



Expertise
and insight
for the future

Risto Salama

“Down With Regression!” – Generating Test Suites for the Web

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

30 April 2020

PREFACE

I want to give thanks to my caring wife, who took the main role of looking after our firstborn baby and household in these difficult times of epidemic. This helped me immensely to focus on finishing the project described in this thesis in time.

Helsinki, 29 April 2020
Risto Salama

Author Title	Risto Salama "Down With Regression!" – Generating Test Suites for the Web
Number of Pages Date	62 pages + 2 appendices 30 April 2020
Degree	Master of Engineering
Degree Programme	Information Technology
Instructors	Ville Jääskeläinen, Principal Lecturer Mikko Mäkipää, Director, ICT and digitalization
<p>Software regression can and have caused problems in the software industry. Regression can be mitigated by extensive test suites and laborious manual testing, both which cost time and money.</p> <p>Web accessibility is a topic that helps to ensure that Internet is accessible to everyone; Internet should not be restricted by abilities or disabilities of a person. There exist guidelines and tools that can help to ensure accessibility of a website.</p> <p>The goal of this project was to create a testing tool that could help to automate tests both for regression and accessibility. This tool should be on a level of a proof of concept; it should demonstrate that it is indeed possible to test both the regression and accessibility of a website. There was no need to make it so ready in the scope of this thesis that it could be used in a production.</p> <p>This tool was implemented using key technologies such as crawling, scraping, Lighthouse, Puppeteer, etc. First a list of specifications was generated and the implementation of the proof of concept managed to accomplish them all. However, many things can be improved or implemented at a later stage.</p>	
Keywords	regression, crawling, accessibility, testing

Tekijä Otsikko Sivumäärä Aika	Risto Salama "Down With Regression!" – Generating Test Suites for the Web 62 pages + 2 appendices 30 April 2020
Tutkinto	Master of Engineering
Koulutusohjelma	Information Technology
Ohjaajat	Ville Jääskeläinen, Principal Lecturer Mikko Mäkipää, Director, ICT and digitalization
<p>Sovellukselle tehdyistä muutoksista aiheutuvat virheet ovat pitkään olleet ongelma IT-alalla. Näitä virheitä pystytään estämään laajoilla automaatiotesteillä ja manuaalisella testaamisella, mutta molemmat maksavat aikaa ja rahaa.</p> <p>Saavutettavuudella autetaan ihmisiä pääsemään käsiksi nettisivustojen sisältöön ja toimintoihin, huolimatta siitä minkälaisia fyysisiä, teknillisiä tai psyykkisiä esteitä käyttäjällä on. On olemassa oppaita ja teknologioita mitkä voivat auttaa varmistamaan nettisivun saavutettavuuden.</p> <p>Tämän projektin päämääränä oli tehdä testaustyökalu, joka voisi auttaa automatisoimaan testejä muutosten aiheuttamia virheitä sekä saavutettavuusongelmia vastaan. Työ rajattiin niin, että testaustyökalusta tehdään versio, joka riittää osoittamaan työkalun olevan toteutuskelpoinen.</p> <p>Työkalu toteutettiin käyttäen erilaisia teknologioita kuten nettisivustojen crawlaus ja taltiointi sekä Lighthouse ja Puppeteer. Työssä määriteltiin lista ominaisuuksista mitä testaustyökalu vaatii ja nämä toteutettiin onnistuneesti projektissa. Testaustyökalussa on kuitenkin monta ominaisuutta mitä voitaisiin päivittää sekä myös mahdollisesti lisätä uusia ominaisuuksia, mikäli työkalua päätetään jatkokehittää.</p>	
Avainsanat	virheet, crawlaus, saavutettavuus, testaus

Contents

Preface

Abstract

List of Abbreviations

1	Introduction	1
2	Project Specifications	6
2.1	Current State Analysis	6
2.2	Specifications	7
3	Accessibility	11
3.1	Web Content Accessibility Guidelines	12
3.2	Accessibility Tools	15
4	Theoretical Background	21
4.1	Web Crawling	21
4.2	Web Scraping	21
4.3	UI Testing Tools	22
4.4	Visual Regression Testing	23
5	Implementation of Proof of Concept	24
5.1	Project Setup	24
5.2	Configuration	25
5.3	Crawling & Recording	28
5.4	Verifying	34
5.5	Accessibility	40
5.6	Testing	45
5.7	Reports	47
6	Conclusions	59
6.1	Challenges	61
6.2	Future Improvements	62

References

Appendices

Appendix 1. Code Listing For Crawling

Appendix 2. Accessibility Rules

List of Abbreviations

CI	Continuous Integration
CLI	Command Line Interface
CMS	Content Management System
CSA	Current State Analysis
DOM	Document Object Model
FIOH	The Finnish Institute of Occupational Health
GUI	Graphical User Interface
POC	Proof of Concept
SPA	Single Page Architecture
SVG	Scalable Vector Graphics
UI	User Interface
UX	User Experience
WCAG	Web Content Accessibility Guideline

1 Introduction

As new features are added into an application, old features removed from the application, bug is fixed or some software dependency is updated to a newer version, there exists a chance that one or more other features break creating unwanted changes. This is called regression and it can occur at any point in software's lifecycle. Regression can be mitigated to a degree by an extensive test suite, continuous development, manual acceptance testing and good coding practices.

When it comes to software with GUI (Graphical User Interface), like most web applications, there are often multiple different tests to be run with different purposes. End-to-end testing for use cases that touch every part of the software from database to GUI, integration testing for API, unit testing for granular class or function level testing and user interface (UI) testing for testing the GUI itself, to which the topic of this thesis focuses on.

Creating an extensive test suite for the UI to prevent regression can require a lot of repetitive work. However, this work can be automated. Bots, such as googlebot, have technology to crawl and roam the websites by going through one link to another, information which they can use for indexing for example. Document Object Model (DOM) of website can be utilized with headless browsers to click any element and check if DOM or webs address changes as a result. These changes can be recorded and a test suite generated for the website that, when executed, will either pass or fail depending if the recorded changes were resolved or not.

The Finnish Institute of Occupational Health (FIOH) is a multidisciplinary research and specialist organization that focuses on well-being at work, research, advisory services and training [1]. Organization operates under the Ministry of Social Affairs and Health as an independent legal entity, and has around 500 employees.

FIOH commissioned a tool that can help with regression of website functionality and accessibility. Accessibility is covered later in this chapter but briefly it can be described that website is accessible if any person with any abilities or disabilities is able to use it, accessibility is a way to measure how accessible is the website. This commissioned tool uses as much existing technology as possible.

The idea was to create a tool for developers to programmatically test a user interface of an application by commanding a browser to click through links and press buttons and record all changes to DOM and all other actions that were taken. When the tool is rerun against the same application on a later time, it will compare the results of the rerun to those recorded. Changes from the comparison are recorded in such a form that it is useful for developer to identify what has changed.

If nothing has changed from previous recording, then a developer can safely presume that everything is working as intended. If something has changed, however, then a developer must check that changes are desired. In case changes were desired, then developer can just make a new recording with the tool to be used with any future runs. In case changes were not wanted, then developer has to fix the issue causing them until tool passes the check without any changes. Following activity diagram in Figure 1 describes the flow of these two use cases.

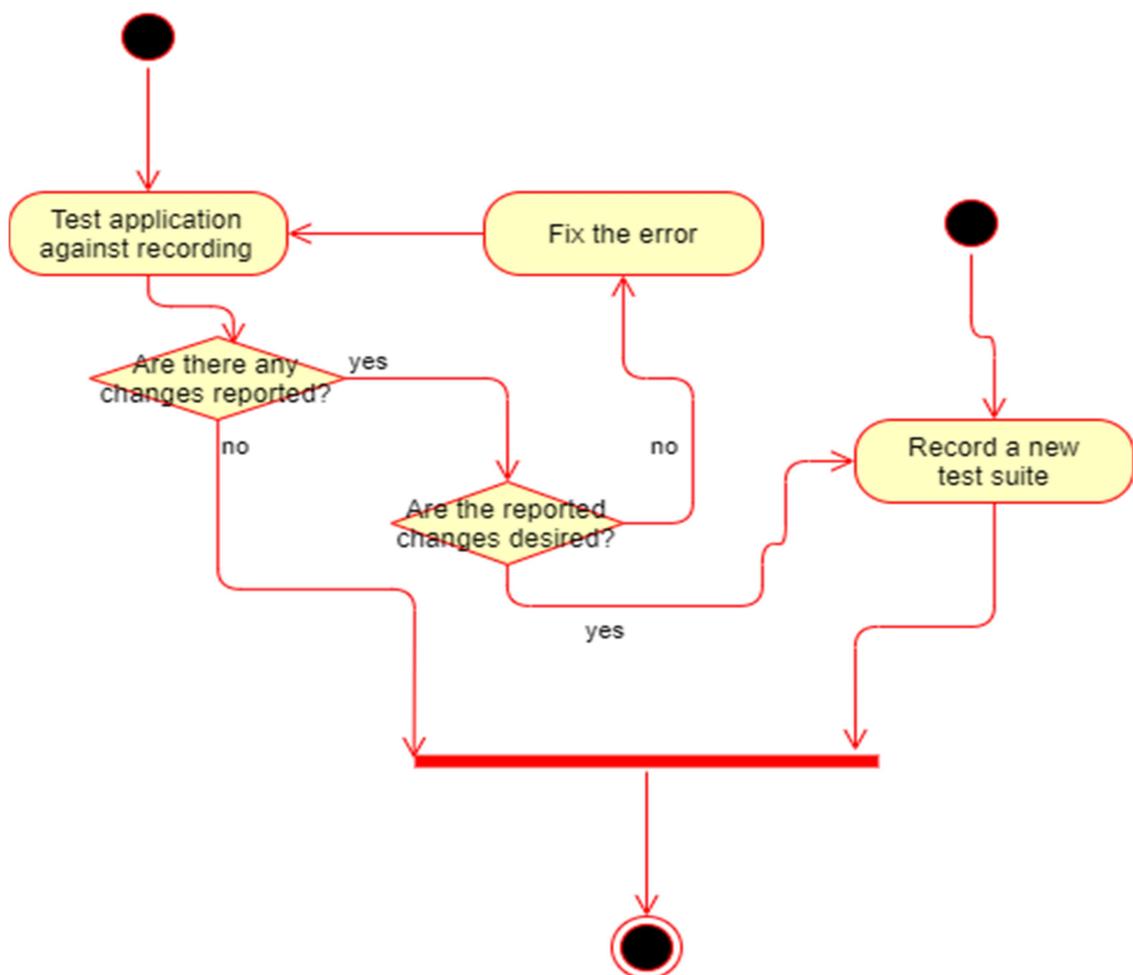


Figure 1. Activity diagram of the test tool

Note that the tool can be also executed in CI (Continuous Integration) environment as one of the tests to be checked before build passes. In such a case it can be useful to have an option in CI to automate the recording of a new test suite. This option can be useful when the changes to the tested project are desired.

An accessibility directive set on place by European parliament came into effect on 22.12.2016 [2]. In the directive accessibility is stated as such: “In the context of this Directive, accessibility should be understood as principles and techniques to be observed when designing, constructing, maintaining, and updating websites and mobile applications in order to make them more accessible to users, in particular persons with disabilities.” [3]. This accessibility directive sets minimum level of accessibility for online services provided by a public administration. The directive also describes methods how implementation of the accessibility guidelines is moderated.

In Finland legislation requiring online services provided by public administration to reach desired level of accessibility was set into effect on 1.4.2019 and the goal to reach the required level of accessibility starts stepwise 23.9.2019:

- 23.9.2018 and after published websites must be accessible by 23.9.2019
- Websites published before 23.9.2018 must be accessible by 23.9.2020

FIOH has some online services that have been published before 23.9.2018 and legislation demands that they must be accessible by 23.9.2020. This thesis attempts to find a reasonable answer in a form of a developer tool to the following research questions:

- How unwanted changes can be noticed better?
- How can the accessibility of a website be automatically tested?

The target of this thesis is to create a POC (proof of concept) testing tool that allows testing both the regression and the accessibility of a website. From the scoping point of view, it is enough that the tool can be used with some websites, the tool does not have to work with all websites and it does not have to be error free. Deployment of the tool is not part of this thesis. Excluded from the scope are also items of Web Content Accessibility Guideline (WCAG) that cannot be automatically tested using current tools and libraries.

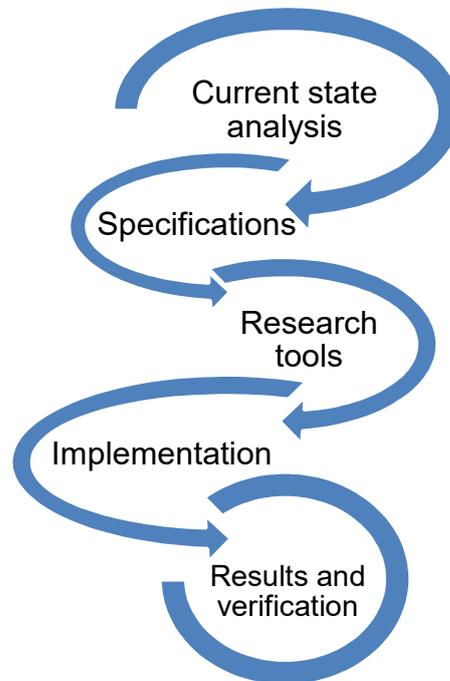


Figure 2. Phases of project work.

The development of the test tool was done in the following order. First a Current State Analysis (CSA) on different software projects inside FIOH was conducted, with emphasis on frontend projects and accessibility. After the CSA was done, specifications for the POC tool were made. Researching and picking suitable tools and libraries for the POC was the next step. Implementation of the POC took place after and finally the results were tested and verified.

This thesis has been divided into 6 sections. The first and current section, Introduction, is an introductory section of this thesis that had covered briefly the main topics of this thesis. The second section, Project Specifications, contains both CSA and the specifications of the test tool. The third section, Accessibility, covers the topic of accessibility in a greater detail. The fourth section, Theoretical Background, describes technologies and topics related to this thesis and to the POC. The fifth section, Implementation of

Proof of Concept, goes through the implementation of POC; how it works and demonstrates different reports the testing tool generates. The last section, Conclusions, goes through what was achieved, what challenges there were and what improvements can be made in the future.

2 Project Specifications

This section lists the specifications needed for the tool that tests both the regression and the accessibility of a website. This section also introduces the automated testing situation of some of the applications in FIOH, some of which are under active development and some which are in a maintenance phase. These applications are discussed here only in a surface detail and what those applications are or what they are used for is left unmentioned on purpose.

2.1 Current State Analysis

Investigated applications can roughly be divided into two sections: front-end applications with a single page architecture (SPA) and back-end applications. The architecture of applications has been implemented utilizing micro services approach and different combinations of those services are used for different products.

Back-end applications have been implemented either in Java using Spring Boot Framework [4] or in JavaScript utilizing Node.js [5]. All back-end applications have at least unit tests used with them. Back-end applications with a database access usually also have integration tests which tests the application together with a database schema utilizing Flyway [6] or Knex [7] database migration tools together with Docker[8].

More recently developed front-end applications are implemented utilizing Vue.js [9] framework while legacy applications have been implemented with AngularJS [10] framework. All front-end applications have at least unit tests used for automated testing.

Two of the products use end to end test suites, which encompass most if not all the micro services used to form the product. These end to end test suites usually test only the most common use cases but they test multiple layers of those products together from a user interface to a database layer and back.

All these tests are used together with CI to ensure that no software builds which fail to pass the tests are deployed to staging or production environments. Besides tests, CI also runs lint tasks to ensure that code follows good coding practices and is uniform.

Most of the products also have some automated accessibility tests and they are included also in CI. However as accessibility rules are not yet enforced in most of the products, if software build fails to pass the accessibility tests, it does not affect deployment of application as other tests do. There is one product that enforces passing accessibility, search engine optimization (SEO) and some best practices test suites before deployment. The project also has some performance tests implemented, but the current performance tests are deemed too fragile to be part of the deployment checks.

Accessibility tests are not however all encompassing, and having them automated does not ensure WCAG guidelines are completely followed. Tools and test frameworks used for automated accessibility testing do not support some more complicated HTML elements, for example SVG (Scalable Vector Graphics) or canvas. FIOH's Work-life Knowledge Service has stated following [11]: "The website of the Work-life Knowledge Service has been implemented mostly in accordance with level A of the Web Content Accessibility Guidelines (WCAG 2.1)." It is not necessary feasible to achieve certain level of WCAG with an application and maintain that level due to problems described with the tools and the cost of testing accessibility manually to ensure it never fails to meet accessibility guidelines.

In case new subpages are added into the product, the automated accessibility tests must currently be manually updated to include those subpages. One of the products has functionality implemented to scan for new subpages that can be added to the accessibility tests, but the functionality scans only for subpages in a very specific group and cannot find any subpages excluded from that group.

2.2 Specifications

This section contains the list of specifications intended for the tool used for both regression and accessibility testing. Section also explains the purpose of each specification.

Table 1: Specifications of the Proof of Concept

Specification / Attribute / Characteristic	Purpose
Crawler	To find all the subpages of the website
Scraper	To record DOM on all the subpages
Captures Screenshots	Captures screenshots of the web pages in the website to be used for both verification and reporting
Storing the scraped content with screenshots	Scraped content with screenshots must be stored for later verification use
Verification	Verifies whether the website has changed
Reporting via command line interface (CLI)	Reports all the changes found during verification
Visual report	Generates visual report listing all the changes found during verification
Accessibility checks	The tool tests the accessibility of web-

	site
Ability to log in	The tool can be used to crawl private websites
Ability to ignore marked content	Some content like time changes all the time and should be ignored during verification
Configurable	The tool can be configured to run on different websites requiring different information like log in credentials
Support static website	The tool can be run against a static website
Support SPA	The tool can be run against SPA

The tool has to be able to crawl a webpage to find all the possible subpages. If the webpage is not crawled, then the subpages cannot be automatically found and thus would have to be parameterized separately.

The tool has to be able to scrape the contents of the crawled webpage. DOM of the webpage and its subpages must be gathered. This is required for verification and helps to ensure that structure of the webpage is intact.

