

HERIOT-WATT UNIVERSITY

FOURTH YEAR DISSERTATION

A Generator for Accessible HTML Forms

James McLean

Software Engineering MEng

Supervised by

Andrew Ireland

Declaration of Authorship

I, James William McLean confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: James McLean

Date: 14/04/2024

Acknowledgements

Firstly I would like to thank my supervisor, Andrew Ireland, for guiding me through the process of this project and organising various opportunities for my research and evaluation.

I would also like to thank Jim McCafferty for talking with me about his experiences and expertise in the area of accessible formats.

Thirdly, I would like to thank Michael Conlon and John Spinks from the Heriot-Watt Web Applications team for providing important feedback during the evaluation stage of this project.

Finally, I would like to thank my father, Ian McLean, for taking the time to proofread this document.

Abstract

Making the Web accessible is a chief concern in today's internet reliant world. This document will outline the solutions that have been developed for making Websites accessible as well as how these solutions have been standardised. It also analyses the different approaches that can be taken in the application of these standards, as well as identifying a new approach. The document details the development and evaluation of a tool which aids in applying this new approach to web forms.

Abbreviations

- WF - Well Formed
- V1 - (Well Formed) Version 1
- V2 - (Well Formed) Version 2
- FT - Form test (test case for Version 1)
- AT - Application Test (test case for Version 2)
- DOM - Document Object Model

Contents

1	Introduction	6
1.1	Introduction to subject matter and motivations	6
1.2	Aims and Objectives	6
1.2.1	Objectives	7
1.3	Overview of Report	7
1.4	Citation Note	8
2	Background	9
2.1	Need for accessibility in web services	9
2.2	What came before WCAG	9
2.3	Web Content Accessibility Guidelines	10
2.3.1	Version 1.0	10
2.3.2	Version 2.0 - 2.2	11
2.3.3	Version 3	12
2.3.4	Timeline	13
2.4	Other Standards	13
2.4.1	DAISY	13
2.4.2	UKAAF	13
2.5	Stake Holders	14
2.5.1	Deaf Users	14
2.5.2	Users with vision impairment	14
2.5.3	Users with motor difficulties	15
2.6	Existing Software Infrastructure	16
2.6.1	ARIA	16
2.6.2	Keyboard Navigation	17
2.6.3	Skip to content button	17
2.7	Technical Tools	19
2.7.1	React	19

2.7.2	Visual Studio Code	19
2.7.3	GitHub	19
3	Requirements and Analysis	21
3.1	Problem Description and Analysis	21
3.2	Solution Description	23
3.3	User Story	23
3.4	Functional Requirements	24
3.5	Non-Functional Requirements	27
3.6	Design Summary	28
4	Version 1 - Core Functionality	29
4.1	Version Breakdown	29
4.2	System Design	31
4.3	Scripting	32
4.4	Development	32
4.4.1	Initial Setup	32
4.4.2	Basic Classes	32
4.4.3	Generation	33
4.4.4	Completion	35
4.4.5	Styling	37
4.5	Version 1 Summary	37
5	Version 2 - Web Application	38
5.1	Version Breakdown	38
5.2	System Design	39
5.3	Library and Framework	39
5.3.1	React	39
5.3.2	create-react-app	40
5.4	Development	41
5.4.1	Setup and boilerplate modification	41
5.4.2	Create Form Area	41
5.4.3	Delivery Area	44
5.4.4	Branding and Further Visual Additions	48
5.5	Version 2 Summary	49

6	Testing and Design Refinements	50
6.1	Version 1 Testing	50
6.1.1	Version 1 Manual Accessibility Testing	50
6.1.2	Version 1 Extension Accessibility Testing	51
6.2	Version 2 Testing	52
6.2.1	Version 2 Manual Accessibility Testing	52
6.2.2	Version 2 Extension Testing	54
6.3	Testing Summary	55
7	Evaluation	56
7.1	Evaluation Description	56
7.2	Evaluation Results	57
7.2.1	Presentation	57
7.2.2	Bad Form Example	57
7.2.3	Demonstration	58
7.2.4	Discussion	58
8	Conclusion	61
8.1	Requirements Conformance	61
8.2	Non-Functional Requirements	65
8.3	Future Work	66
A	Testing Log	67
B	PLES	72
B.1	Social Implications	72
B.2	Consent Form	72
C	User Guide	74
D	Presentation	76

