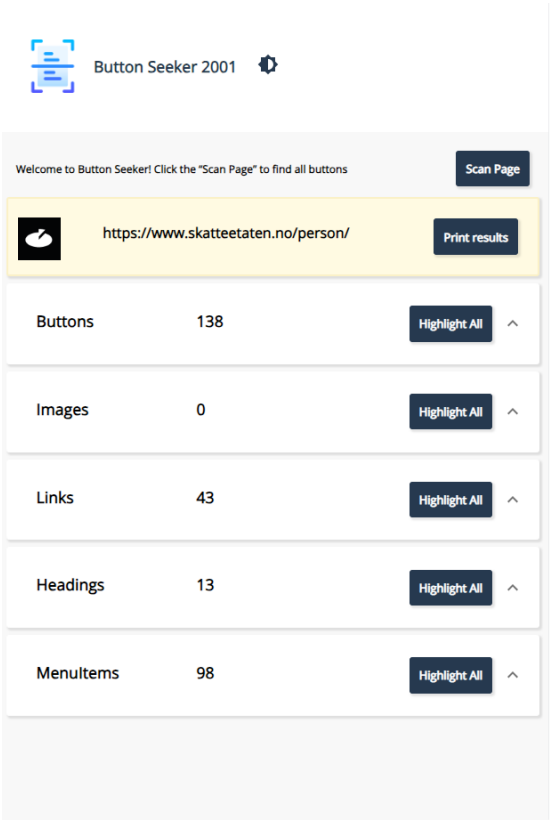


Fig. 2. After clicking "Scan Page"

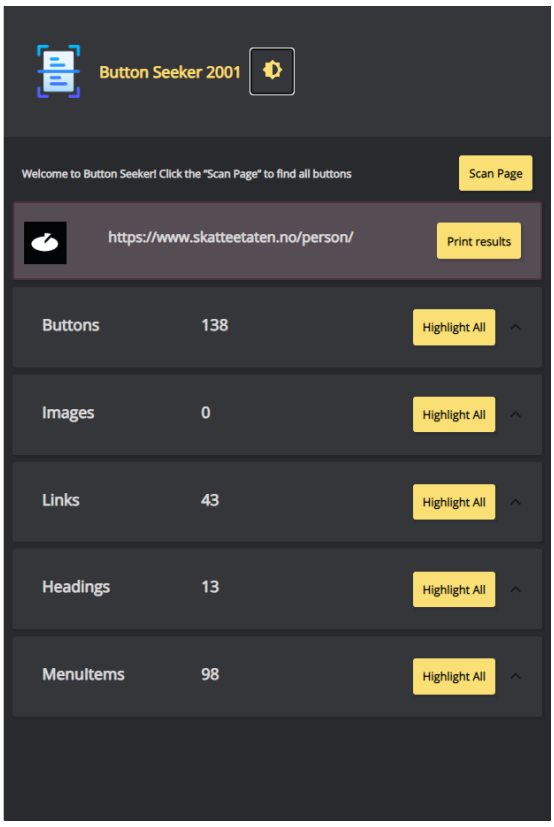


Fig. 3. Darkmode

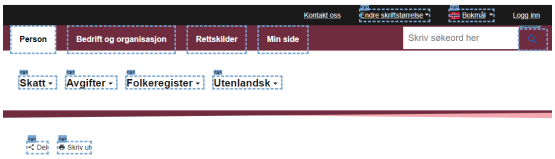


Fig. 4. Dashed border around elements on the page after clicking "Buttons" (skatteetaten.no)

Fig. 5. Red border after clicking "Highlight All" on the "Buttons" category (skatteetaten.no)

images, links, headings, and menu items and also shows the webpage logo (Fig. 2 & 3).

After scanning the page, all elements in the categories listed above are highlighted with a dashed border (Fig. 4). You can choose to highlight all elements within each category or expand specific categories for a more detailed inspection. When clicking "highlight all" next to a category, all elements in that category are highlighted in red instead of with a dashed line (Fig. 5). When clicking "highlight all" a second time, the red border is replaced by the dashed border again.