The tool has to take screenshots of webpage and its subpages. This is also required for verification and ensures that the visuals of the webpage are intact.

The tool has to be able to store the scraped DOM and captured screenshots of the webpage. This is required for verifying that there is no regression. The stored information can optionally be overwritten in case there is no regression and the changes to the webpage are desired.

The tool has to be able to verify whether the webpage has changed or not. Verification is made by comparing the recently scraped data against previously stored scraped data and by comparing recently captured screenshots against previously stored screenshots.

The tool has to be able to report to the user all the changes verification process finds. This report should be outputted in the used command line interface tool and should specify the changed content. Also visual report should be generated, so user can see how the screenshots compared differ on various subpages.

The tool has to be able to check and report what accessibility problems are identified on the webpage and its subpages. This includes only accessibility problems that can be automatically tested.

The tool has to be able to log in to a webpage. Some webpages on FIOH require that user logs in to be able to access the content. In order to test such a webpage, the tool must be able to log in with given credentials.

The tool can be configured to run against different websites with different parameters. The tool should ideally have configuration files which contain the address of the website and possibly login information required by website.

Lastly the tool should support both static websites and websites made with popular SPA technologies. These two implementation approaches differ greatly from each other, but they are both very common and the tool should be able to adequately support both of them.

3 Accessibility

This section covers the topic of accessibility: what accessibility means and the reasons why accessibility is important. Following subsections go through Web Content Accessibility Guidelines (WCAG) and the tools related to accessibility.

“The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect.” as stated by Tim-Berners-Lee, inventor of the World Wide Web. Regardless of what is a person’s location, ability, language, software or hardware the Web has been fundamentally designed to serve everyone. Purpose of Web accessibility is that people with disabilities can use websites, tools and technologies [12]. When web pages are accessible, they serve a diverse audience of people with different backgrounds and both mental and physical abilities. Accessibility is an essential aspect for organizations that want to create websites and web tools and do not want to exclude people from using their products and services.

Accessibility encompasses all disabilities that can affect access to the Web whether they are related to vision, hearing, physical, speech or mental abilities. Accessibility is also beneficial to the people who use different devices like mobile phones or screen readers, have temporary disabilities like a broken arm, have situational limitations such as being in a noisy environment where they cannot listen to audio and even people with slow Internet connection.

If a web page has a low color contrast ratio between text and background, such as using black and gray colors, that particular content can be inaccessible for a person with poor vision. Likewise, if the color contrast ratio would be good, but the font size of the text would be too small, the person with poor vision could not again properly see the content. If a web page has multiple clickable elements that are close together, that may very well be usable with a mouse but could be frustrating or even impossible to use with touch sensors of a mobile screen rendering those elements inaccessible to a mobile user. If a web page has audio content but does not provide subtitles that particular content is inaccessible to a deaf person. If there are elements on a web page that cannot be reached using keyboard commands, it makes those elements inaccessible to a person using only a keyboard. Even a whole webpage can be made inaccessible to a person if the content requires user action to continue and blocks other elements of the web

page. Example for such a case can be a simple notification for user that requires user to click some button that is not accessible.

Web has removed many physical barriers for people to partake in vast amounts of information, education, entertainment, different tools and can also enable people to contribute to those areas. But if tool or website has not been designed with accessibility in mind, it can create barriers to people to use or access the content.

Accessibility testing is either a manual or automatic process or combination of both to ensure that GUI is usable by people of all abilities and disabilities. Ideally website is tested both with automated tests and manual tests, particularly manual tests made by users with varying disabilities. Arranging manual tests with users that have disabilities can be quite challenging for small organizations to arrange [13].

3.1 Web Content Accessibility Guidelines

Web Content Accessibility Guidelines (WCAG) 2.1 [14] by the World Wide Web Consortium (W3C) contains recommendations for making Web content more accessible. Guidelines are not specific to any particular technology but are intended to apply for all different Web technologies that exist in the present and in the future. Success criteria in WCAG 2.1 are written as testable statements. WCAG 2.1 extends previous 2.0 - version so that if content that conforms to 2.1 then it also conforms to 2.0. However, it is recommended that the current version of WCAG is utilized when updating accessibility policies for a website.

These recommendations provided by WCAG are categorized into four main categories called principles. The four principles and their definitions provided by WCAG are shown in the following Table 2:

Table 2: WCAG recommendations

#	Principle	Definition
1	Perceivable	Information and user interface components must be presentable to users in ways they can perceive
2	Operable	User interface components and navigation must be operable.
3	Understandable	Information and the operation of user interface must be understandable.
4	Robust	Content must be robust enough that it can be interpreted by a wide variety of user agents, including assistive technologies.

Guidelines such as having text alternative for any non-text content like images or guideline that color is not alone used to convey information are categorized under the first principle: Perceivable. Guidelines like bypassing blocks of content utilizing anchors or guideline that all functionality is operable with keyboard interface are categorized under the second principle: Operable. Guidelines such as that language of the Web page is programmatically determinable or guideline which states that context sensitive help is offered to the user are categorized under the third principle: Understandable. The last principle, Robust, contains guidelines like compatibility with user agents so that there are no duplicate identifiers in content made with markup language..

Web page conforms to WCAG 2.1 only if one of the conformance levels is reached, that is all the requirements indicated by conformance level are satisfactory met. There are three different conformance levels A, AA and AAA. Levels are hierarchical; in order

for Web page to reach conformance level AA, all level A requirements must also be met in addition to level AA requirements. Note that conformance level AAA is not recommended to be general policy for the entire site because it is considered impossible by W3C for all the content to satisfy level AAA requirements.

Generally it can be considered that requirements of a higher conformance level affects the Web page design and development more than the lower and requires more work to achieve. For example, while captions for prerecorded audio content is mandatory to reach even level A, level AAA requires that there is also a sign language interpretation for the audio.

W3C has also provided quick reference tool [15] for checking WCAG requirements and techniques how those requirements can be fulfilled. The tool provides filtering options so user can filter rules that are connected to animation for example.

The screenshot displays the WCAG Quick Reference Tool interface. The browser address bar shows the URL: www.w3.org/WAI/WCAG21/quickref/?showtechniques=121%2C141¤tsidebar=%23col_customize. The page title is "How to Meet WCAG (Quickref R...". The main content area is titled "Guideline 1.4 – Distinguishable" with the sub-heading "Make it easier for users to see and hear content including separating foreground from background." Below this, the section "1.4.1 Use of Color — Level A" is highlighted. The text explains that color is not used as the only visual means of conveying information. A note states: "Note 1: This success criterion addresses color perception specifically. Other forms of perception are covered in Guideline 1.3 including programmatic access to color and other visual presentation coding." The tool provides filters for "Sufficient", "Advisory", and "Failures" techniques. Under "Sufficient Techniques", it lists: "Situation A: If the color of particular words, backgrounds, or other content is used to indicate information:" followed by G14, G205, G182, and G183. "Situation B: If color is used within an image to convey information:" followed by G111 and G14. Under "Advisory Techniques", it lists C15. Under "Failures", it lists F13, F73, and F81. The left sidebar contains filters for WCAG Version (WCAG 2.1), Tags (Developing, Interaction Design, Content Creation, Visual Design), Levels (Level A, AA, AAA), Techniques (Sufficient, Advisory, Failures), and Technologies (HTML, CSS, ARIA, Client-side Scripting, Server-side Scripting, SMIL, PDF, Flash, Silverlight).

Figure 3. WCAG Quick Reference Tool

The quick reference tool is shown in action in Figure 3. The tool provides information about techniques that are enough to meet the requirement. The tool also provides advice what techniques can additionally be used and explains the failure cases.

3.2 Accessibility Tools

This section covers briefly what kind of accessibility tools exists. Mostly tools can be divided into three categories: online tools that are often commercial, browser plugins

and command line tools. Some of the tools are able to show wide range of accessibility violations in the website while others are narrower. However none of the tools encompass all possible issues as outlined by WCAG 2.1 and there are even arguments that it is not even technologically feasible to cover them all.

As stated by Marcy Sutton, team member of axe core libraries, accessibility issues have to be able to be detected with code that works cross platform and has no false positives [16]. Mrs. Sutton further states that some of the trickier items in WCAG 2.1 cannot be automatically tested due to lack of web platform APIs. This means that website developers and designers should keep in mind that not every item in WCAG can be automatically tested and manual testing might be required to confirm that website's accessibility achieves some level of WCAG 2.1. Still testing of many items outlined by WCAG 2.1 can indeed be automated. Furthermore the website tested may not even have content such as audio or video thus eliminating need to test items in WCAG 2.1 altogether.

Axe

A popular toolset (452,922 weekly downloads for axe-core as reported by npm [17] in 8 Sep 2019), axe [18] provides both tools to test the accessibility of web page and api for other tools to use axe's open source accessibility rule libraries.

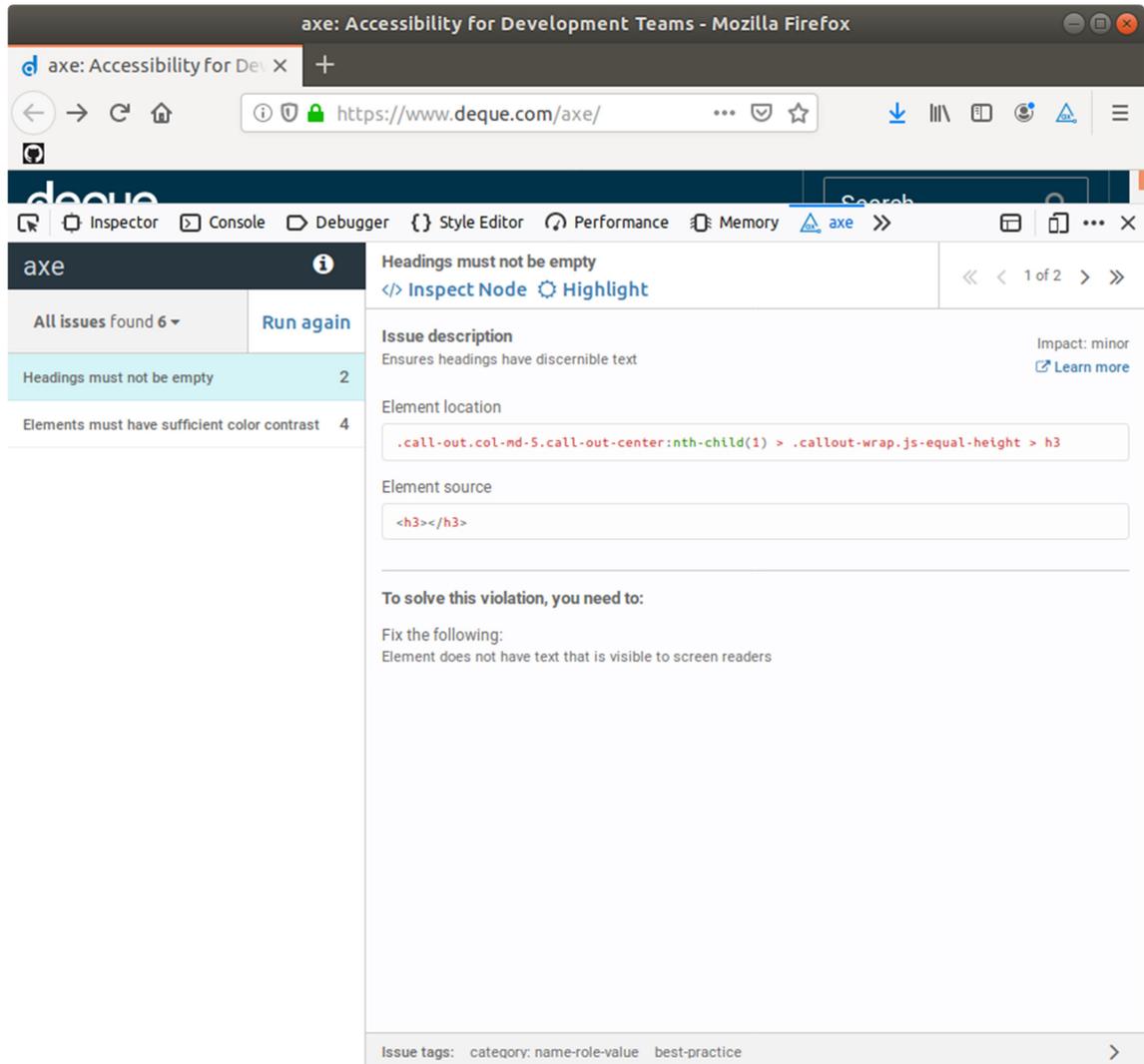


Figure 4. Axe plugin in action on Firefox.

In Figure 4 is a demonstration what Axe plugin run on Firefox browser outputs. It describes the violations and displays the location of the issue in the DOM tree.

Lighthouse

Lighthouse [19] is an open-source, automated tool for improving quality of web pages. It provides both plugin for browsers if not already integrated (see Figure 4) and command line interface. Besides accessibility, Lighthouse audits web pages for variety of topics including performance, progressive web application and search engine optimization. It uses axe's core libraries for accessibility rules.

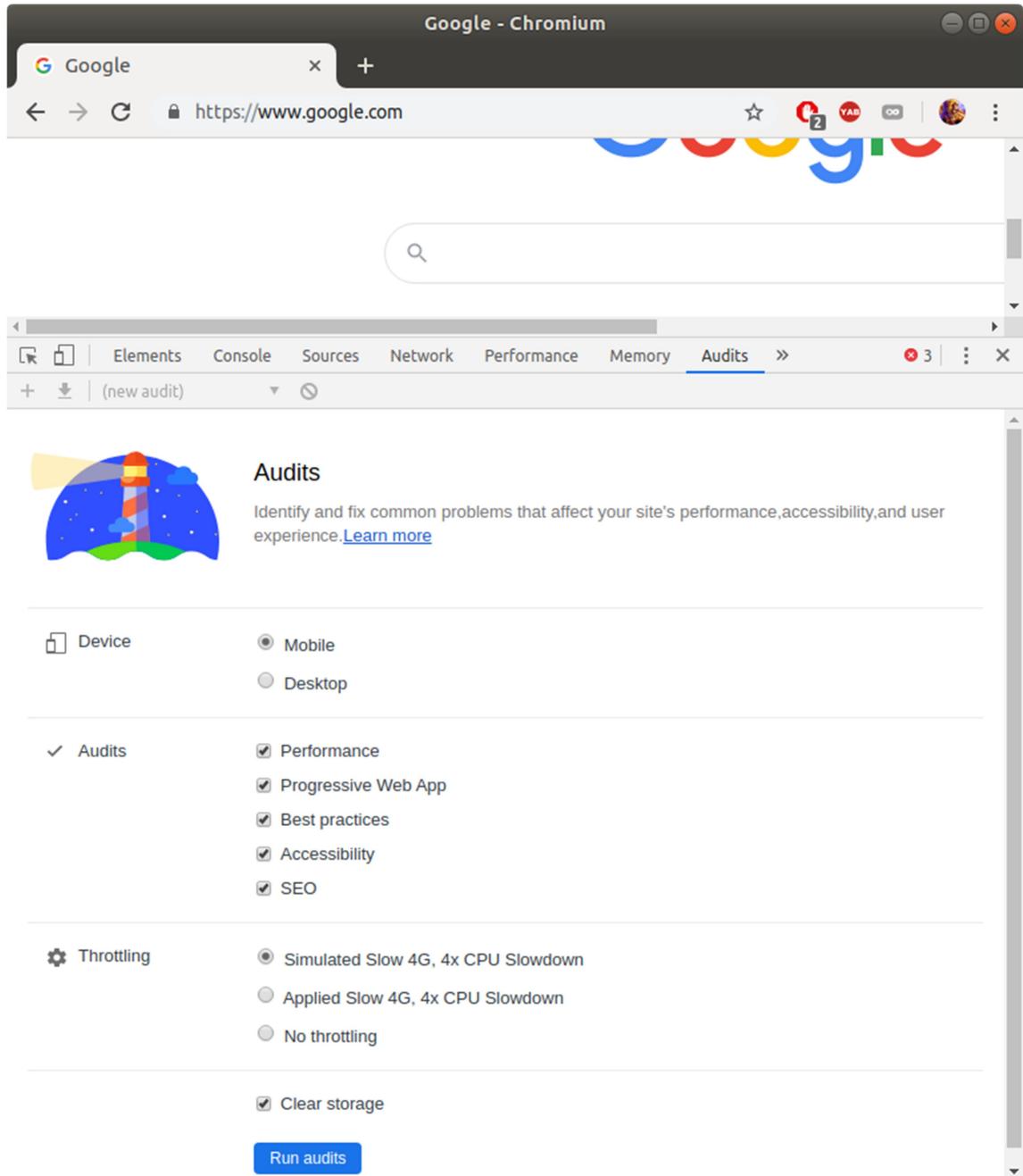
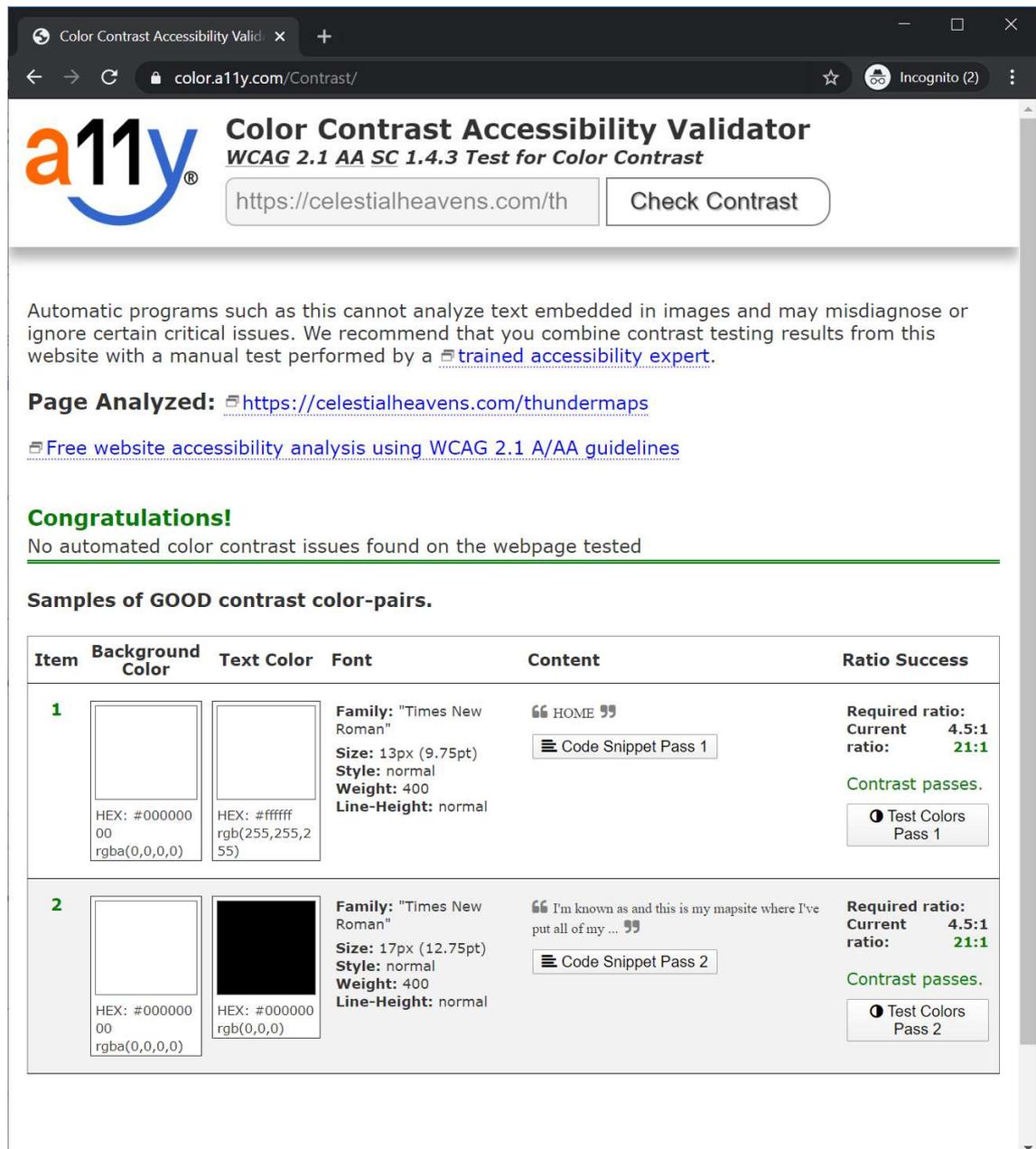


Figure 5. Lighthouse plugin

As shown in Figure 5, Lighthouse can also be run in either mobile or desktop to see potential flaws in both forms as many websites can have some functionality that is hidden in one of the modes and the visual style can be vastly different between the various screen sizes. There's also support for throttling which can help developers to see how the website manages on slower machines or connections.