Chapter 1

Introduction

1.1 Introduction to subject matter and motivations

In any system which involves a person's participation, accessibility is a vital concern. Public buildings must provide disabled access in the form of ramps and lifts; road crossings must have tactile paving to indicate location and type of crossing.

In this vein, the importance of accessibility in computer interaction cannot be understated. Websites in particular need to have good accessibility as they are such a universal aspect of computer usage. With so many general services provided through the web, if accessibility is not considered, a large section of the user base is cut off.

There are many aspects to web accessibility, so this project will focus in on one particular area: forms. These are an essential component of information exchange online, so they need to be fully usable.

1.2 Aims and Objectives

This document will discuss the challenges faced in making the web accessible and the solutions already available to overcome these. It also details the development of Well Formed, an application which generates user designed accessible web forms.

These forms conform to the WCAG Standard, the universally accepted

accessibility standard devised by The World Wide Web Consortium (W3C).

Well Formed has an easy to use interface accessed through a web browser, and allows users to input the field information for their desired form. After designing the form, the tool generates it and presents it to the user along with an accompanying CSS file. This allows developers to create a form which will be compatible with screen readers and other accessibility tools without needing to take a course in online accessibility.

1.2.1 Objectives

The following is a list of the primary objectives of the project:

1. To highlight the need for accessibility in digital systems.
2. To show the challenges faced by users with disabilities and how to overcome them through system design.
3. To analyse the various methods of achieving accessibility in a websites.
4. To identify an alternative approach to developing website features with accessibility.
5. To create a system which can take a form design and turn it into an accessible HTML form elements.
6. To create an easy to use interface to this system which is itself accessible.

1.3 Overview of Report

This report is split into several chapters to give a wide overview of the subject as well as detail the development and evaluation of the application. Chapter 2 gives an in depth look at the history of web accessibility, especially the WCAG Standard Specification since its inception in 1999. Chapter 3 studies the problem of developing an accessible website and the approaches developers can take. It outlines the requirements for the Well Formed application.

Chapter 4 covers the creation of the form generation code, detailing the development and design decisions taken. Chapter 5 covers the expansion of this system into a full web application by integrating it into a React

framework with a front end. Chapter 6 explains the testing and refinement process applied to the application.

Chapter 7 details the evaluation process of the application run with members of the Heriot-Watt Web Applications Team. Finally Chapter 8 is a conclusion, reflecting on what was achieved during the project and how work could be continued in future.

1.4 Citation Note

Sources, where used, are cited, however, much of my knowledge of this subject comes from the edX course “Introduction to Web Accessibility” [1]. I completed this course as part of an internship I had over summer 2023.

Chapter 2

Background

2.1 Need for accessibility in web services

This chapter will discuss the challenges faced in delivering an equally functional experience to all users regardless of ability, as well as the solutions currently available. The primary point of interest is the WCAG standards maintained by W3C since the late nineties, with the various methods of implementing the specifications on the web also being detailed.

2.2 What came before WCAG

In 1990, the Americans with Disabilities Act (ADA) was passed, prohibiting disability discrimination in employment and a number of other areas. One of these areas, public accommodation, originally referred to places such as shops and residential locations, had its definition expanded in 1996 to include websites.

In 1998 the US Government made an amendment to the 1967 Rehabilitation Act, stipulating that Federal digital technologies needed to be accessible.

However, despite these acts being passed, there was still little guidance in how to achieve digital accessibility until the WCAG standard was released.

2.3 Web Content Accessibility Guidelines

2.3.1 Version 1.0

The first version of WCAG was released by W3C in 1999[2]. It covers HTML features such as:

- Alternative text (Guideline 1)
- Graceful transformation of elements such as tables, as well as new element types in future (Guidelines 5 & 6)
- Allowing for user control for time sensitive content (Guideline 7)
- Etc.