If you want to highlight specific elements, you can expand each category further. For instance, expanding "Buttons" displays a list of buttons found on the page (Fig. 6). Clicking on a specific button will highlight only that element in red, and also show a table presenting attributes such as aria-labelledby, aria-label, title, Description, Role, and Focusable, along with their corresponding values when available (Fig. 7 & 8).

You can then evaluate each button to assess its adherence to universal design principles and user-friendliness. By clicking "Yes", "No", or "Not a button", you answer the question: "Does the button's prompt identify its function?" (Fig. 9). In addition, you can add a comment to provide more information if necessary. The comment is automatically saved, and the user is alerted with a popup. By clicking "Print Results", the results are shown in the console log in JSON format (Fig. 10) and also stored in a database for analysis and future improvements.

B. Backend

This application also has a restful server processing incoming HTTP requests from the chrome extension. This server is responsible for storing the results from a semi-automatic test completed in the extension, and sent to the backend as JSON data.

The extension is also using this backend to fetch the data which is being displayed in the extension's sidebar to the user conducting the semi-automatic test. This functionality is not fully completed yet, but it is intended to replace our current approach where we are getting this data manually from the

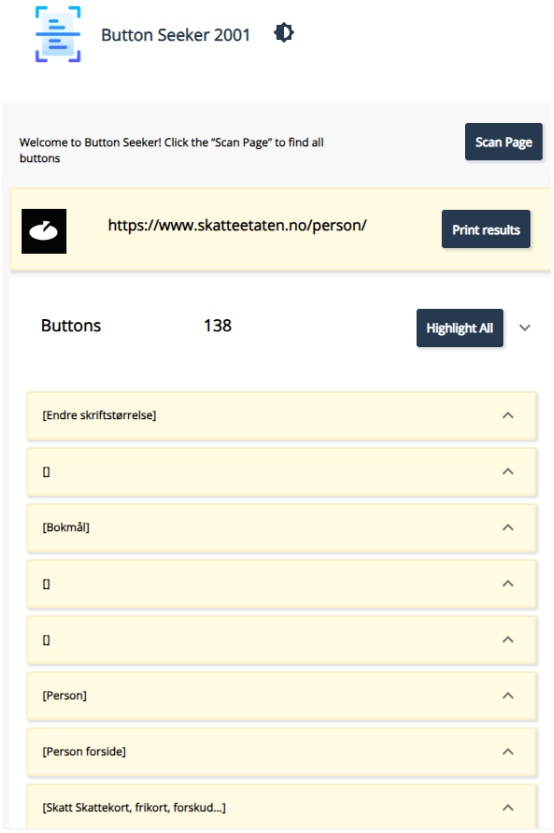


Fig. 6. After expanding the ”Buttons” category

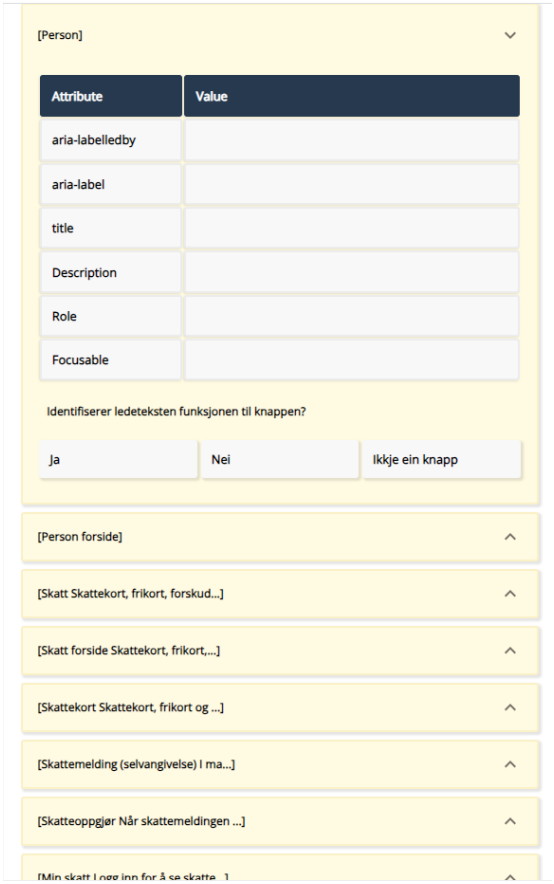


Fig. 7. After expanding the button ”Person”

html data.

The backend is divided into the modules server and WCAG. WCAG is then divided into the modules accessibility and database, as shown in the figure.

1) The server: is currently only serving at localhost on port 8080, unless another port is specified as a process environment variable. The server is set up to process the following HTTP requests:

- GET 127.0.0.1:8080/buttons
  - Getting all test results for buttons
  - Not implemented yet, currently only replying with a message
- POST 127.0.0.1:8080/storage/saveButtons
  - Saving the test result in a database
  - The data should be sent in the request body as JSON data
  - The request handler for this request will validate the JSON data before trying to save the data
- GET 127.0.0.1:8080/storage/deleteAllButtons