Color Contrast Accessibility Validator

Color Contrast Accessibility Validator [20] by a11y is an online tool for checking color contrast of a webpage's foreground and background colors. However, it states that it is not able to analyze color contrast of text embedded in images and may misdiagnose.



The screenshot shows the Color Contrast Accessibility Validator interface. The browser address bar displays `color.a11y.com/Contrast/`. The page title is "Color Contrast Accessibility Validator" with the subtitle "WCAG 2.1 AA SC 1.4.3 Test for Color Contrast". A text input field contains the URL `https://celestialheavens.com/th` and a "Check Contrast" button is visible.

A warning message states: "Automatic programs such as this cannot analyze text embedded in images and may misdiagnose or ignore certain critical issues. We recommend that you combine contrast testing results from this website with a manual test performed by a [trained accessibility expert](#)."

The "Page Analyzed:" section shows the URL `https://celestialheavens.com/thundermaps` and a link to "Free website accessibility analysis using WCAG 2.1 A/AA guidelines".

A green "Congratulations!" message follows, stating: "No automated color contrast issues found on the webpage tested".

The "Samples of GOOD contrast color-pairs." section contains a table with two entries:

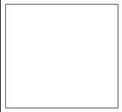
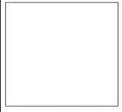
Item	Background Color	Text Color	Font	Content	Ratio Success
1	 HEX: #000000 00 rgba(0,0,0,0)	 HEX: #ffffff rgb(255,255,255)	Family: "Times New Roman" Size: 13px (9.75pt) Style: normal Weight: 400 Line-Height: normal	“ HOME ” <code>Code Snippet Pass 1</code>	Required ratio: 4.5:1 Current ratio: 21:1 Contrast passes. Test Colors Pass 1
2	 HEX: #000000 00 rgba(0,0,0,0)	 HEX: #000000 rgb(0,0,0)	Family: "Times New Roman" Size: 17px (12.75pt) Style: normal Weight: 400 Line-Height: normal	“ I'm known as and this is my mapsite where I've put all of my ... ” <code>Code Snippet Pass 2</code>	Required ratio: 4.5:1 Current ratio: 21:1 Contrast passes. Test Colors Pass 2

Figure 6. Color Contrast Accessibility Validator

Color Contrast Accessibility Validator is shown in action in Figure 6, where it has assessed that a webpage has sufficient color contrast and passes the test. It also demonstrates samples of color-pairs on webpage with good contrast.

4 Theoretical Background

This section describes the relevant topics that are used in this thesis. Some of the topics are related to crawling while others are related to accessibility, verification and reporting. This is not intended to be a comprehensive listing of applicable tools but to highlight couple different technologies to choose from or which are related to the topic somehow and provide a reasoning why this area is important.

Note also that this section focuses on the tools that are intended to be used with programming languages that were selected for implementing the proof of concept: some JavaScript but mostly its superset TypeScript. This choice for programming languages was partially determined by the current technology stack used in the organization.

4.1 Web Crawling

A web crawler [21] also referred as spider or search engine bot, searches the website primarily for search engine purposes. Crawler goes through the webpage, going through subpages of the webpage by all the links it can find and gathers all the relevant data it finds for search engine indexing. Thus a crawled website can be found in a search engine like Google Search by its name, keyword or content it has.

Since one of the purposes of this thesis work is to automate regression testing process of a website as much as possible, mainly by automatically finding subpages of the website and navigating through them, this is a relevant technology.

Googlebot

Googlebot is a name used to describe Google's web crawlers [22]. There are two types of web crawlers used by Google: one for mobile devices and another for desktops and they both simulate users. Googlebots are designed to scale and to be run simultaneously on thousands of machines.

4.2 Web Scraping

Web Scraping is a term for technology that is used to scrape websites for a meaningful data. It can be used for various purposes such as research, price comparisons or extracting contact information. It is somewhat connected to Web Crawling as indexing

technology relies on scraping the website to get the meaningful content to show on search engines.

As the proof of concept created during this thesis aims to scrape the DOM tree of the website for verification purposes for regression testing, this is a relevant technology.

Scrapy

Scrapy [23] is a framework for extracting data from a website. The framework enables developer to create custom crawlers that can be instructed to scrape some data from a website. Scrapy is an open source project written in Python.

4.3 UI Testing Tools

One approach to make the website crawling is to use existing tools. Many UI testing tools are already geared towards testing client implementations on a browser; they allow executing code on the browser and even capturing screenshots of the browser window which can both be useful for verification purposes.

Headless Browser

Headless Browser [24] is a web browser that is run without any GUI. Headless Browsers can be used for example to scrape data from a website or to test web pages programmatically. They are often default choice when running automation tests and browsers like Firefox and Chrome provide headless mode to run them.

Nightwatch.js

Nightwatch.js [25] is an end-to-end testing framework for Node. It can be used to test websites or web applications. Nightwatch.js uses W3C Webdriver [26] under the hood and supports multiple browsers.

Puppeteer

Puppeteer [27] provides a high-level API for controlling Chrome or Chromium browsers over the DevTools Protocol [28]. It is a Node-library and runs browser's in headless mode by default but can easily be configured to run on non-headless mode. Among other features it provides supports for capturing screenshots, UI testing tools and running code in browser. However, it does not directly support browsers besides Chrome and Chromium.

4.4 Visual Regression Testing

Visual regression testing aims to test the visual appearance of website. There are few tools written for visual regression testing, but many of them are not maintained anymore. Nevertheless parts of them can be useful for researching how to implement screenshot comparison.

Wraith

Written in Ruby, Wraith [29] is a responsive screenshot comparison tool that can either be used to compare two different websites or compare history of one site by regularly adding new screenshots.

5 Implementation of Proof of Concept

This section describes the implementation of POC. Following seven subsections cover most of the project's implementation details. The first subsection is Project Setup, which explains the structure and tools selected for the project. The second subsection is Configuration, which describes what configuration options were implemented and for what purpose they are used. The third subsection, Crawling & Recording explains both the crawling process in greater detail and how crawling results were stored. The fourth subsection, Verifying, explains how the tool verifies that there have been no regressions. The fifth subsection is Accessibility and covers in detail how accessibility was covered with this tool.

Last two subsections cover the topics testing and reporting. The sixth subsection is about testing, describes how the tool was tested and what kind of testing utilities were used. The last subsection demonstrates how and what kind of reports the tool outputs.

5.1 Project Setup

CLI version of LightHouse was decided to be utilized for accessibility testing since it is free, open source, actively developed and has already been used in other projects in FIOH. For crawling purposes, Puppeteer was decided to be utilized primarily for the same reasons as LightHouse. As these are both JavaScript libraries, it made lot of sense to utilize Node.js a JavaScript runtime, together with Node Package Manager (npm) [30].

Programming language for the project was selected to be primarily TypeScript [31]. It is a superset of JavaScript that brings static type definitions, which allows describing objects and that enables both better documentation, as the object itself documents what it is, and allows TypeScript compiler to validate that code is working correctly. JavaScript was still used for some sections, particularly related to the testing of the tool.

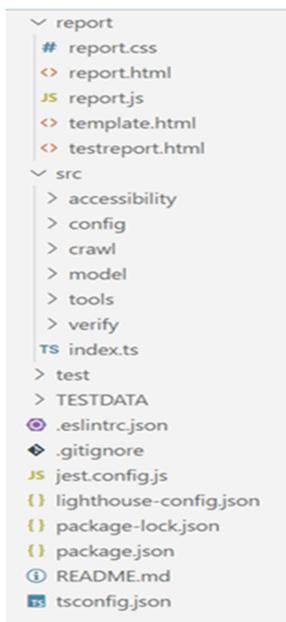


Figure 7. Directory structure

In Figure 7 is shown the directory structure of the project. Very popular Git [32] is used as a version control system. ESLint [33] is being used as linter to help ensure that code quality is good. It is run primarily and automatically in commit hooks, ensuring that no badly formatted code can enter version control. Jest [34] is a JavaScript testing framework, and here it is used with accessibility tests. Folder containing test data has all the crawled and recorded websites to be used with regression testing. The test folder contains tools used for making different scenarios for manually testing the tool.

5.2 Configuration

Configuration is necessary for running the tool against different websites. Also different websites might require user login and potentially different rules for what content to ignore while scraping the DOM. There is a base configuration file which contains interface for configuration that all other configuration files implement.

```
interface TimeConfig {
  wait: number;
  loadWait: number;
}

enum IgnoredContentType {
  TAG, ATTRIBUTE
}
```

```

interface IgnoredContent {
  value: string;
  contentType: IgnoredContentType;
}

interface Config {
  name: string;
  website: string;
  needsLogin: boolean;
  login(webSocketDebuggerUrl: string): Promise<void>;
  timeConfig: TimeConfig;
  ignoredContent: IgnoredContent[];
}

```

Listing 1. Configuration interfaces.

In Listing 1 is shown the configuration interfaces. Interface TimeConfig is utilized while crawling; some websites require more time as the page or the dynamic content loads. Interface IgnoredContent is used for leaving out some content while scraping the DOM of crawled sites. Some content can be, for example, time based and change on every run; running regression tests against such content would practically almost always fail. POC supports ignoring whole html elements (tags) or just attributes of elements.

There is a flag used for checking whether the login is necessary for testing the website, and if login is required, function for handling the login must be provided by object implementing Config interface. Name given to the configuration could be a project name for example and website in configuration has to be a web address to the website to test.

```

const defaultConfig: Config = {
  website: 'http://localhost:5005/',
  name: 'default',
  needsLogin: false,
  login: (): Promise<void> => {
    return Promise.resolve();
  },
  timeConfig: {
    wait: 20,
    loadWait: 1000,
  },
  ignoredContent: []
};

```

Listing 2. Default configuration.

In Listing 2 is shown the default configuration, which is used for testing the POC with different scenarios. No login is necessary so login method simply returns an empty promise. Units in time configuration are milliseconds.

```
const telluConfig: Config = {
  name: 'tellu',
  website: 'http://localhost:3500/#/fi/',
  needsLogin: false,
  login: (): Promise<void> => {
    return Promise.resolve();
  },
  timeConfig: {
    wait: 300,
    loadWait: 5000,
  },
  ignoredContent: [{value: 'svg', contentType: IgnoredContentType.TAG}]
};
```

Listing 3. Configuration for project titled tellu.

In Listing 3 is an example of configuration with some content ignored. Project tellu has lots of SVG elements generated runtime with different chart libraries and they have different attributes each time. In that context it is logical to leave out all the svg-elements from scraped content to enable regression testing against rest of the site without lots of false negatives.

```
const solmuConfig: Config = {
  name: 'solmu',
  website: site,
  needsLogin: true,
  login: async (websocketDebuggerUrl: string): Promise<void> => {
    const puppeteerBrowser = await puppeteer.connect({browserWSEndpoint: websocketDebuggerUrl});
    const page = await puppeteerBrowser.newPage();
    await page.goto(login);
    await page.content();
    await page.waitForSelector('#loginForm');
    await page.type('.user-data-field:first-child', 'fioh@fioh.fi');
    await page.type('.user-data-field:last-child', 'fiohfioh');
    await page.click('button.btn[type=submit]');
  },
  timeConfig: {
    wait: 100,
    loadWait: 1500,
  },
  ignoredContent: []
};
```

Listing 4. Configuration for project titled solmu.

In Listing 4 is an example of configuration with login functionality. Puppeteer is instructed to login to the solmu’s website using variety of selectors and mimicking user actions like typing and clicking.

```
...
"scripts": {
  "crawl:tellu": "tsc && node
    -e 'require(\"./dist/index.js\").main(\"tellu\").crawlAndMake()'",
  "verify:tellu": "tsc && node
    -e 'require(\"./dist/index.js\").main(\"tellu\").verify()'",
  "accessibility:tellu": "jest --configName='tellu'",
  ...
}
```

Listing 5. Scripts for configuration in package.json.

In order to add a configuration for a new website to be tested, the configuration must be imported into the configuration util and scripts for using the configuration have to be added to package.json -file like in Listing 5. The script “crawl:tellu” instructs the tool to run crawl operation with configuration for tellu project, The scripts “verify:tellu” and “accessibility:tellu” are likewise used to run the tool to verify there is no regression or to check that accessibility audits pass on tellu.

5.3 Crawling & Recording

Since only the web address is given from the website to use for testing, tool relies on crawling the website for recording the DOM and all the possible subpages. Crawling is done utilizing puppeteer which launches a headless browser instance and then goes through the website.

```
export const launchBrowser = async (config: Config,
  headless: boolean = true): Promise<Browser> => {
  const browser = await launch({
    headless,
    defaultViewport: {width: 1920, height: 1080},
    args: [
      '--no-sandbox',
      `--remote-debugging-port=${REMOTE_DEBUGGING_PORT}`,
      '--disable-web-security',
      '--ignore-certificate-errors',
      '--allow-insecure-localhost'
    ]
  });
  if (config.needsLogin) {
    logger.info('Login required. Logging into the service');
```

```

    const resp = await request.get({uri: `http://localhost:${REMOTE_DE
BUGGING_PORT}/json/version`});
    const {websocketDebuggerUrl} = JSON.parse(resp);
    await config.login(websocketDebuggerUrl);
  }
  return browser;
}

```

Listing 6. Launching the browser with Puppeteer.

Function for launching the browser instance is shown in Listing 6. In order to test sites with localhost and with potential cross origin resource sharing issues CORS [35] browser instance must be launched with flags that allow certificate errors, allow insecure localhost and disables web security. Function utilizes the login method in passed configuration parameter for websites that require user to make a login to properly test the site. Remote debugging port flag is required in order to use WebSocket [36] link provided by browser to make a successful login attempt to the site.

```

async function crawl(config: Config, action: CrawlAction):
  Promise<TestData> {
  const browser = await launchBrowser(config, false);
  const baseHref = config.website;
  logger.info('START of crawling');
  const screenshotFolder =
    getScreenshotFolderAndCreateIfItDoesNotExist(
      config.name ? config.name : config.website,
      action);
  const page = await browser.newPage();
  await page.goto(baseHref, {waitUntil: 'networkidle2'});
  const crawler = new Crawler(config, screenshotFolder);
  await crawler.crawlInternal(page, baseHref);
  browser.close();
  logger.info(`Found ${crawler.crawledPages.length} sites`);
  logger.info('END of crawling');
  return {
    pagesAndDoms: crawler.crawledPages,
    siteTested: baseHref,
    name: config.name
  };
}

```

Listing 7. Main function for crawling.

Main function for crawling as shown in Listing 7 is given the configuration object which contains the web address to crawl through among other required configurations. Function from Listing 6 is called at start then browser is instructed to navigate to the base webpage where the crawling begins.

Screenshot folder is determined using passed action parameter. There are two different actions crawling function is used for. The first action is to record crawled results in which case all the data is stored into the test data folder and it is going to be used in the future regression and accessibility test runs. The second action is to verify the site, which is used to run those regression tests against the previously stored data. Screenshots for verify action are written under temporary folder which is purged every time verification is rerun.

Main function creates a crawler object that handles the actual crawling through its `crawlInternal` method. Lastly the main function returns an object which contains name of the configuration, the website tested and an array containing crawled pages and their respective DOMs. Browser instance is shut down after the crawling has completed.

```
const DOM = await page.evaluate(): string => document.body.innerHTML)
const screenshotFile = `${this.screenshotFolder}/
  ${this.screenshotIdentifier++}.jpg`;
await page.screenshot({path: screenshotFile, fullPage: false,
  type: 'jpeg', quality: 25});
this.crawledPages.push({page: path, dom: this.cleanDOM(DOM),
  screenshotFile});
```

Listing 8. Scraping the DOM and taking a screenshot of the crawled page.