It also gives guidance on page content, advising developers on writing text in a way that will be clear to all. (Guidelines 4 & 14)

The checkpoints of each guideline are prioritised from one to three; 1: Must satisfy, 2: Should satisfy, 3: May satisfy. Conformance levels are defined for meeting checkpoints at a certain priority:

- *A: All priority 1 checkpoints are satisfied*
- *Double-A: All priority 1 & 2 checkpoints are satisfied*
- *Triple-A: All priority 1, 2 & 3 checkpoints are satisfied*

For example, Guideline 2 “Don’t rely on color alone” has two checkpoints:

- *2.1 Ensure that all information conveyed with color is also available without color, for example from context or markup. [Priority 1]*
- *2.2 Ensure that foreground and background color combinations provide sufficient contrast when viewed by someone having color deficits or when viewed on a black and white screen. [Priority 2 for images, Priority 3 for text]*

The conformance levels for a web page would be:

Checkpoint	A	AA	AAA
2.1	✓	✓	✓
2.2 <i>[for images]</i>		✓	✓
2.2 <i>[for text]</i>			✓

However, the specification could have been written and structured in a better way. It is overly wordy with each guideline having a preamble before each checkpoint. Jonathan Hassell, Founder and CEO of Hassell Inclusion, an accessibility consultancy firm, worked at the BBC when the standard was released. He said this: “[WCAG] had also just come out and nobody understood them. It was my job to translate it. I had to understand what this thing was, and how to make sure that people in all the BBC teams could implement it.”[3]

Furthermore, the guidelines (especially the later ones) lack clear organisation, leading to the document feeling cobbled together with little regard for flow.

2.3.2 Version 2.0 - 2.2

WCAG 2.0 was finalised in 2008[4]. It completely overhauled and updated the original 1999 specification to reflect how mainstream web use had solidified over the previous nine years.

The main incentive behind this new standard was the diversification of web connected devices that had developed through the 2000s. Since the 1990s the World Wide Web had gone from being confined to a plugged in desktop computer to being available on the go through mobile phones and other such devices. This brings the added complexity of creating a functional website even if all features are not supported by the device.

Version 2.0 introduced four principles:

- *Perceivable* - elements must be visible to the user
- *Operable* - the user must be able to operate interface elements
- *Understandable* - information and interfaces must be clear and comprehensible to the user
- *Robust* - content must be designed thoroughly to work across different technologies

The twelve guidelines are organised by these principles, adding some much needed structure to the document.

A further improvement from the previous specification is the removal of the preamble given to each guideline before the checkpoints (now referred to as success criteria), greatly improving the readability by getting straight to the important points.

The compliance levels are also plainer, with priorities 1, 2 and 3 being replaced with “A”, “AA”, “AAA” respectively. This is more direct than the previous specification’s approach, showing the separation of priority and compliance levels to be redundant and confusing.

In 2018 the standard was updated to 2.1[5] reflecting the further emergence of mobile browsing. It also addressed users with low vision[6], bringing in requirements for page scaling and adjustment. These new success criteria accommodate users with reduced vision, allowing them to use websites visually rather than relying on a screen reader.

Version 2.2 was released in October 2023, and is just a minor update on 2.1.[7] Nine new success criteria are added, three of which target the visibility of keyboard focus, a vital consideration for keyboard users who do not have a vision impairment.[8] Two of the new criteria (3.3.8 & 3.3.9 “Accessible Authentication”) address users with cognitive disabilities that may impact their ability to recall login details or solve authentication puzzles. It promotes proving identity through a link in an email.