  - Deleting all rows in the database table used for saving the test results for buttons
  - This should not be in deployment of the application, but it has been useful while developing when there is no important data in the database that should not be deleted

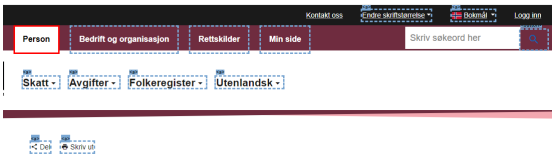


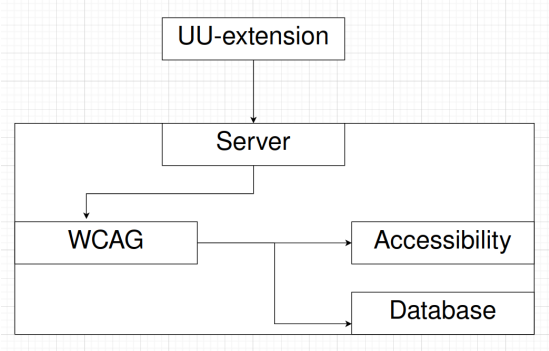
Fig. 8. Red border around the “Person” button on the page after clicking it in the extension (skatteetaten.no)

2) WCAG: The module WCAG is responsible for everything related to the semi-automatic test happening in the backend. While the server module is responsible for handling all incoming requests, and sending a response back to the client. This module is responsible for all of the interaction with the database and getting the computed properties from the accessibility tree, by querying a specific website. This functionality is being implemented in the accessibility module in the backend, because it proved to be challenging to get it to work in the frontend.

Fig. 9. Evaluating the button and adding a comment

Fig. 10. Inspecting the extension and opening the console log after clicking "Print Result"

Fig. 11. Chrome extension and backend



IV. Implementation

A. Technology

In our project implementation, we utilized a range of technologies to ensure its success and efficiency:

1) TypeScript (TS) - Our primary language for the codebase due to strict type declarations, providing better code control and error detection. TypeScript was translated to JavaScript automatically during the build process for compatibility with the Chrome extension.
2) JavaScript (JS) - Used to handle specific extension functionalities, such as side panel behavior and Chrome API interactions. Also responsible for creating extension update notifications.
3) Cascading Style Sheets (CSS) - Played a vital role in styling and design, offering colors, shapes, spacing, and animations to create a visually appealing and user-friendly interface.
4) Hypertext Markup Language (HTML) - Structured the user interface, with the sidebar.html file defining the main extension interface and incorporating JavaScript files for functionality.
5) React - Employed as a JavaScript library to create a modular and component-based UI. React's virtual DOM ensured smoother updates, improved responsiveness, and enhanced code reusability.
6) Webpack - A powerful build tool that optimized source code, assets, and dependencies for deployment. It transpiled TypeScript and bundled CSS, contributing to a streamlined development and deployment process.