In Listing 8 Puppeteer's `evaluate` function is utilized to retrieve the DOM from the page. Evaluate function's contents are executed inside the browser's console and this enables retrieving DOM of the page. Lower quality screenshot is also taken of the page. Retrieved DOM is also removed from any content that is configured to be ignored. Both the screenshot and DOM are stored along the page address inside the array in the crawler object. This functionality happens inside `crawlInternal` method.

```
private cleanDOM(dom: string): string {
  let cleanedDom = dom.replace(/<span id="testmakerloadready"></span>
  `\/gi, '');
  try {
    this.ignoredContent.forEach((ignored: IgnoredContent): void => {
      if (ignored.contentType === IgnoredContentType.ATTRIBUTE) {
        const regex = new RegExp(`${ignored.value}=["^"]*"`, 'gi');
        cleanedDom = cleanedDom.replace(regex, '');
      } else if (ignored.contentType === IgnoredContentType.TAG
        && dom.includes(`<${ignored.value}`)) {
        // This presumes dom is proper HTML with end tags"
        const splitDom = cleanedDom.split(`<${ignored.value}`);
        cleanedDom = splitDom[0];
        for (let i = 1; i < splitDom.length; i++) {
```

```

        cleanedDom += splitDom[i].split(`</${ignored.value}>`)[1];
    }
}
});
} catch (e) {
    logger.warn(`Could not clean ignored content. Error: ${e}`);
}
return cleanedDom;
}

```

Listing 9. Removing ignored content from the DOM.

In Listing 9 is shown how the DOM is removed from ignored content. First there is a chance that span element that is used to indicate to the crawler that the web page has been loaded has not been removed. This is removed using regular expression with replace method. Afterwards ignored content is removed from retrieved DOM within a loop. In the case ignored content type is an attribute then regular expression is used to remove the specified attribute from all the elements. In the case ignored content is tag, then all html elements with same tag name, including their child elements, are removed from the DOM.

Already crawled pages should not be re-crawled as it would very likely end in an infinite loop. Likewise leaving the website to crawl other web pages is not desirable as tool could potentially end crawling finite but vast Internet. Both of these scenarios are prevented by the tool.

```

let items = await page.evaluate((obj): string[] => {
    const ret = [];
    for (const item of document.querySelectorAll(obj.sel)) {
        const href = item.getAttribute('href');
        const target = item.getAttribute('target');
        if (href
            && !href.includes('mailto:')
            && !href.includes('tel:')
            && !target
            && (!obj.base.includes('#') || (href.includes('#')
                && href.startsWith('#' + obj.base.split('#')[1])))
            && href.split('#').length < 3 // excludes anchor links in spa
            && (!href.includes('http') || href.includes(obj.base))) {
            ret.push(href);
        }
    }
    return ret;
}, {sel: 'a', base: this.baseHref});
items = items.filter((item: string): boolean => this.isNotAlreadyFound(
    path, item));

```

Listing 10. Fetching the suitable links from the web page.

In the Listing 10 the crawler is trying to search for suitable links inside the DOM of visited page. Puppeteer's evaluate function is again utilized to execute anonymous function to search for the suitable links to click and navigate to.

Suitable links must not have following properties:

- Attributes mailto or tel as they indicate links are used for sending email or calling with a phone.
- Attribute target as it is usually used to open new tabs in the browser and that could confuse the crawler.
- SPA links that have different route than potentially specified in the configuration.
- Anchor links with SPA as they use the same symbol.
- Links should not have protocol defined unless the web address crawled is part of the link's href attribute value.
- Link should not be already found during the current crawling.

As the crawler crawls through the subpages of the website it must wait until the page has been loaded and possibly wait until JavaScript of the page has finished executing,

which is especially important in the case of many SPA webpages. Without this wait, the web page might not have properly loaded and the DOM that is retrieved from it might be incomplete and cause lot of false negative results during verification. As the crawler clicks on the links it has found, it must wait until the new content has been loaded or JavaScript has been executed.

```
await page.evaluate((loadReadyFlag): void => {
  let counter = new Date().getTime();
  const resetCounter = (): void => {
    counter = new Date().getTime();
  }
  const observer = new MutationObserver(resetCounter);
  observer.observe(document.body,
    {attributes: true, childList: true, subtree: true});
  setInterval(() : void => {
    if (new Date().getTime() - counter > 500) {
      const span = document.createElement('span');
      span.id = loadReadyFlag;
      document.body.appendChild(span);
    }
  }, 50);
}, LOAD_READY_FLAG);
```

```

await page.waitForSelector(`#${LOAD_READY_FLAG}`, {
  timeout: this.timeConfig.loadWait});
;

```

Listing 11. Waiting until the page has loaded before crawling it.

The crawler is utilizing experimental technology called MutationObserver [37] in Listing 11 to wait until no more changes to the DOM can be detected. Counter is used to detect if enough idle time has passed to mark the page loading ready by creating a span element with a specified id attribute. If observer detects any changes in the DOM it will reset the counter. Puppeteer is instructed to wait until css selector finds the span with a specified id attribute or until the configured time for waiting loading has passed, in which case an error is thrown by Puppeteer. In case of an error there is a fallback and Puppeteer is instructed to wait for 500 milliseconds instead for the page to load.

```

for (const item of items) {
  try {
    await page.waitForSelector(`[href="${item}"]`);
    await page.evaluate((element): void => {
      document.querySelector(element).scrollIntoView();
      return;
    }, `[href="${item}"]`);
    await page.waitFor(this.timeConfig.wait);
    await page.click(`[href="${item}"]`);
    await page.waitFor(this.timeConfig.wait);
    await this.waitForPageLoad(page);
    try {
      const url = await page.evaluate((): string =>
        window.location.href);
      await this.crawlInternal(page, url, depth + 1);
    } catch (e) {
      logger.error(e);
    }
  } catch (e) {
    logger.error(`Could not click the link ${item} on ${path}.
      Error: ${e}`);
  }
  await page.waitFor(this.timeConfig.wait);
  await page.goto(path, {waitUntil: 'networkidle2'});
  await this.waitForPageLoad(page);
}

```

Listing 12. Crawling through the links found on the web page.

In Listing 12, the crawler is going through all the eligible links it has found while scraping the current web page. As crawlInternal method is recursive function, it is all that is needed to go through the links in web page. If error is caught while crawling through the subpages, it is logged into the console but not otherwise allowed to interrupt the program. Code for the method waitForPageLoad is shown in Listing 11, and here it is

used two occasions. The first time the method is used to ensure that subpage has been loaded after clicking the link. The second time the method is used to ensure that the page where the link to subpage is loaded properly before clicking the link for the next subpage in the loop. The whole code listing for the crawler object can be found in the Appendix 1.

```
const makeTestDataFiles = (testData: TestData): void => {
  createRootFolderIfItDoesNotExist();
  const name = testData.name ? testData.name : testData.siteTested;
  const siteFolder = createSiteFolderIfItDoesNotExist(name);
  fs.writeFileSync(`${siteFolder}/site`, JSON.stringify(
    testData.pagesAndDoms), {encoding: 'utf8'});
  logger.info(`Crawling results written to ${siteFolder}/site`);
}
```

Listing 13. Storing the crawled website to filesystem.

After the test tool has made a crawl of the web page for recording action, the results of the scraping are stored into the filesystem for later verification purposes as shown in Listing 13. The result of the latest crawl is stored inside root test folder and into a folder named after the name specified or the website crawled.

5.4 Verifying

In order to verify that the website is unchanged from previous run, tool has to make a new crawl on the website. Afterwards the results can be verified against the previously stored run.

```
const verifyTestData = async (testData: TestData,
  config: Config): Promise<void> => {
  reportBuilder = new ReportBuilder();
  const storedResults: PageAndDom[] = getWebsitePagesAndDoms(
    testData.name ? testData.name : testData.siteTested);
  for (const page of storedResults) {
    await verifyPageAndDom(page, testData.pagesAndDoms, config, true);
  }
  const pageNotFoundInOriginal: PageAndDom[] = testData.pagesAndDoms
    .filter((siteAndDom: SiteAndDom): boolean => {
      for (const site of storedResults) {
        if (site.site === siteAndDom.site) {
          return false;
        }
      }
      return true;
    });
  for (const page of pageNotFoundInOriginal) {
```

```

    await verifySiteAndDom(page, storedResults, config);
  }
  reportBuilder.buildReport();
  logger.info('End of test');
}

```

Listing 14. Verifying the tested website against previous recording.

After the crawling has completed, results of the crawl are passed into the function listed above. Function will fetch the previously stored results and compare the output by looping contents of both test runs. Both test runs must be verified against each other, as there could be missing subpages from the new test run or new test run has more subpages. There is an additional html report built after the verification is done.

There are four different scenarios for verification not passing:

1. Subpage existed in the original crawling but not in the current test run
2. Subpage existed in the new test run but not in the original crawling.
3. DOMs of the subpages are not matching.
4. Screenshots of the subpages are not matching.

```

const verifyPageAndDom = async (pageAndDom: PageAndDom, pages: PageAndDom[], config: Config, original: boolean = false): Promise<void> => {
  ...
  let pageFound: PageAndDom = undefined;
  let passes = false;
  let screenShotsAreSame = false;
  for (const page of pages) {
    if (page .dom == pageAndDom.dom && page.page == pageAndDom.page) {
      screenShotsAreSame = await verifyScreenshots(pageAndDom, page);
      passes = screenShotsAreSame;
      pageFound = page;
    } else if (page.page == pageAndDom.page) {
      pageFound = site;
      screenShotsAreSame = await verifyScreenshots(pageAndDom, page)
    }
  }
  ...
}

```

Listing 15. Checking that web page is found and DOM and screenshot matches

Checking that web page is found between the two different crawls is rather straightforward as the Listing 15 demonstrates. If DOM and screenshot are also matching between the two instances of web page, then the web page is considered to have successfully passed the regression test. If DOM between the two instances of the web page are not matching then checking and reporting what is not matching is a bit more challenging.

```

const findDifference = (dom1: string, dom2: string,
  selectorPrefix = '>'): Level[] => {
  let difference: Level[] = [];
  const nodes1 = convertToNodeArray(dom1);
  const nodes2 = convertToNodeArray(dom2);
  nodes1.forEach((node: string, index: number): void => {
    if (index > nodes2.length) {
      difference.push({level: 'path',
        msg: getSelector(selectorPrefix, node)});
      difference.push({level: 'error', msg: 'NODE WAS NOT FOUND'});
      difference.push({level: 'old', msg: `${node}

    `});
    } else {
      const nodeInSecondList = nodes2[index];
      if (nodeInSecondList !== node) {
        if (hasChildren(node) && hasChildren(nodeInSecondList)) {
          const differenceFromChildren = findDifference(
            getChildren(node), getChildren(nodeInSecondList),
            `${getSelector(selectorPrefix, node)} >`);
          if (differenceFromChildren.length > 0) {
            difference = difference.concat(differenceFromChildren);
          } else {
            difference = difference
              .concat(getTextDifferenceFromNodes(selectorPrefix,
                node, nodeInSecondList));
          }
        } else {
          difference = difference
            .concat(getTextDifferenceFromNodes(selectorPrefix,
              node, nodeInSecondList));
        }
      }
    }
  });
  if (nodes2.length > nodes1.length) {
    difference.push({level: 'error', msg: 'FOLLOWING NODES ARE NEW'});
    for (let index = nodes1.length; index < nodes2.length; index++) {
      difference.push({level: 'new',
        msg: `${getSelector(selectorPrefix, nodes2[index])}
        ${nodes2[index]}

    `});
    }
  }
  return difference;
}

```

Listing 16. Finding what is different between two DOMs.

Function `findDifference` tries to recursively find the HTML elements that differ between two DOM trees. Both DOM trees are converted into two arrays consisting of node objects at the top level of DOM hierarchy utilizing JSDOM library. This recursive function checks that matching element is found in the same position and it checks the node children as well. Finally it also checks whether the second DOM tree it is comparing to have elements that are not part of the first DOM tree. Following flowchart (see figure 5) describes the decision process of the above function.

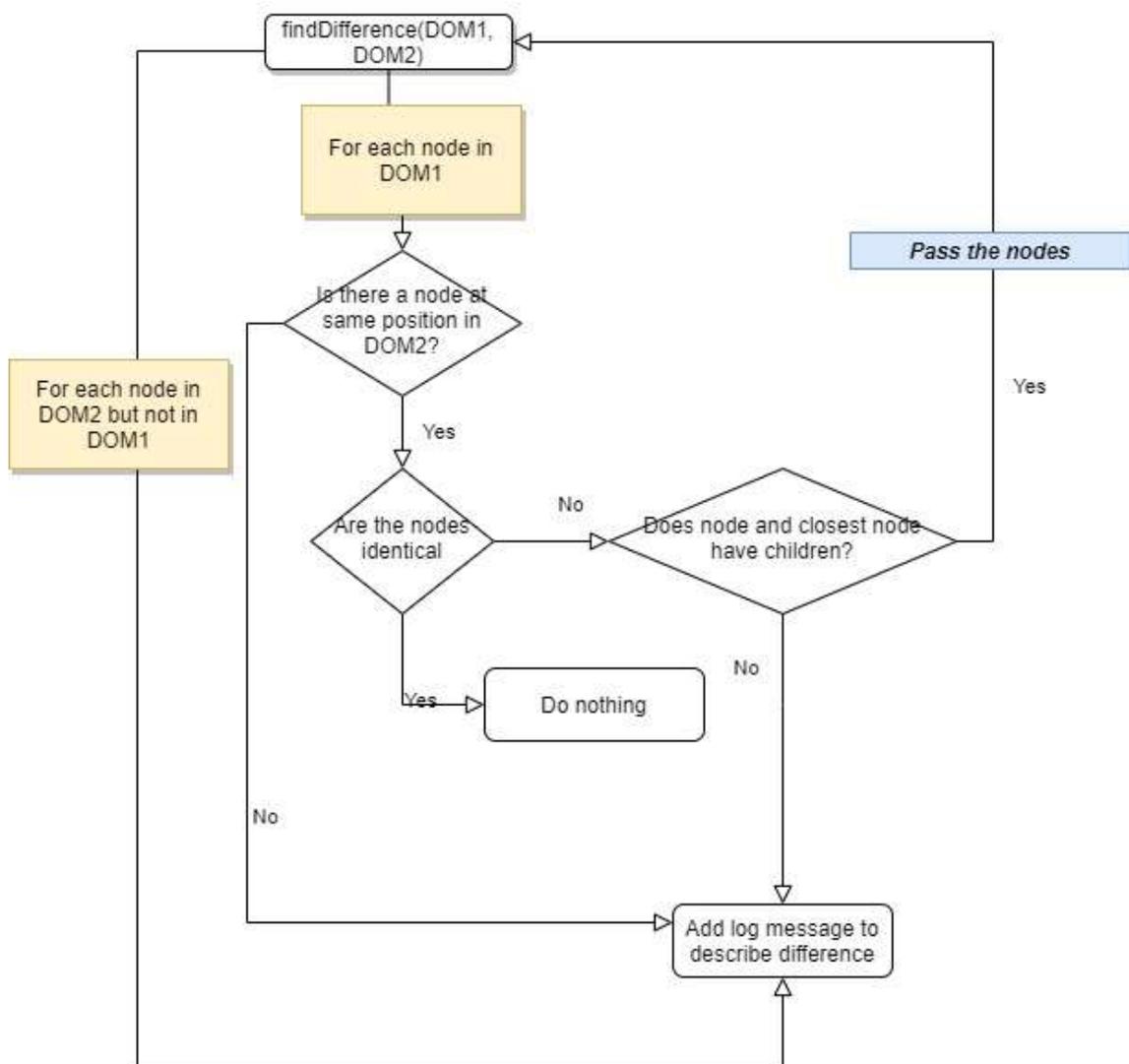


Figure 8. Flowchart describing decision process for `findDifference` function.

As shown in Figure 8 the `findDifference` function goes through each node in the first DOM at the top level. First it checks that if there is a node at the same position in the

second DOM. If there is no node in the same position in the second DOM as there was in the first DOM, log message is added to be reported to show the difference. If node is found in the second position, it checks whether the nodes are identical. In the case nodes are not identical it will recursively check node's children until all the nodes that differ are found, and their differences are added into the log message.

```
const getDom = (dom: string): HTMLCollection => {
  return new JSDOM(dom)
    .window.document.querySelector('body').children;
}

const getChildren = (node: string): string => {
  return [...getDom(node)[0].children]
    .map((element: Element): string => element.outerHTML)
    .reduce((previous: string, next: string): string =>
      previous.concat(next), '');
}

const convertToArray = (dom: string): string[] => {
  return [...getDom(dom)]
    .map((element: Element): string => element.outerHTML);
}
```

Listing 17. Three functions that are utilized when finding difference in DOMs.

In Listing 17 is shown three different functions that are used for checking the differences in two DOMs. Function `getDom` utilizes `JSDOM` to convert DOM in string form to `HTMLCollection`, which makes it easier to fetch topmost elements in DOM tree which is utilized by `convertToArray` function that maps nodes to an array of strings by their `outerHTML` property. Function `getChildren` also utilizes the `getDom` function but in this case children of the passed node are returned as array of strings and that is utilized by the recursive `findDifferences` function shown in Listing 16.

```
const equal = require('image-equal');
const load = require('image-pixels');

const verifyScreenshots = async (page1: PageAndDom, page2: PageAndDom)
: Promise<boolean> => {
  const screenshot1 = await load(page1.screenshotFile);
  const screenshot2 = await load(page2.screenshotFile);
  if(!equal(screenshot1, screenshot2)) {
    logger.error('Screenshots of the sites were not matching');
    return false;
  }
  return true;
}
```

Listing 18. Verifying the screenshots of the pages.