These technologies collectively contributed to the project's success by providing better code control, enhanced user experience, efficient development, and reliable testing practices.

B. Integration

As our project is a Chrome extension, we utilize the Chrome API, a set of interfaces provided by the Chrome browser. This API allows us to interact with browser functionalities, such as manipulating tabs, accessing web page content, and creating notifications. By leveraging the Chrome API, we could build custom functionality and integrate our extension seamlessly into the user's browsing experience.

C. Deployment

D. Version Control

For management of our project's codebase and collaboration among team members, we adopted the version control system, Git, along with the collaborative platform, GitHub.

Within the GitHub repository, we utilized the "Issues" feature to create, assign, and track tasks related to bug fixes, feature implementations, research activities and more. We used labels to categorize issues based on their nature, such as "bug," "enhancement," or "research." Issues were linked to projects and milestones, aligning with different sprints for clear planning.

For each issue, we adhered to a structured workflow using branches. When an issue was assigned to a team member, GitHub automatically generated a new branch with a descriptive name corresponding to the issue. This approach allowed developers to work independently on their tasks without interfering with the main codebase.

Once a developer completed their work, they opened a pull request on GitHub. Before merging the code into the main branch, the pull request required approval from at least one team member. This practice ensured that all code changes

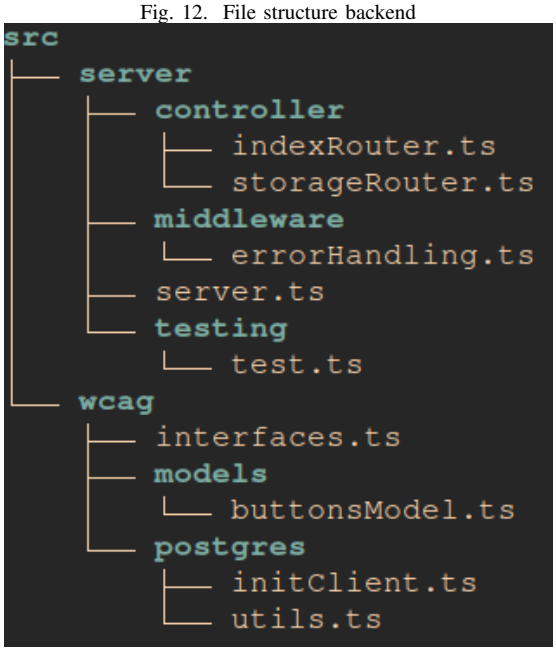
underwent a peer review process, enhancing code quality and maintaining project integrity.

In conclusion, Git and GitHub helped us manage version control efficiently. GitHub also helped our collaboration, with possibilities for discussing issues, sharing ideas and providing feedback.

E. Backend

The file structure in the backend is as shown in the figure, implementing the modules server and WCAG as explained above. The main file is '/src/server/server.ts', and this file is setting up an express server using routes and request handlers defined in the '/src/server/controller' directory. The file '/src/server/middleware/errorHandling.ts' is implementing an error_responder middleware function used in error handling. Currently the server is sending the error messages to the client, in case of failures during the processing of a request, which from a security perspective is not ideal. This information is useful while developing, but it should be logged to a file instead, and sending more general error response messages back to the client. The file '/src/server/testing/test.ts' is being used to implement end-to-end testing of the server, by sending HTTP requests to the available endpoint, and verifying that the response is as expected.

Fig. 12. File structure backend



V. Evaluation

A. Methods/ Experiments

1) Unit Testing: To ensure the reliability of our codebase, we utilized the Jest, a JavaScript testing framework, to conduct unit testing for approximately 60% of our code. While not all code was covered due to development priorities, we focused on testing critical and complex components to catch bugs and prevent regressions. These tests played a significant role in validating functionality and providing confidence in the stability of key modules. While aiming for 100% test coverage is ideal, our pragmatic approach prioritized essential testing for a functional and robust application. Please refer to Fig 14 for details on which files have or have not been completed.

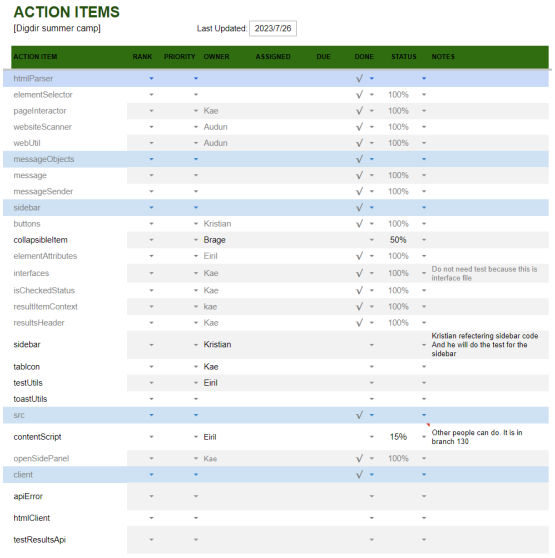


Fig. 13. Unit test progress management table

2) Manual End-to-End Testing: In our development process, we relied on manual end-to-end testing to verify new implementations. Whenever we worked on a new feature, we built and tested the Chrome extension to ensure correctness and functionality. This approach allowed us to catch potential issues early on and bugs noticed while using the system were corrected and addressed.

B. Results

Development of the project was actively pursued until August 2, 2023, with the final unit test code being composed on July 26, 2023. The test results (Fig. 12) obtained as of July 31 were derived from the test code corresponding to the latest development version up to that date. It is worth noting that due to the recent refactoring of the file structure and code, the results may not be entirely precise. While comprehensive automatic testing was not executed, the system has undergone extensive real-world usage, revealing no significant issues during its operation. Remarkably, despite the absence of comprehensive testing, the system has consistently demonstrated the desired performance even after the integration of new minor implementations following the completion of unit tests.

VI. Future work

A. Analyzing Elements other than Buttons

During the course of our project, our primary focus has been on implementing the Buttons functionality within our extension. It's important to note that functionality for analyzing Images, Links, Headings, and Menu Items on web pages are yet to be fully integrated. The future implementation

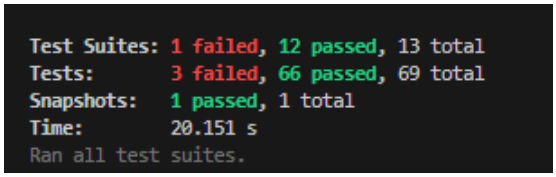


Fig. 14. Unit tests results as of July 31

of these additional features would enhance our extension's capabilities to comprehensively evaluate the user-friendliness and adherence to WCAG guidelines of various websites.

B. Computed Properties

In future development, the UU-extension can be further enhanced by utilizing computed properties. Although our project concluded before complete implementation, the progress made on a development branch lays the groundwork for future developers to integrate computed properties effectively.

By leveraging computed properties, the extension can access existing data more efficiently, eliminating the need for redundant custom solutions. This advancement opens up possibilities for analyzing other elements such as images etc. in perhaps a better way than if one would continue with our original approach.

VII. Challenges and Solutions

1) Misunderstanding of the Task: Early on, we faced a critical issue when we misunderstood the requirements of the task related to analyzing buttons and other elements. Our initial approach was to make the process entirely automatic, whereas the product owners at UU Tilsynet desired a semi-automatic process, allowing them to evaluate and label the buttons themselves. This misalignment was partially due to the lack of specific questions from our end and limited understanding of the task. Fortunately, after clarifying the requirements with UU Tilsynet in a subsequent meeting, we quickly corrected our direction and aligned our efforts accordingly.