Comparison which determines whether the screenshots are same or not utilizes NodeJS library called image-equal [38], which compares images pixel by pixel. Comparing screenshots of two instances of the web page ensures finding regression in the visual appearance. Even small change in styles can make drastically different web pages, which might cause undesired side effects. In Listing 18 is shown the function that makes comparison between two screenshots utilizing image-equal library.

```

risto@Lightning: ~/git/ttl/testmaker
File Edit View Search Terminal Help
(base) risto@Lightning:~/git/ttl/testmaker$ clear

(base) risto@Lightning:~/git/ttl/testmaker$ npm run verify

> testmaker@0.1.1 verify /home/risto/git/ttl/testmaker
> tsc && node -e 'require("./dist/index.js").main().verify()'

info: CONFIGURATION:
info: {"domain":"http://localhost:5005/","name":"default","needsLogin":false,"timeConfig":{"wait":20,"loadWait":1000},"ignoredContent":[]}
info: START of crawling
info: Crawling page http://localhost:5005/ at depth 0
info: Crawling page http://localhost:5005/about.html at depth 1
info: Crawling page http://localhost:5005/cookies.html at depth 1
info: Found 3 sites
info: END of crawling
info: Verifying original site http://localhost:5005/
error: Screenshots of the sites were not matching
info: RUNNING ACCESSIBILITY TESTS ON: http://localhost:5005/
error: Document Title did not pass.
error: HTML Has Language did not pass.
error: Font Size did not pass.
error: Tap Targets did not pass.
error: Stored site http://localhost:5005/ did not pass the test
error: Site was found but dom tree was not matching. Difference:
path: > a
old: was <a href="cookies.html">Cookies</a>
new: now <a href="about.html">About</a>

error: FOLLOWING NODES ARE NEW
new: > a
    <a href="cookies.html">Cookies</a>

info: Verifying original site http://localhost:5005/cookies.html
error: Screenshots of the sites were not matching
info: RUNNING ACCESSIBILITY TESTS ON: http://localhost:5005/cookies.html
error: Document Title did not pass.
error: HTML Has Language did not pass.
error: Font Size did not pass.
error: Tap Targets did not pass.
error: Stored site http://localhost:5005/cookies.html did not pass the test
error: Site was found but dom tree was not matching. Difference:
path: > h1
old: was <h1>Cookies and Milk!</h1>
new: now <h1>Cookies and Milk</h1>

error: FOLLOWING NODES ARE NEW
new: > a
    <a href="about.html">About</a>

info: Verifying site http://localhost:5005/about.html
info: RUNNING ACCESSIBILITY TESTS ON: http://localhost:5005/about.html
error: Document Title did not pass.
error: HTML Has Language did not pass.
error: Font Size did not pass.
error: Tap Targets did not pass.
error: Site http://localhost:5005/about.html did not pass the test
error: Site does not exist in records
info: End of test
(base) risto@Lightning:~/git/ttl/testmaker$ █

```

Figure 9. Output of the verification script.

In Figure 9 is shown the output of the verification process. First configuration used is logged into console and then the crawling starts. After the crawling is completed the script will start verifying the different web pages found by the crawling process comparing the results to previously recorded results. The verification script will output both the path to the node and what has changed. The verification script will also run some ac-

cessibility audits and output also their results but that topic will be covered in the following section.

5.5 Accessibility

There exists three ways to view accessibility audits on the website with the testing tool. One way is to run verification for the website which outputs the failed audits to the console. Verification script also creates a HTML report for the tested website that contains the failed audits per page and can simply be opened with a browser. The last way is to run accessibility test suite for the configuration which requires that the website has been crawled previously so all the web pages are found for the accessibility test suite to audit.

Accessibility test suite utilizes Jest test framework to run and verify the audits, puppeteer to launch the browser instance and lastly Lighthouse CLI to run the audits on the browser. Lighthouse CLI allows running tests for other categories besides accessibility such as performance but those were not implemented in POC.

```
export async function audit(url: string,
  logLevel: string = 'info'): Promise<RunnerResult> {
  const report = await lighthouse(url, {
    port: REMOTE_DEBUGGING_PORT,
    output: 'json',
    logLevel,
  }, config);
  return report;
}

export async function getResult(report: RunnerResult,
  property: string): Promise<string> {
  const propertyType = new Map();
  for (const rule of accessibilityRules) {
    propertyType.set(
      rule.name, await report.lhr.audits[rule.lighthouseRule].score);
  }

  const score = new Map()
    .set(0, 'Fail')
    .set(1, 'Pass')
    .set(undefined, 'Pass')
    .set(null, 'Pass');

  const result = await score.get(propertyType.get(property));

  return result;
}
```

Listing 19. Functions for auditing the web page and fetching the results.

In Listing 19 is shown the audit function which executes asynchronous lighthouse command passing to it the same port that is used for logging into the website. Desired output is in json format so it is programmatically easier to parse and utilize in the tests. Also given to lighthouse function is the logging level which defaults to logging level info and lighthouse configurations which are retrieved from a file named lighthouse-config.json. Function getResult is utilized to retrieve a result of single audit by a given property. Result of the audit property is mapped into a string with value pass or fail, depending on the score.

```
const config = getConfig(argv['configName'] as string);

let browser: Browser;

beforeAll(async (): Promise<void> => {
  jest.setTimeout(100000);
  browser = await launchBrowser(config);
});

afterAll(async (): Promise<void> => {
  await browser.close();
});

logger.info('CONFIGURATION:');
logger.info(JSON.stringify(config));
const storedResults: PageAndDom[] = getWebsitePagesAndDoms(config.name
  ? config.name
  : config.website);
const pages = storedResults.map((page: PageAndDom): string =>
  page.page);
for (const page of pages) {
  runAccessibilityOnPage(page);
}
logger.info('End of test');
```

Listing 20. Code for running the accessibility tests.

In Listing 20 is shown the code used for running the accessibility tests. First the configuration name that is passed as a command line parameter is used to fetch the configuration. Functions `beforeAll` and `afterAll` are hooks provided by Jest and all the tests are executed after `beforeAll` function is executed and before `afterAll` function is executed respectively. Function `beforeAll` is used to launch the browser instance used by lighthouse and make a login to the website if required by configuration. Function `afterAll` is used to close the browser after all the audits and tests have been run. Finally the crawled website is retrieved from the filesystem using the configuration and the `runMetricsOnSite` function is executed for every web page.

```
const runAccessibilityOnPage = async function(page: string):
  Promise<void> {
  let report: RunnerResult;

  describe('Accessibility', (): void => {

    beforeEach(async (): Promise<void> => {
      jest.setTimeout(100000);
      report = report || await audit(page);
    });

    it(`${page}: accessibility score`, async (): Promise<void> => {
      const accessibilityScore = await getScore(report,
        'accessibility');
      expect(accessibilityScore).toBeGreaterThanOrEqual(100);
    });

    it(`${page}: passes accessibility level A checks`,
      async(): Promise<void> => {
      const rules = getRulesByAccessibilityLevel(
        AccessibilityLevel.A);
      const rulesNotPassing: AccessibilityRule[] =
        await getNotPassingRules(rules, report);
      expect(checkAllRulesPassed(rulesNotPassing)).toBeTruthy();
    });

    ...

  });
};
```

Listing 21. Accessibility tests.

In Listing 21 some of the accessibility tests are shown. Function `beforeEach` is another Jest hook function that is run before each test case and it sets Lighthouse report to a variable. Here are two test cases shown in Listing 21. The first test case expects that accessibility score for the web page calculated by Lighthouse is at least 100 percent. The second test case checks that accessibility rules of Lighthouse, that are defined here to belong to conformance level A of WCAG 2.1, all pass on the web page.

```
enum AccessibilityLevel {
  A, AA, AAA, NOT_DETERMINED
}

interface AccessibilityRule {
  lighthouseRule: string;
  description: string;
  name: string;
  accessibilityLevel: AccessibilityLevel;
  url: string;
}

const accessibilityRules: AccessibilityRule[] = [
  {
    lighthouseRule: 'color-contrast',
    description:
      'The visual presentation of text and images of text has a
      contrast ratio of at least 4.5:1',
    name: 'Contrast',
    accessibilityLevel: AccessibilityLevel.AA,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#contrast-minimum'
  },
  {
    lighthouseRule: 'image-alt',
    description:
      'All non-text content that is presented to the user
      has a text alternative that serves the equivalent purpose,',
    name: 'Image Alt Text',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#non-text-content'
  },
],
```

Listing 22. Accessibility rules defined in the POC.

In Listing 22 is shown some accessibility rules and associated definitions that are used for POC. First there is an enumeration `AccessibilityLevel` which is used to determine if the rule belong to one of the conformation levels of WCAG 2.1. Accessibility rule has five variables: the first is the name of the Lighthouse audit that rule refers to, the second variable provides description for the rule, the third variable is the name of the rule, the fourth variable is the conformance level rule belongs to and the last variable is a web address to the rule in WCAG 2.1 Quick Reference guide. Complete list of accessibility rules used for POC can be found in Appendix 2.

```

risto@Lightning: ~/git/ttl/testmaker
File Edit View Search Terminal Help
status Auditing: Structured data is valid +0ms
status Generating results... +0ms
FAIL src/accessibility/accesslibbiter.spec.ts (6.49s)
Metrics
  X http://localhost:5005/: accessibility score (2042ms)
  X http://localhost:5005/: best practices score
  X http://localhost:5005/: SEO score
  X http://localhost:5005/: passes accessibility level A checks (2ms)
  X http://localhost:5005/: passes accessibility level AA checks
  X http://localhost:5005/: passes accessibility level AAA checks
  X http://localhost:5005/: passes other accessibility checks (1ms)
  X http://localhost:5005/cookies.html: accessibility score (1880ms)
  X http://localhost:5005/cookies.html: best practices score
  X http://localhost:5005/cookies.html: SEO score
  X http://localhost:5005/cookies.html: passes accessibility level A checks (3ms)
  X http://localhost:5005/cookies.html: passes accessibility level AA checks
  X http://localhost:5005/cookies.html: passes accessibility level AAA checks (1ms)
  X http://localhost:5005/cookies.html: passes other accessibility checks (1ms)

● Metrics > http://localhost:5005/: accessibility score
expect(received).toBeGreaterThanOrEqual(expected)

Expected: >= 100
Received: 60

   61 |     it(`${site}: accessibility score`, async (): Promise<void> => {
   62 |       const accessibilityScore = await getScore(report, 'accessibility');
>  63 |       expect(accessibilityScore).toBeGreaterThanOrEqual(100);
      |                                     ^
   64 |     });
   65 |
   66 |     it(`${site}: best practices score`, async (): Promise<void> => {
      at Object.<anonymous> (src/accessibility/accesslibbiter.spec.ts:63:34)

● Metrics > http://localhost:5005/: SEO score
expect(received).toBeGreaterThanOrEqual(expected)

Expected: >= 75
Received: 50

   71 |     it(`${site}: SEO score`, async (): Promise<void> => {
   72 |       const SEOScore = await getScore(report, 'seo');
>  73 |       expect(SEOScore).toBeGreaterThanOrEqual(75);
      |                                     ^
   74 |     });
   75 |
   76 |     it(`${site}: passes accessibility level A checks`, async(): Promise<void> => {
      at Object.<anonymous> (src/accessibility/accesslibbiter.spec.ts:73:24)

● Metrics > http://localhost:5005/: passes accessibility level A checks
expect(received).toBeTruthy()

Received: false

   77 |     const rules = getRulesByAccessibilityLevel(AccessibilityLevel.A);
   78 |     const rulesNotPassing: AccessibilityRule[] = await getNotPassingRules(rules, report);
>  79 |     expect(checkAllRulesPassed(rulesNotPassing)).toBeTruthy();
      |                                     ^
   80 |   });
   81 |
   82 |   it(`${site}: passes accessibility level AA checks`, async(): Promise<void> => {
      at Object.<anonymous> (src/accessibility/accesslibbiter.spec.ts:79:52)

● Metrics > http://localhost:5005/: passes accessibility level AAA checks
expect(received).toBeTruthy()

```

Figure 10. Output of the accessibility script.

Figure 10 shows output of the accessibility tests. In the screenshot you can see the tests that failed and passed for each web page first and then Jest outputs the failures in greater detail.

5.6 Testing

Testing of the tool is handled by running different versions of a website. Test folder of the tool contains JavaScript file which creates a HTML files and launches a server on localhost serving the generated static content. This test utility provides different versions of the website and utilizing a script to launch different scenarios of the website provides a quick manual process to test the actual tool's website crawling, regression testing and accessibility testing capabilities along with the reporting capabilities. Different workflows for testing the tool are listed in Table 3.

Table 3: Different possible workflows for testing the tool

#	Test Scenario	Steps
1	Crawling a website	<ol style="list-style-type: none"> 1. Serve version one of the test website by executing: npm run scenario:1 2. Execute: npm run crawl 3. Check that TESTDATA folder contains the scraped data and screenshots for default configuration
2	Checking that regression tests pass	<ol style="list-style-type: none"> 1. Do the steps in Test Scenario 1. 2. Execute: npm run verify. 3. Check that there are no regression errors outputted into console or in HTML report
3	Checking that regression tests	<ol style="list-style-type: none"> 1. Do the steps in Test Scenario 1. 2. Stop the server serving version one of the website

	fail	<ol style="list-style-type: none"> 3. Serve version two of the test website by executing: npm run scenario:2 4. Execute: npm run verify 5. Check that there are regression errors outputted into console or in HTML report
4	Running accessibility tests	<ol style="list-style-type: none"> 1. Do the steps in Test Scenario 1. 2. Execute: npm run accessibility

The tool provides a default configuration which contains the local web address for the served test website. How different versions of the website are served is shown in Listing 23.

```
const SCENARIOS = [require('./scenario1'), require('./scenario2'), require('./scenario3'), require('./scenario4')];

const TEST_FOLDER = 'test/testsite/';

const addHTML = (dom) => {
  if (dom.includes('<head>')) {
    return `<html>${dom}</html>`;
  }
  return `<html><head></head>${dom}</html>`;
}

const SCENARIO = SCENARIOS[argv.scenario];
if (fs.existsSync(TEST_FOLDER)) {
  fs.rmdirSync(TEST_FOLDER, {recursive: true});
}
fs.mkdirSync(TEST_FOLDER);

fs.writeFileSync(`${TEST_FOLDER}index.html`, addHTML(SCENARIO.HOME), {
  encoding: 'utf8'});
if (SCENARIO.ABOUT) {
  fs.writeFileSync(`${TEST_FOLDER}about.html`, addHTML(SCENARIO.ABOUT), {
    encoding: 'utf8'});
}

if (SCENARIO.COOKIES) {
  fs.writeFileSync(`${TEST_FOLDER}cookies.html`, addHTML(SCENARIO.COOKIES), {
    encoding: 'utf8'});
}

connect().use(serveStatic('test/testsite')).listen(5005, function() {
  console.log('Server running on 5005...');
});
```

Listing 23. Code for serving the test website

Different versions of the website are stored in their respective JavaScript files as shown in the Listing 23. Runtime argument named scenario instructs which version of the website to generate into the filesystem. Lastly the Node libraries `serve-static` [39] and `connect` [40] are used to serve the static website on localhost post 5005.

```
const HOME = `
  <body>
    <h1>Test Title</h1>
    <p>Lorem Ipsum</p>
    <a href="cookies.html">Cookies</a>
  </body>
`;

const COOKIES = `
  <body>
    <h1>Cookies and Milk!</h1>
    <p>Yum!</p>
    <a href="/">Home</a>
  </body>
`;

module.exports = {HOME, COOKIES};
```

Listing 24. Third version of the test website

In Listing 24 is shown the third version of the website that contains only two web pages. Adding new versions of the test website requires following: a new JavaScript file that has the DOM of the pages that are to be served, adding that version to the array of scenarios and instructing node execute the JavaScript file that creates the test website and runs the server with appropriate scenario command line argument.

5.7 Reports

Verification script in the tool also generates a HTML report. The generated report shows following things for each page that regression test was run on: information whether the site passed regression test, differences in DOM between stored original run and the latest run and screenshots from both runs. The generated report also shows following items for accessibility of the each web page: conformation level to WCAG 2.1, accessibility score calculated by the Lighthouse and all the audits that failed.

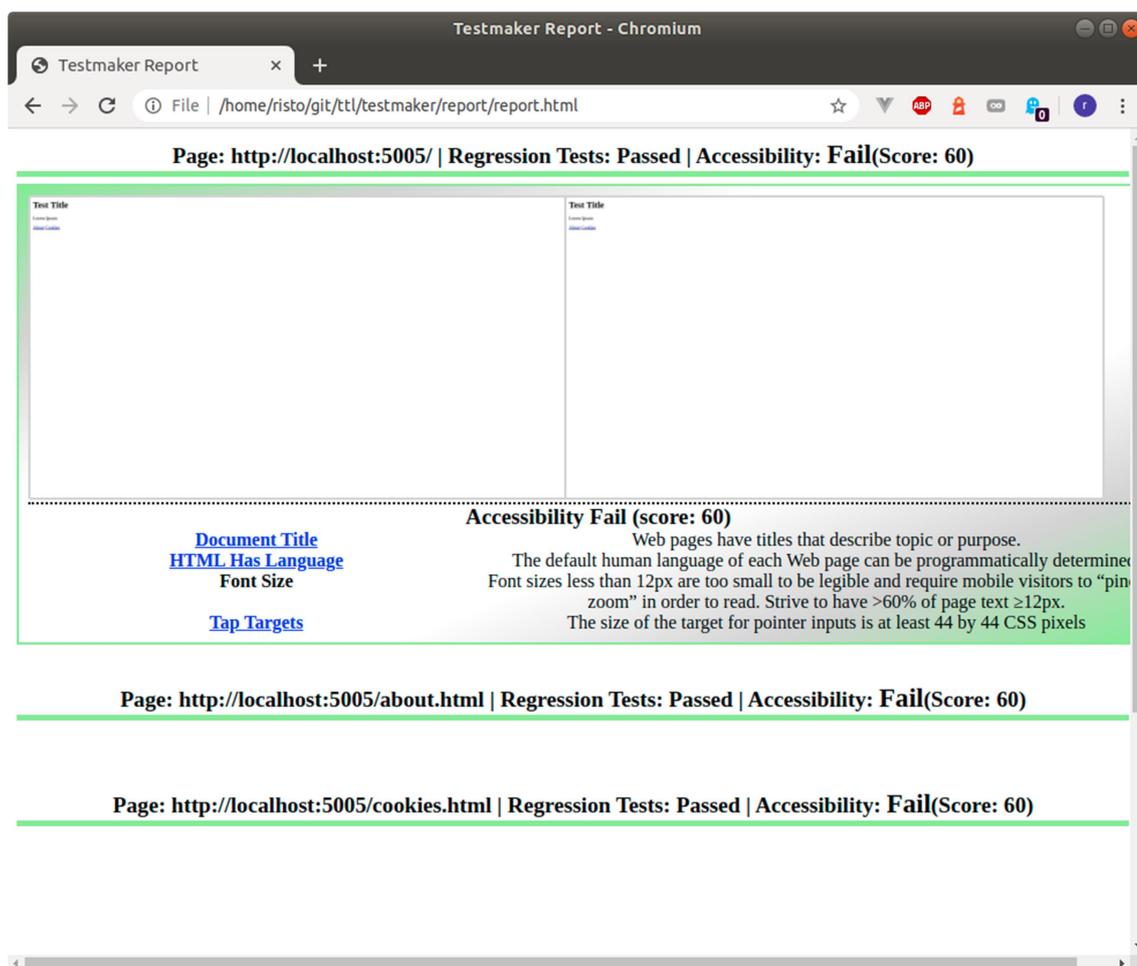


Figure 11. Generated HTML Report

In Figure 11 is shown a generated HTML report for a website that contained three different web pages. In the report each web page passed the regression tests but did not pass the accessibility audits, that is not even the conformation level A of WCAG 2.1 was reached. Each page element can be toggled to show or hide the details of the verification. In Figure 11 screenshots of the original and the latest run are shown side by side. Accessibility section on opened page element also lists all the checks that did not pass during the audit providing for each item a name and a description. Also if there is link provided for accessibility item, the name acts as a link.

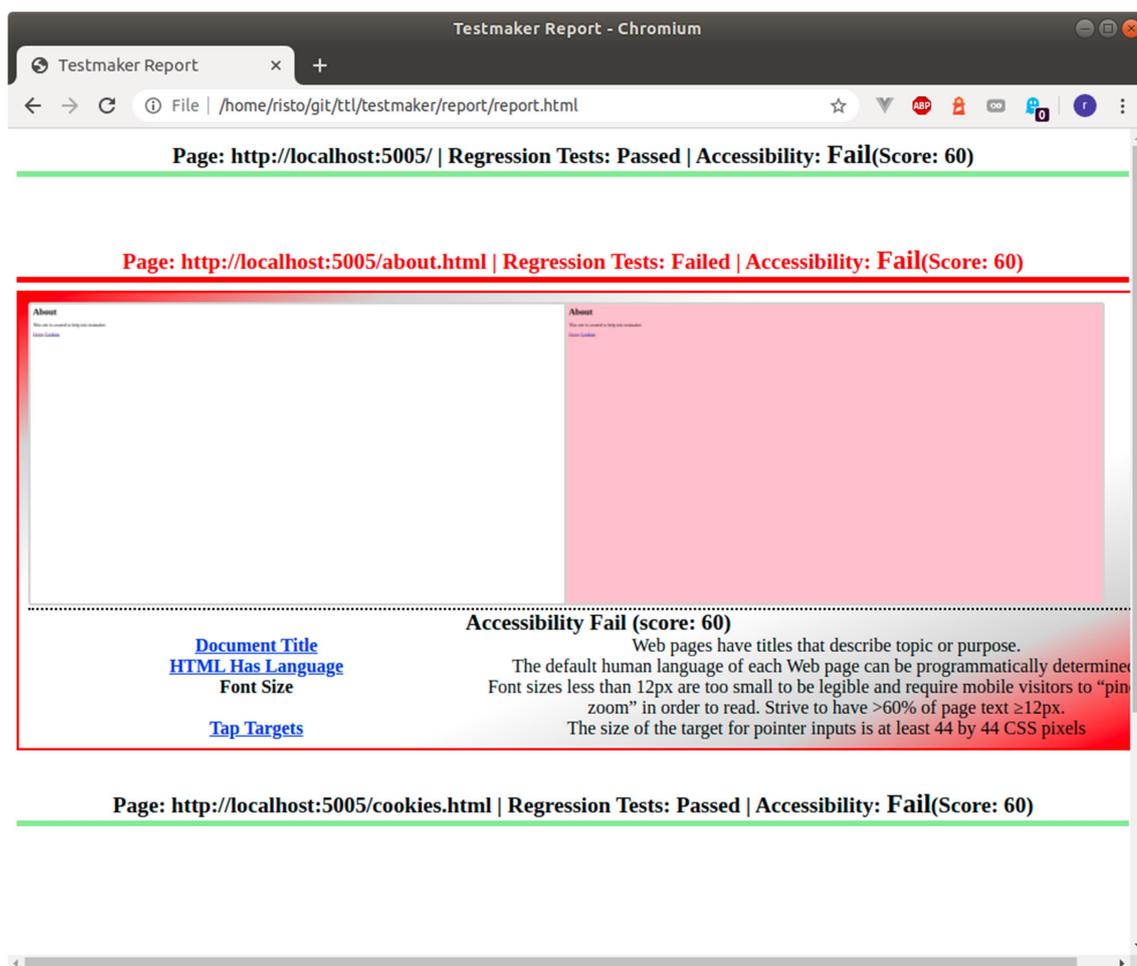


Figure 12. Generated HTML Report where regression test did not pass

A generated HTML report where regression test did not pass is shown in Figure 12. The reason why the regression test did not pass was that screenshots were different: screenshot on left has a white background while screenshot on right has a pink background.

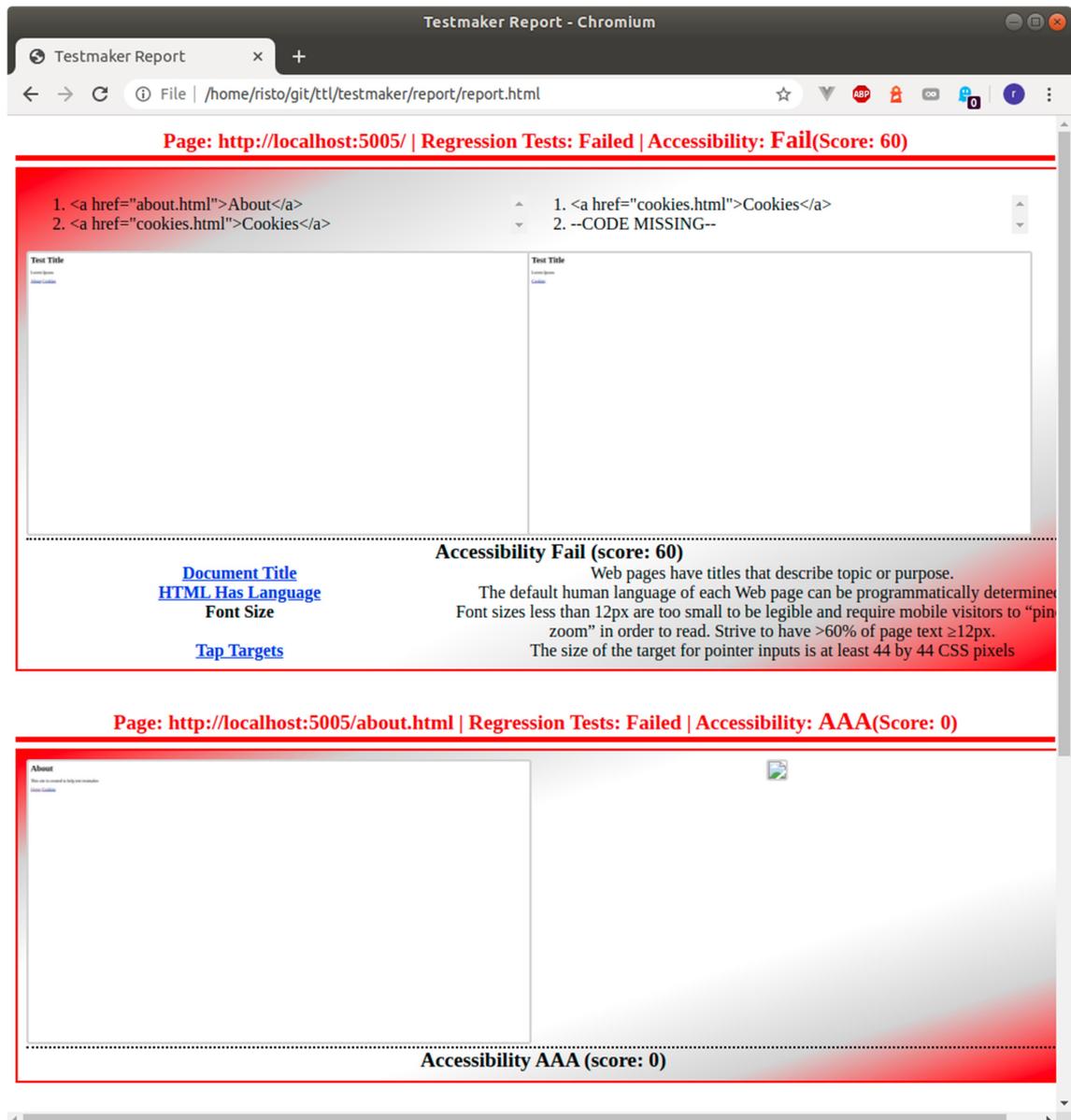


Figure 13. Another HTML Report where regression tests did not pass.

In Figure 13 is another example of a report where regression tests did not pass. Code differences between two instances of the home web page of the tested site were different. Since the link to about page was missing on the instance on the right side in Figure 13, it changed the order of elements and that is the reason both links are shown to be different between the instances. As about page was missing in the verification run the screenshot of about page on the report is also missing; no code differences are shown for the missing site.

Rest of this section goes over sections of code describing how the report is generated. First interfaces and class for building reports are listed and explained. After that is shown how accessibility part of the HTML report is generated. And lastly is shown how verification script utilizes previous parts to generate HTML report.

```
interface ReportHalf {
  mismatches: string[];
  screenshot: string;
}

interface ReportSection {
  page: string;
  original: ReportHalf;
  current: ReportHalf;
  screenshotsMatch: boolean;
  accessibility: AccessibilityReport;
}
```

Listing 25. Interfaces used for building report

Listing 25 shows two interfaces used for building HTML report. As regression testing is comparing two different instances of the same page, instance specific code is stored into their own halves called ReportHalf, which contains array of differences in DOM between two instances of the web page and screenshots of a web page made from the instance. These are stored into ReportSection as original and current values, original being the previously crawled and stored webpage and current the webpage from the recent crawling. In addition to those two values, ReportSection also contains the page address, information whether the screenshots were matching and lastly and accessibility report object.

```
class ReportBuilder {
  private sections: ReportSection[] = [];

  public createSection(page: string,
    screenshotOriginal: string, screenshotCurrent: string,
    screenshotsMatch: boolean,
    accessibility: AccessibilityReport): void {
    if (this.sections.some((section): boolean =>
      section.page === page )) {
      return;
    }
    const original: ReportHalf = {screenshot: screenshotOriginal,
      mismatches: []};
    const current: ReportHalf = {screenshot: screenshotCurrent,
      mismatches: []};
    this.sections.push({page, original, current, screenshotsMatch,
      accessibility});
  }
}
```

```

public addMismatch(page: string, original: string,
  current: string): void {
  const convertedOriginal = convertDom(original);
  const convertedCurrent = convertDom(current);
  const section = this.sections.find((section): boolean =>
    section.page === page);
  section.current.mismatches.push(convertedCurrent);
  section.original.mismatches.push(convertedOriginal);
}

public buildReport(): void {
  let content = ''
  ...
  // Sections are converted to DOM string here
  ...
  const template = fs.readFileSync('./report/template.html',
    { encoding: 'utf8'});
  const report = template.replace('{{}}', content);
  fs.writeFileSync('./report/report.html', report, {encoding: 'utf8'
});
}
}

```

Listing 26. ReportBuilder Class.

Class used for building report is shown in Listing 26. Builder class contains array of report sections, one section for each web page on the report. Builder class also has three public methods. Method for creating section can be used once per web page, function simply returns if web page is already contained in one of the sections. Method for adding differences in DOM between two instances of web page is called addMismatch, which will convert the DOM strings passed into it to escaped HTML so the tags can be displayed in the HTML report and then adds them into respective halves of report section. Last method in the Builder class is used to actually build the report, report sections are converted into HTML elements and they are stored into a HTML file utilizing an existing template.

```

<html>
<head>
  <title>Testmaker Report</title>
  <link rel="stylesheet" href="report.css" />
  <script src="./report.js" ></script>
  <meta charset="UTF-8">
</head>
<body onload="registerListeners()" >
  {{}}
</body>
</html>

```

Listing 27. Template used for generating HTML report.

In Listing 27 is the template used for generating HTML report. Basically only the brackets inside body element are replaced by the generated DOM from the report builder. Toggling page sections on the report is enabled by calling a function that adds required event listeners.

```
interface AccessibilityReport {
  page: string;
  score: number;
  levelReached: AccessibilityLevel;
  failingAudits: AccessibilityRule[];
}

const runAccessibilityTestsOnPage = async (page: string, config: Config): Promise<AccessibilityReport> => {
  logger.info(`RUNNING ACCESSIBILITY TESTS ON: ${page}`);
  const browser: Browser = await launchBrowser(config);
  const report: RunnerResult = await audit(page, 'silent');
  const accessibilityScore = await getScore(report, 'accessibility');
  const rulesNotPassing = await getNotPassingRules(accessibilityRules, report);
  await browser.close();
  return {
    failingAudits: rulesNotPassing,
    score: accessibilityScore,
    levelReached: getLevelReached(rulesNotPassing),
    page
  }
}
```

Listing 28. Interface for accessibility report and a function that generates the report.

In Listing 28 is shown an interface `AccessibilityReport` that contains following information: web page that accessibility report is run on, score calculated by Lighthouse, conformance level of WCAG 2.1 reached by the web page and array of any audits that did not pass. Function that generates accessibility report object also runs the Lighthouse audits on the web page. The function utilizes the `launchBrowser` function that launches the browser instance and makes a login attempt on the site if necessary. Conformance level reached is determined by the following logic:

- Conformance level A is reached if all accessibility rules related to it have passed.
- Conformance level AA is reached if all accessibility rules related to it and to conformance level A have passed.

- Conformance level AAA is reached if all accessibility rules related to it and to conformance levels A and AA have passed.

The function to run accessibility audits and to generate the accessibility report is called during verifying whether there is regression on the web page. Lighthouse audit is initiated with logging level silent so only the failed audits are outputted on the terminal.

```
const verifyPageAndDom = async (pageAndDom: PageAndDom,
  pages: PageAndDom[], config: Config,
  original: boolean = false): Promise<void> => {
  ...
  // Code that does initial verification and comparisons are left out
  ...
  const accessibilityReport: AccessibilityReport = await
    runAccessibilityTestsOnPage(pageAndDom. page, config);
  reportBuilder.createSection(pageAndDom. page,
    screenshotOriginal, screenShotCurrent,
    screenShotsAreSame, accessibilityReport);
  if (!passes) {
    ...
    storeDifferenceForReporter(pageAndDom, pageFound);
  }
  ...
  // Code that does rest of the verification is left out
}

const verifyTestData = async (testData: TestData, config: Config):
  Promise<void> => {
  reportBuilder = new ReportBuilder();
  ...
  // Here is the logic that calls verifyPageAndDom function for each
  // web page
  ...
  reportBuilder.buildReport();
  logger.info('End of test');
}
```

Listing 29. Parts of regression testing related to building the HTML report

Function that does the regression verification creates a ReportBuilder object as shown in Listing 29. The same function calls verifyPageAndDom function for each web page found during the crawling process. This function is used to check if there is any regression in the web page but it also adds the web page along with generated accessibility report to the builder in the form of report section. In case there is any regression found, the function also calls function named storeDifferenceForReporter, which is a recursive function that has logic as shown in Figure 13. Lastly verifyTestData function instructs the report builder to build the HTML report.

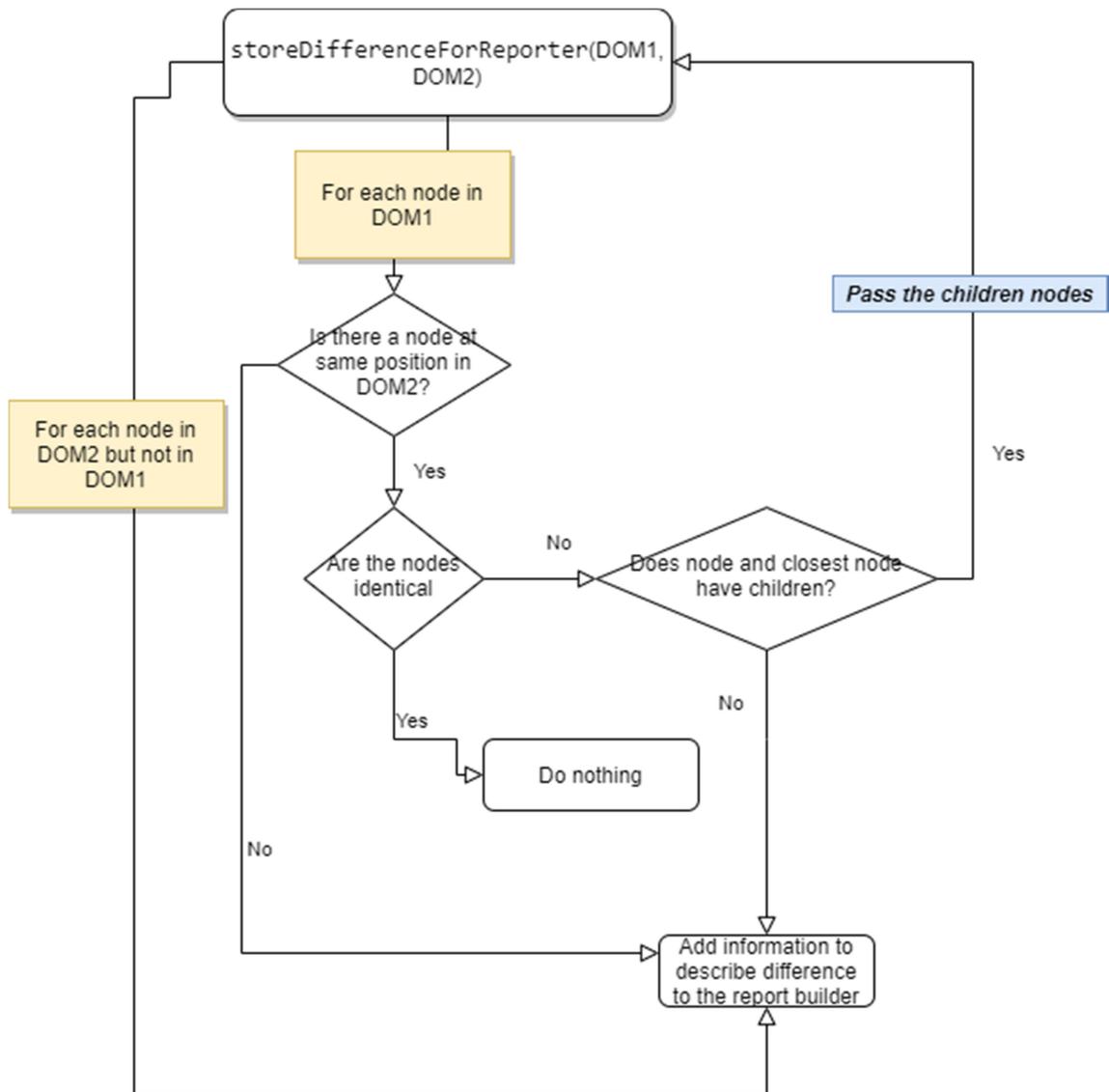


Figure 14. Logic to store differences between two DOM trees to the builder.