2) The Use of Computed Properties: Towards the project's conclusion, we received valuable feedback from UU Tilsynet, which led to an important discovery. We found that the information they needed could be obtained from the computed properties section in Chrome's inspect mode. This realization came late in the project, and it highlighted an opportunity we hadn't fully utilized.

Initially, we implemented a solution to gather the required information ourselves without the use of computed properties, possibly due to a lack of explicit communication or oversight. However, upon understanding the potential of accessing computed properties, we started working on an improved approach. This approach would allow us to perhaps enhance the extension's functionality for analyzing buttons, and would also make it easier to implement similar functionality for other elements like images. By addressing this challenge, we laid the groundwork for more flexible and comprehensive future development as mentioned under "Future Work".

3) Structure and Workflow: At the project's outset, we found structuring our work and establishing an efficient workflow to be challenging. This was primarily due to the task being new and the teammates not having met before. However, as we gradually got to know each other better and gained deeper insights into the tasks, our workflow improved.

4) Dividing Work: This project has been one of two projects the team has been working on this summer. The presence of two distinct tasks presented a challenge in determining how to divide the work effectively. Initially, we decided that the best would be to allow everyone to contribute to both tasks. Eventually, two groups naturally formed, leading to more structured teamwork with each group focusing on its designated task.

VIII. Discussion on the future use of computed properties

1) Technical challenges in the use of computed properties: Essential accessibility information related to elements on a web-page can (and should) be derived from the computed properties of the accessibility tree. This is information that is manually and programatically available through the developer tool in Chrome's inspect mode. However, the computed properties of an element/node does not carry all the information we need in order to do the manual testing of the buttons. Thus the crux of the problem; both the DOM tree and the Accessibility tree (AXTree) contains information that we need, but connecting the nodes from the two trees proved to be a great challenge. Part of the reason for this is that it is not always a 1:1 relationship between nodes in the two trees. Additionally, it seems that there is no current information readily exposed to developers that will connect the nodes in the two trees.

A. What have we tried:

At first this seemed like a manageable task - we were able to retrieve the accessible name (which is part of the computed properties) and its underlying attributes from which the accessible name is computed. However, the computed properties did not contain any information about the elements' HTML and as such, we did not have any way of connecting the computed properties to a specific element from the DOM (where the HTML of an element is available). After some research on the relationship between the DOM tree and the AXTree we thought that the nodeId (available as a property of a node in the AXTree) could be used to establish a relationship between nodes across trees - but it could not.

B. The development branch

The development branch is a result of experimenting and trying to work around the problem. As of now, we have set up a post-request to a backend endpoint which in turn fires up a headless chrome and queries the accessible tree from the relevant URL. Once we have the AXTree we extract the role and name property of the elements which have 'role = button'. The computed properties are sent back to frontend but not

handled any further. From our experience of manually testing different websites the solution finds close to every relevant button and returns its accessible name. However, as described above, it does not return an HTML element and thus has no connection to the DOM tree.

C. Our thoughts moving forward:

Our current thoughts on this matter is that there is no readily available technology to handle this specific problem. There might be ways of working around the issue; for instance if one were to change the demand for the HTML-element and only require the information provided by the AXTree. There is also some interesting reading on AOM (Accessibility Object Model) that (when available) might aid the quest for connecting the two trees (https://wicg.github.io/aom/explainer.html).

IX. Conclusion

We have successfully developed Digdir's internal system, known as UU-extension, which is specifically designed to assist UU-tilsynet in their universal design endeavors in Norway. The UU-extension system offers the potential to streamline their testing processes by enabling semi-automatic testing, thereby alleviating the burden of manual testing procedures. Nevertheless, it is essential to acknowledge that, in its current state, the system does face certain challenges and requires further improvements, as outlined in the future work and challenges and solutions sections of our report.