In Figure 14 is shown how the function `storeDifferenceForReporter` goes through each node in the first DOM at the top level. First it checks that if there is a node at the same position in the second DOM. If there is no node in the same position in the second DOM as there was in the first DOM, information that describes the difference is stored to the report builder. If node is found in the second position, it checks whether the nodes are identical. In the case nodes are not identical it will recursively check node's children until all the nodes that differ are found, and their differences are added into the report builder as well.

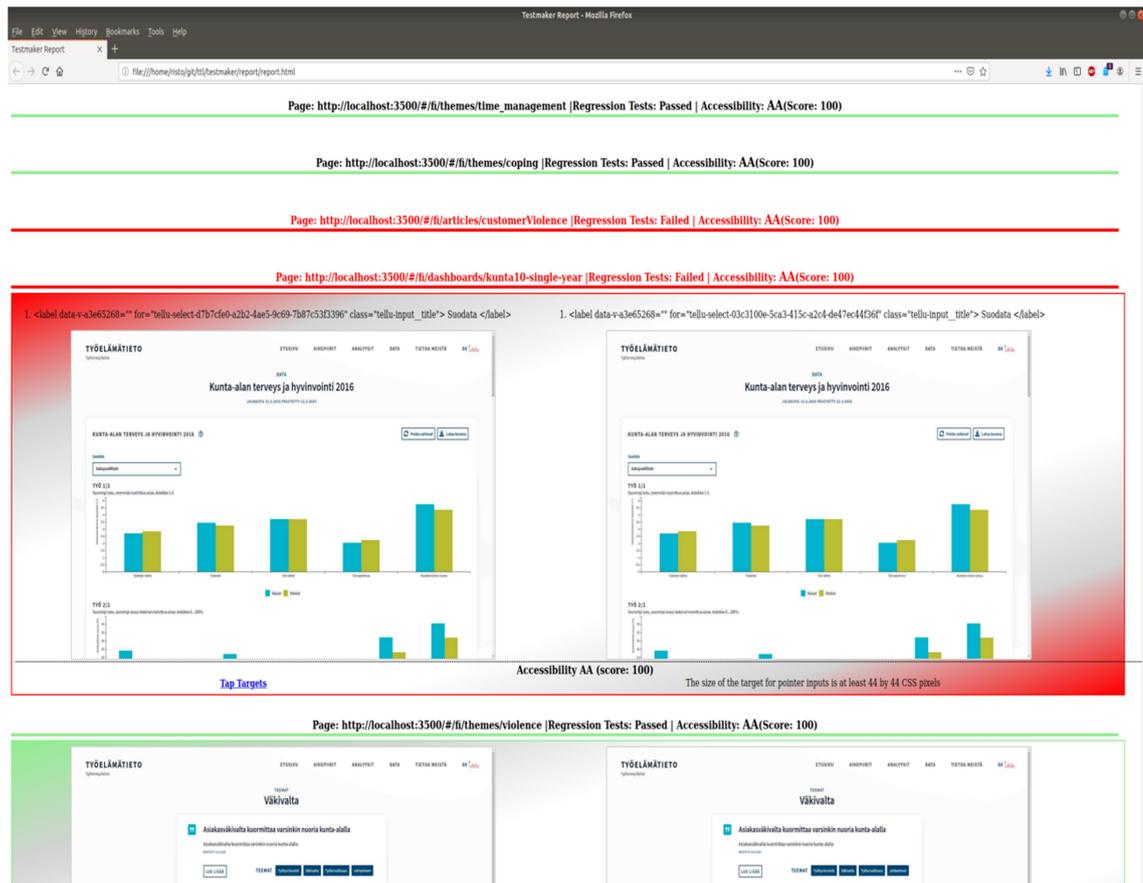


Figure 15. Report of Work-life data service.

A report for a Work-life data service is shown in Figure 15. The website has received a perfect accessibility score from Lighthouse audit and has reached conformance level AA of WCAG 2.1 for many web pages. However, it did not pass all the Lighthouse audits, one being the audit named tap targets and that particular audit is connected to conformance level AAA of WCAG 2.1. Interestingly, the audit did not have a distinguishable part in Lighthouse accessibility score. Also of note, is that the website has labels with attributes that can change on each page load on some of the web pages. That causes the regression tests to not pass on some of the web pages and the tool reports a false negative.

The rest of this chapter shows couple more examples of public websites this tool is run against for demonstration purposes. This is done to demonstrate that the testing tool can be used to test regression and accessibility of various websites.

Testmaker Report - Mozilla Firefox

Testmaker Report

file:///home/risto/git/tl/testmaker/report/report.html

Page: <https://metropolia.fi/fi/opiskelu-metropoliassa> | Regression Tests: Failed | Accessibility: Fail(Score: 80)

<p>1. <code><iframe allowtransparency="true" frameborder="0" allow="fullscreen; camera; microphone" id="giosg_chat_history" name="giosg_chat_history" src="https://5493.clients.giosgusercontent.com/cd/5493/viiln/?url=https%3A%2F%2Fmetropolia.fi%2Ffi%2Fopiskelu-metropoliassa%234c808631&dialog_id=8634"></iframe></code></p>	<p>1. <code><iframe allowtransparency="true" frameborder="0" allow="fullscreen; camera; microphone" id="giosg_chat_history" name="giosg_chat_history" src="https://5493.clients.giosgusercontent.com/cd/5493/fdq3p/?url=https%3A%2F%2Fmetropolia.fi%2Ffi%2Fopiskelu-metropoliassa%23d0d7bea0&dialog_id=8634"></iframe></code></p>
	
<p>Contrast Image Alt Text ARIA Required Children Link Has Discernible Name Tap Targets</p>	<p>Accessibility Fail (score: 80) The visual presentation of text and images of text has a contrast ratio of at least 4.5:1 All non-text content that is presented to the user has a text alternative that serves the equivalent purpose, Valid ARIA required children The purpose of each link can be determined from the link text alone or from the link text together with its programmatically determined link context, except where the purpose of link would be ambiguous to users in general. The size of the target for pointer inputs is at least 44 by 44 CSS pixels</p>

Page: <https://metropolia.fi/fi/opiskelu-metropoliassa#4c808631> | Regression Tests: Failed | Accessibility: Fail(Score: 80)

Figure 16. Report of Metropolia website.

A report for a Metropolia website is shown in Figure 16. The website utilizes a Content Management System (CMS) that seems to be the cause for the regression issue reported. This issue is a false negative because it seems to be caused by a generated id on runtime. Configuration option to ignore content can be utilized to prevent that particular false negative from reoccurring. The web page shown in Figure 15 has also not reached conformance level A of WCAG 2.1 since it has not passed all the accessibility audits that are labeled with conformance level A.

6 Conclusions

This section covers what the POC project achieved. This was done mostly by checking if all the specifications were actually implemented. Subsections describe what challenges were when implementing specifications of the POC and what could be improved in the future.

Table 4: Checking Implementation of Specifications of the Proof of Concept

Specification	Purpose	Implemented
Crawler	To find all the subpages of the website	Yes
Scraper	To record DOM on all the subpages	Yes
Captures Screenshots	Captures screenshots of the web pages in the website to be used for both verification and reporting	Yes
Storing the scraped content with screenshots	Scraped content with screenshots must be stored for later verification use	Yes
Verification	Verifies whether the website has changed	Yes
Reporting via command	Reports all the changes found during	Yes

line interface (CLI)	verification	
Visual report	Generates visual report listing all the changes found during verification	Yes by generating a HTML report
Accessibility checks	The tool tests the accessibility of webpage	Yes
Ability to log in	The tool can be used to crawl private webpages	Yes
Ability to ignore marked content	Some content like time changes all the time and should be ignored during verification	Yes but implemented instead an ability to ignore content during scraping
Configurable	The tool can be configured to run on different web pages requiring different information like log in credentials	Yes
Support static website	The tool can be run against a static website	Yes
Support SPA	The tool can be run against SPA	Yes

As shown in Table 4, POC managed to implement every specification. Only small exception is the implementation detail of specification for ignoring content. To be specific, POC does allow a user to ignore some content, but instead of during verification process the content is actually ignored already during the scraping process.

6.1 Challenges

There were three major challenges during implementation of the POC. The first challenge was how crawling and scraping of the website could be implemented. There did exist some examples in Internet on how to crawl and scrape the website but all of them were either instructed to run against a specific web page instead of trying to crawl through all the links or they could only work on static websites. Implementing the POC to recursively crawl a website was challenging and especially challenging was to find proper methods to wait for SPA site to load dynamically. Luckily experimental technology MutationObserver allowed recursive crawling and scraping to work reliably with websites that load dynamical content.

The second challenge, although a smaller one, was during regression testing. Particularly finding the differences recursively in two instances of the website was tricky to implement. Running the verification code successfully for a one website without errors in the verification process itself might not work on another website, and changes were required to the code. At one time JavaScript Runtime even crashed during verification process as it did not properly manage the DOM hierarchy but ended up in infinite loop instead.

The third challenge was related to accessibility. First there existed no tools that can provide complete confirmation that a website conforms to WCAG 2.1 specifications as many of the checks required to conform level A could not be automated reliably. Also mapping the accessibility audits of the Lighthouse to the conformance levels and specifications of the WCAG 2.1 proved challenging. For that reason many of the accessibility rules used in POC listed in Appendix 2 have no conformance level or have a link to WCAG 2.1 Quick Reference site.

6.2 Future Improvements

Some of the potential future improvements for the testing tool are about improving the reports and making crawling and scraping process faster and more reliable. The testing tool takes a considerable amount of time to process large websites, and this could potentially be improved with multithreading and optimizing the code.

The visual report generated by the tool could be improved to be even more readable and DOM comparisons could be enhanced. A nice feature would be to highlight what is different between two nodes and also a feature to show the difference between screenshots of the web page. Some user experience (UX) and UI designs could be utilized to enhance the generated visual report.

One useful change that would require a major overhaul would be to change the crawling and scraping process to work on other interactive elements besides links such as buttons and form controls. That change could potentially detect more changes that can cause regression. Ability to ignore content could also be implemented during verification process where user would be queried if he or she wants to ignore some dynamic content in the future regression tests for the website.

The testing tool should also be added into the CI pipeline of different projects, where it could potentially prevent builds that have regression or accessibility issues from being deployed to production for example. This might require some changes in the tool itself as well.

The testing tool also lacks unit tests that could verify its code is working as intended. Particularly the logic related to crawling and verification are important to be automatically tested but it can be also very useful to have tests for accessibility and report functionality of the tool as well.

Lighthouse CLI contains also audits for SEO, performance and best practices. Those audits could also be added into the testing tool with reasonable effort by adding results of those audits into the reports and making new test suites.

References

- 1 About Finnish Institute of Occupational Health. <https://www.ttl.fi/en/about-us/>. Accessed 23 Nov 2019.
- 2 Saavutettavuus. <https://vm.fi/saavutettavuusdirektiivi>. Accessed 26 Jan 2020.
- 3 DIRECTIVE (EU) 2016/2102 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL. <https://eur-lex.europa.eu/eli/dir/2016/2102/oj>. Accessed 1 Apr.2020.
- 4 Spring Boot. <https://spring.io/projects/spring-boot>. Accessed 30 Apr 2020.
- 5 Node.js. <https://nodejs.org/en/>. Accessed 19 Apr 2020.
- 6 Flyway by Redgate – Database Migrations Made Easy. <https://flywaydb.org/>. Accessed 30 Apr 2020.
- 7 Knex.js – A SQL Query Builder for JavaScript. <http://knexjs.org/>. Accessed 30 Apr 2020.
- 8 Docker: Empowering App Development for Developers. <https://www.docker.com/>. Accessed 30 Apr 2020.
- 9 Vue.js - The Progressive JavaScript Framework. <https://vuejs.org/>. Accessed 30 Apr 2020.
- 10 AngularJS - Superheroic JavaScript MVW Framework. <https://angularjs.org/>. Accessed 30 Apr 2020.
- 11 Accessibility Statement | Work-life Knowledge Service. <https://worklifedata.fi/#/en/accessibility>. Accessed 19 Apr 2020.
- 12 Introduction to Web Accessibility. <https://www.w3.org/WAI/fundamentals/accessibility-intro/>. Accessed 6 Apr 2020.
- 13 Brewer, J., Web accessibility highlights and trends. 2004. Proceedings of the 2004 international cross-disciplinary workshop on Web accessibility (W4A), pp. 51–55.
- 14 Web Content Accessibility Guidelines (WCAG) 2.1. <https://www.w3.org/TR/WCAG21/>. Accessed 5 Apr 2020.
- 15 How to Meet WCAG (Quick Reference). <https://www.w3.org/WAI/WCAG21/quickref/>. Accessed 19 Apr 2020.
- 16 Support for WCAG 2.1 in axe-core. <https://www.deque.com/blog/support-for-wcag-2-1-in-axe-core/>. Accessed 5 Apr 2020.
- 17 Npm page for axe-core. <https://www.npmjs.com/package/axe-core>. Accessed 8 Sep 2019
- 18 Axe. <https://www.deque.com/axe/>. Accessed 8 Sep 2019.
- 19 Lighthouse. <https://developers.google.com/web/tools/lighthouse>. Accessed 19 Apr 2020.

- 20 Color Contrast Accessibility Validator. <https://color.a11y.com/>. Accessed 19 Apr 2020.
- 21 What is a Web Crawler? | How Web Spiders work. <https://www.cloudflare.com/learning/bots/what-is-a-web-crawler/>. Accessed 29 Mar 2020.
- 22 Googlebot. <https://support.google.com/webmasters/answer/182072?hl=en>. Accessed 29 Apr 2020.
- 23 Introductions to Headless Browsers – Multidots. <https://www.multidots.com/introduction-to-headless-browsers/>. Accessed 30 Apr 2020.
- 24 Scrapy. <https://scrapy.org/>. Accessed 29 Apr 2020.
- 25 Nightwatch.js. <https://nightwatchjs.org/>. Accessed 29 Mar 2020.
- 26 W3C Webdriver. <https://www.w3.org/TR/webdriver/>. Accessed 29 Mar 2020.
- 27 Puppeteer. <https://developers.google.com/web/tools/puppeteer>. Accessed 29 Mar 2020.
- 28 Wraith Documentation. <https://bbc-news.github.io/wraith/>. Accessed 30 Mar 2020.
- 29 DevTools Protocol. <https://chromedevtools.github.io/devtools-protocol/>. Accessed 29 Mar 2020.
- 30 npm. <https://www.npmjs.com/>. Accessed 19 Apr 2020.
- 31 TypeScript: Typed JavaScript at Any Scale. <https://www.typescriptlang.org/>. Accessed 19 Apr 2020.
- 32 Git. <https://git-scm.com/>. Accessed 19 Apr 2020.
- 33 ESLint - Pluggable JavaScript linter. <https://eslint.org/>. Accessed 19 Apr 2020.
- 34 Jest – Delightful JavaScript testing. <https://jestjs.io/>. Accessed 19 Apr 2020.
- 35 Cross-Origin Resource Sharing (CORS). <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. Accessed 19 Apr 2020.
- 36 WebSockets. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed 19 Apr 2020.
- 37 MutationObserver – Web APIs | MDN. <https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>. Accessed 30 Apr 2020.
- 38 image-equal – npm. <https://www.npmjs.com/package/image-equal>. Accessed 19 Apr 2020.
- 39 serve-static – npm. <https://www.npmjs.com/package/serve-static>. Accessed 19 Apr 2020.

40 connect – npm. <https://www.npmjs.com/package/connect>. Accessed 19 Apr 2020.

Code Listing For Crawling

```
import {Page} from 'puppeteer';
import {TestData, PageAndDom} from '../model/testdata';
import logger from '../config/testlogger';
import {Config, TimeConfig, IgnoredContentType, IgnoredContent} from '../config/config';
import {launchBrowser} from '../tools/browserutil';
import {getScreenshotFolderAndCreatelfItDoesNotExist} from '../tools/fileutil';

interface FoundLocation {
  path: string;
  item: string;
}

enum CrawlAction {
  RECORD, VERIFY
}

const LOAD_READY_FLAG = 'testmakerloadready';

const removeProtocol = (path: string): string => path.replace('https://', '').replace('http://', '');

class Crawler {

  private alreadyFound: FoundLocation[] = [];
  public crawledSites: PageAndDom[] = [];
  private readonly baseHref: string;
  private screenshotIdentifier: number = 1;
  private screenshotFolder: string;
  private timeConfig: TimeConfig;
  private ignoredContent: IgnoredContent[] = [];

  public constructor(config: Config, screenshotFolder: string) {
    this.baseHref = config.website;
    this.screenshotFolder = screenshotFolder;
    this.timeConfig = config.timeConfig;
    this.ignoredContent = config.ignoredContent;
  }

  private isNotAlreadyFound(path: string, item: string): boolean {
    if (item.includes('#')) {
      return !this.alreadyFound.some((found: FoundLocation): boolean => found.item === item);
    }
    if (!item.startsWith('/')
      && (path.endsWith(item)
        || this.alreadyFound.some((found: FoundLocation): boolean => found.path.concat(found.item).
endsWith(item)))) {
      return false;
    }
    let fullPath = item;
    if (fullPath.endsWith('/')) {
      fullPath = fullPath.substring(0, fullPath.lastIndexOf('/'));
    }
    fullPath = removeProtocol(path).concat(fullPath);
  }
}
```

```

return !this.alreadyFound.some((found): boolean => {
  return found.path.concat(found.item).endsWith(fullPath);
});
}

private async waitForPageLoad(page: Page): Promise<void> {
  try {
    await page.evaluate((loadReadyFlag): void => {
      let counter = new Date().getTime();
      const resetCounter = (): void => {
        counter = new Date().getTime();
      }
      const observer = new MutationObserver(resetCounter);
      observer.observe(document.body, {attributes: true, childList: true, subtree: true});
      setInterval(() => {
        if (new Date().getTime() - counter > 500) {
          const span = document.createElement('span');
          span.id = loadReadyFlag;
          document.body.appendChild(span);
        }
      }, 50);
    }, LOAD_READY_FLAG);
    await page.waitForSelector(`#${LOAD_READY_FLAG}`, {timeout: this.timeConfig.loadWait});
    await page.evaluate((loadReadyFlag): void => {
      // Removing load ready element is done in loop, in case there are some that are left on previous load
      const elements = document.querySelectorAll(`#${loadReadyFlag}`);
      elements.forEach((element: Element): void => {
        document.body.removeChild(element);
      })
    }, LOAD_READY_FLAG);
    await page.waitForSelector(`#${LOAD_READY_FLAG}`, {hidden: true, timeout: this.timeConfig.loadWait});
  } catch (e) {
    logger.warn(`Could not wait using observer, waiting for 500 ms instead. Error: ${e}`);
    await page.waitFor(500);
  }
}

private cleanDOM(dom: string): string {
  let cleanedDom = dom.replace(/<span id="testmakerloadready"></span>/gi, "");
  try {
    this.ignoredContent.forEach((ignored: IgnoredContent): void => {
      if (ignored.contentType === IgnoredContentType.ATTRIBUTE) {
        const regEx = new RegExp(`${ignored.value}="[^"]*"`, 'gi');
        cleanedDom = cleanedDom.replace(regEx, "");
      } else if (ignored.contentType === IgnoredContentType.TAG && dom.includes(`<${ignored.value}>`)) {
        // This presumes dom is proper HTML with end tags"
        const splitDom = cleanedDom.split(`<${ignored.value}>`);
        cleanedDom = splitDom[0];
        for (let i = 1; i < splitDom.length; i++) {
          cleanedDom += splitDom[i].split(`</${ignored.value}>`)[1];
        }
      }
    })
  }
}

```

```

});
} catch (e) {
  logger.warn(`Could not clean ignored content. Error: ${e}`);
}
return cleanedDom;
}

public async crawlInternal(page: Page, path: string, depth: number = 0): Promise<void> {
  if (!path.includes(this.baseHref) || this.crawledSites.some((site: PageAndDom): boolean => site.
page === path)) {
    return;
  }
  logger.info(`Crawling page ${path} at depth ${depth}`)
  const DOM = await page.evaluate(): string => document.body.innerHTML);
  const screenshotFile = `${this.screenshotFolder}/${this.screenshotIdentifier++}.jpg`;
  await page.waitFor(this.timeConfig.wait);
  await page.screenshot({path: screenshotFile, fullPage: false, type: 'jpeg', quality: 25});
  this.crawledSites.push({page: path, dom: this.cleanDOM(DOM), screenshotFile});
  let items = await page.evaluate((obj): string[] => {
    const ret = [];
    for (const item of document.querySelectorAll(obj.sel)) {
      const href = item.getAttribute('href');
      const target = item.getAttribute('target');
      if (href
        && !href.includes('mailto:')
        && !href.includes('tel:')
        && !target
        && (!obj.base.includes('#') || (href.includes('#') && href.startsWith('#' + obj.base.split('#')[1])))
      )
        && href.split('#').length < 3 // excludes anchor links in spa
        && (!href.includes('http') || href.includes(obj.base))) {
        ret.push(href);
      }
    }
  });
  return ret;
}, {sel: 'a', base: this.baseHref});
items = items.filter((item: string): boolean => this.isNotAlreadyFound(path, item));
this.alreadyFound = this.alreadyFound.concat(items.map((item: string): FoundLocation => {
  return {path: removeProtocol(path), item};
}));
for (const item of items) {
  try {
    await page.waitForSelector(`[href="${item}"]`);
    await page.evaluate((element): void => {
      document.querySelector(element).scrollIntoView();
      return;
    }, `[href="${item}"]`);
    await page.waitFor(this.timeConfig.wait);
    await page.click(`[href="${item}"]`);
    await page.waitFor(this.timeConfig.wait);
    await this.waitForPageLoad(page);
    try {
      const url = await page.evaluate(): string => window.location.href);
      await this.crawlInternal(page, url, depth + 1);
    } catch (e) {

```

```

        logger.error(e);
    }
} catch (e) {
    logger.error(`Could not click the link ${item} on ${path}. Error: ${e}`);
}
await page.waitFor(this.timeConfig.wait);
await page.goto(path, {waitUntil: 'networkidle2'});
await this.waitForPageLoad(page);
}
}
}
}

```

```

async function crawl(config: Config, action: CrawlAction): Promise<TestData> {
    const browser = await launchBrowser(config, false);
    const baseHref = config.website;
    logger.info('START of crawling');
    const screenshotFolder = getScreenshotFolderAndCreateIfItDoesNotExist(config.name
        ? config.name : config.website, action);
    const page = await browser.newPage();
    await page.goto(baseHref, {waitUntil: 'networkidle2'});
    const crawler = new Crawler(config, screenshotFolder);
    await crawler.crawlInternal(page, baseHref);
    browser.close();
    logger.info(`Found ${crawler.crawledSites.length} sites`);
    logger.info('END of crawling');
    return {pagesAndDoms: crawler.crawledSites, siteTested: baseHref, name: config.name};
}

```

```

export {
    crawl,
    CrawlAction
}

```

Accessibility Rules

```
const accessibilityRules: AccessibilityRule[] = [
  {
    lighthouseRule: 'color-contrast',
    description: 'The visual presentation of text and images of text has a contrast ratio of at least 4.5:1',
    name: 'Contrast',
    accessibilityLevel: AccessibilityLevel.AA,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#contrast-minimum'
  },
  {
    lighthouseRule: 'image-alt',
    description: 'All non-text content that is presented to the user has a text alternative that serves the equivalent purpose',
    name: 'Image Alt Text',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#non-text-content'
  },
  {
    lighthouseRule: 'aria-valid-attr-value',
    description: 'Contains valid values for all ARIA attributes',
    name: 'ARIA Attribute Values Are Correct',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'aria-valid-attr',
    description: 'Contains valid ARIA attributes',
    name: 'ARIA Attributes Are Correct',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'duplicate-id',
    description: 'In content implemented using markup languages, elements have complete start and end tags, elements are nested according to their specifications, elements do not contain duplicate attributes, and any IDs are unique, except where the specifications allow these features.',
    name: 'Duplicate Id',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#parsing'
  },
  {
    lighthouseRule: 'tabindex',
```

```

    descrip-
tion: 'A value greater than 0 implies an explicit navigation ordering. Although tec
hnical-
ly valid, this often creates frustrating experiences for users who rely on assistiv
e technologies',
    name: 'No Tab Index Values Above 0',
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
    url: ''
  },
  {
    lighthouseRule: 'logical-tab-order',
    descrip-
tion: 'If a Web page can be navigated sequentially and the navigation sequences aff
ect meaning or operation, focusable components receive focus in an order that prese
rves meaning and operability.',
    name: 'Logical Tab Order',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#focus-order'
  },
  {
    lighthouseRule: 'aria-allowed-attr',
    description: 'Valid ARIA allowed attributes',
    name: 'ARIA Allowed Attributes',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'aria-required-attr',
    description: 'Valid ARIA allowed attributes',
    name: 'ARIA Required Attributes',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'aria-required-children',
    description: 'Valid ARIA required children',
    name: 'ARIA Required Children',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'aria-required-parent',
    description: 'Valid ARIA required parent',
    name: 'ARIA Required Parent',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {

```

```

    lighthouseRule: 'audio-caption',
    descrip-
tion: 'Captions are provided for all prerecorded audio content in synchronized medi
a, except when the media is a media alternative for text and is clearly labeled as
such.',
    name: 'Audio Captions',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#captions-prerecorded'
  },
  {
    lighthouseRule: 'button-name',
    descrip-
tion: 'For all user interface components (including but not limited to: form elemen
ts, links and components generated by scripts), the name and role can be programmat
ical-
ly determined; states, properties, and values that can be set by the user can be pr
ogrammat
ical-
ly set; and notification of changes to these items is available to user agents, inc
luding assistive technologies.',
    name: 'Button Name',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#name-role-value'
  },
  {
    lighthouseRule: 'bypass',
    descrip-
tion: 'A mechanism is available to bypass blocks of content that are repeated on mu
ltiple Web pages.',
    name: 'bypass',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#bypass-blocks'
  },
  {
    lighthouseRule: 'definition-list',
    description: 'Valid definition lists',
    name: 'Definition List',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'dlitem',
    description: 'Valid definition list items',
    name: 'Definition List Items',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'document-title',

```

```

description: 'Web pages have titles that describe topic or purpose.',
name: 'Document Title',
accessibilityLevel: AccessibilityLevel.A,
url: 'https://www.w3.org/WAI/WCAG21/quickref/#page-titled'
},
{
  lighthouseRule: 'frame-title',
  description: 'For all user interface components (including but not limited to: form elements, links and components generated by scripts), the name and role can be programmatically determined; states, properties, and values that can be set by the user can be programmatically set; and notification of changes to these items is available to user agents, including assistive technologies.',
  name: 'Frame Title',
  accessibilityLevel: AccessibilityLevel.A,
  url: 'https://www.w3.org/WAI/WCAG21/quickref/#name-role-value'
},
{
  lighthouseRule: 'html-has-lang',
  description: 'The default human language of each Web page can be programmatically determined.',
  name: 'HTML Has Language',
  accessibilityLevel: AccessibilityLevel.A,
  url: 'https://www.w3.org/WAI/WCAG21/quickref/#language-of-page'
},
{
  lighthouseRule: 'html-lang-valid',
  description: 'The default human language of each Web page can be programmatically determined.',
  name: 'htmlValidLanguage',
  accessibilityLevel: AccessibilityLevel.A,
  url: 'https://www.w3.org/WAI/WCAG21/quickref/#language-of-page'
},
{
  lighthouseRule: 'input-image-alt',
  description: 'All non-text content that is presented to the user has a text alternative that serves the equivalent purpose',
  name: 'Image Input Alternative Text',
  accessibilityLevel: AccessibilityLevel.A,
  url: 'https://www.w3.org/WAI/WCAG21/quickref/#non-text-content'
},
{
  lighthouseRule: 'label',

```

```

    descrip-
tion: 'Information, structure, and relationships conveyed through presentation can
be programmatically determined or are available in text.',
    name: 'Label',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#info-and-relationships'
  },
  {
    lighthouseRule: 'layout-table',
    descrip-
tion: 'When the sequence in which content is presented affects its meaning, a corre
ct reading sequence can be programmatically determined.',
    name: 'Layout Tables',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#meaningful-sequence'
  },
  {
    lighthouseRule: 'link-name',
    descrip-
tion: 'The purpose of each link can be determined from the link text alone or from
the link text together with its programmatically determined link context, except wh
ere the purpose of the link would be ambiguous to users in general.',
    name: 'Link Has Discernible Name',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#link-purpose-in-context'
  },
  {
    lighthouseRule: 'list',
    description: 'Valid lists',
    name: 'list',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'listitem',
    description: 'Valid list items',
    name: 'list Item',
    accessibilityLevel: AccessibilityLevel.A,
    url: ''
  },
  {
    lighthouseRule: 'meta-refresh',
    descrip-
tion: 'Users do not expect a page to refresh automatically, and doing so will move
fo-
cus back to the top of the page. This may create a frustrating or confusing experie
nce.',
    name: 'Meta Refresh',

```

```

accessibilityLevel: AccessibilityLevel.A,
url: 'https://www.w3.org/WAI/WCAG21/quickref/#timing-adjustable'
},
{
  lighthouseRule: 'meta-viewport',
  description: '[user-scalable="no"]` is used in the `

```

```
    accessibilityLevel: AccessibilityLevel.A,  
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#captions-prerecorded'  
  },  
  {  
    lighthouseRule: 'video-description',  
    description: 'For prerecorded audio-only and prerecorded video-  
on-  
ly media, the following are true, except when the audio or video is a media alterna  
tive for text and is clearly labeled as such:',  
    name: 'Video Description',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#audio-only-and-video-only-  
prerecorded'  
  },  
  {  
    lighthouseRule: 'custom-controls-labels',  
    descrip-  
tion: 'Custom interactive controls have associated labels, provided by aria-  
label or aria-labelledby',  
    name: 'Custom Controls Have Labels',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: ''  
  },  
  {  
    lighthouseRule: 'custom-controls-roles',  
    description: 'ARIA roles for custom controls',  
    name: 'Custom Controls HaveRoles',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: ''  
  },  
  {  
    lighthouseRule: 'focus-traps',  
    descrip-  
tion: 'If keyboard focus can be moved to a component of the page using a keyboard i  
nter-  
face, then focus can be moved away from that component using only a keyboard interf  
ace, and, if it requires more than unmodified arrow or tab keys or other standard e  
xit methods, the user is advised of the method for moving focus away.',  
    name: 'focusTraps',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#no-keyboard-trap'  
  },  
  {  
    lighthouseRule: 'focusable-controls',  
    descrip-  
tion: 'Custom interactive controls are keyboard focusable and display a focus indic  
ator.',  
    name: 'focusableControls',
```

```
    accessibilityLevel: AccessibilityLevel.A,  
    url: ''  
  },  
  {  
    lighthouseRule: 'interactive-element-affordance',  
    description:  
tion: 'Interactive elements, such as links and buttons, should indicate their state  
and be distinguishable from non-interactive elements.',  
    name: 'interactive Elements Are Distinguishable',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: ''  
  },  
  {  
    lighthouseRule: 'link-text',  
    description: 'All non-  
text content that is presented to the user has a text alternative that serves the e  
quivalent purpose',  
    name: 'Link Text',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#non-text-content'  
  },  
  {  
    lighthouseRule: 'structured-data',  
    description:  
tion: 'Information, structure, and relationships conveyed through presentation can  
be programmatically determined or are available in text.',  
    name: 'Structured Data',  
    accessibilityLevel: AccessibilityLevel.A,  
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#info-and-relationships'  
  },  
  {  
    lighthouseRule: 'aria-roles',  
    description: 'Valid Aria Roles',  
    name: 'Aria Roles',  
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,  
    url: ''  
  },  
  {  
    lighthouseRule: 'image-aspect-ratio',  
    description: 'Valid aspect ratios for images',  
    name: 'Image Aspect Ratio',  
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,  
    url: ''  
  },  
  {  
    lighthouseRule: 'notification-on-start',
```

```
    descrip-
tion: 'Users are mistrustful of or confused by sites that request to send notificat
ions without context. Consider tying the request to user gestures instead.',
    name: 'Notification on Start',
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
    url: ''
  },
  {
    lighthouseRule: 'geolocation-on-start',
    descrip-
tion: 'Users are mistrustful of or confused by sites that request their location wi
thout context. Consider tying the request to user gestures instead.',
    name: 'Geolocation on Start',
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
    url: ''
  },
  {
    lighthouseRule: 'valid-lang',
    descrip-
tion: 'The human language of each passage or phrase in the content can be programma
tical-
ly determined except for proper names, technical terms, words of indeterminate lang
uage, and words or phrases that have become part of the vernacular of the immediate
ly surrounding text.',
    name: 'Valid Language On Elements',
    accessibilityLevel: AccessibilityLevel.AA,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#language-of-parts'
  },
  {
    lighthouseRule: 'heading-levels',
    descrip-
tion: 'In content implemented using markup languages, elements have complete start
and end tags, elements are nested according to their specifications, elements do no
t contain duplicate attributes, and any IDs are unique, except where the specificat
ions allow these features.',
    name: 'Heading Levels',
    accessibilityLevel: AccessibilityLevel.A,
    url: 'https://www.w3.org/WAI/WCAG21/quickref/#parsing'
  },
  {
    lighthouseRule: 'managed-focus',
    descrip-
tion: 'If new content, such as a dialog, is added to the page, the user`s focus is
directed to it.',
    name: 'Managed Focus',
    accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
    url: ''
  },
  },
```

```

{
  lighthouseRule: 'use-landmarks',
  description: 'Landmark elements (<main>, <nav>, etc.) are used to improve the keyboard navigation of the page for assistive technology',
  name: 'Use of Landmarks',
  accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
  url: ''
},
{
  lighthouseRule: 'visual-order-follows-dom',
  description: 'DOM order matches the visual order, improving navigation for assistive technology.',
  name: 'Visual Order Follows Dom',
  accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
  url: ''
},
{
  lighthouseRule: 'font-size',
  description: 'Font sizes less than 12px are too small to be legible and require mobile visitors to “pinch to zoom” in order to read. Strive to have >60% of page text ≥12px.'
},
{
  name: 'Font Size',
  accessibilityLevel: AccessibilityLevel.NOT_DETERMINED,
  url: ''
},
{
  lighthouseRule: 'tap-targets',
  description: 'The size of the target for pointer inputs is at least 44 by 44 CSS pixels',
  name: 'Tap Targets',
  accessibilityLevel: AccessibilityLevel.AAA,
  url: 'https://www.w3.org/WAI/WCAG21/quickref/#target-size'
},
];

```