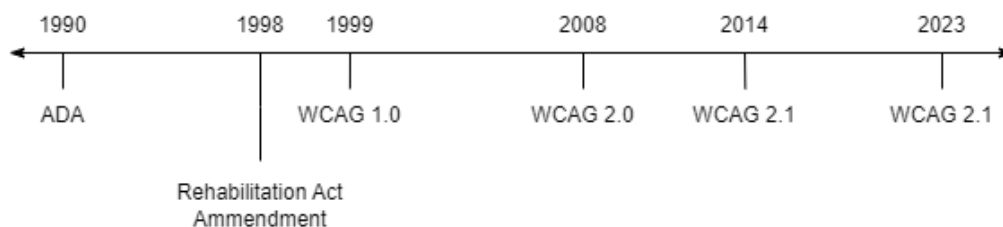
2.3.3 Version 3

In 2021 the first working draft of a new version of the WCAG specification was released. Version 3.0 is intended to be a fresh overhaul, akin to 2.0 in 2008. The primary objectives of this new version are to be more understandable; address more issues faced by users with cognitive disabilities; and cover different types of “web content, apps, tools and organisations”.

The latest draft at time of writing was released in July 2023 and is still mainly composed of placeholder content, however it does contain details of two potential new conformance approaches. The first approach is based around “Outcomes that can be tested”; similar to the previous approach this aims to set a precedent to meet. The second approach is based around “Assertions and procedures” which the document describes as statements that a site was developed according to a certain accessibility procedure.

A change to the structure of the document from previous specifications is the addition of a plain language summary of each section. This is a clear sign of the aim to improve understanding of the standard.

2.3.4 Timeline



2.4 Other Standards

2.4.1 DAISY

For digital documents there is the Digital Accessible Information System (DAISY) standard[9]. This is a specification for a Digital Talking Book (DTB) format for documents such as articles, audio books and reports. It outlines an XML based system designed around providing an intuitive audio based experience for users who cannot make use of visually conveyed text.

The DTB package of DAISY multimedia contains: publication metadata (title, author, year, etc.), the manifest (list of files included), the spine (reading order of documents), tours (alternate reading structures), and the guide (structural fundamentals: contents table, bibliography, etc.).

The great advantage of a dedicated audio format such as DAISY is that it gives users a great deal more flexibility than a standard audio recording of the document. They can adjust voice settings, such as speed and accent, as well as navigate documents in a more intuitive fashion.

DTB packages can be read using a dedicated DAISY reader; software on a computer or mobile device; or other digital devices such as Braille readers.

2.4.2 UKAAF

The UK Association for Accessible Formats maintains a set of standards for making documents accessible in the UK. It covers a variety of formats including: Braille, PDF, Audio, and Office Documents. In the section covering HTML it lists WCAG as the standard to follow[10].

2.5 Stake Holders

Different disabilities will pose different challenges when using a digital system. This section explores those challenges and how users overcome them when interacting with a system.

2.5.1 Deaf Users

Deaf users, unless they have sight or motor impairments, will be able to use a keyboard and mouse in conjunction with a screen. The primary concern for accommodating these users is perceivability. Audio content needs to be available in alternative forms.

This means videos with people must at least be available with closed captions¹. For pure audio content, a transcript of the audio should be supplied so it can be read. This not only to help users who cannot hear the content, but also users are unable to play the audio, for example if their device does not have speakers or they are in a public place.

2.5.2 Users with vision impairment

Vision Impaired users are a major factor when it comes to digital accessibility. Human-computer interaction is heavily reliant on visual displays for conveying information so accommodating those who are unable to utilise displays requires careful consideration when designing content.

These users can generally use some form of keyboard interaction to navigate content, and, depending on the nature of their impairment, a mouse.

A highly valued tool for vision impaired users is screen reader software for describing visual content through audio. This can be very effective provided the content has been formatted correctly. For a screen reader to be useful to the user, alternative text needs to be descriptive and succinct, link text must be meaningful, forms fields must be connected to labels, meta-data must be accurate, and tab order must be correct to the flow of the page.

Classic examples of poor screen reader optimisation include:

- Calling a link “Click Here!” - Confusing when using keyboard navigation as surrounding context will not be read.

¹Not to be confused with subtitles.

- Form fields with no label connection - The screen reader will read the fields as generic inputs.
- Multiple address lines with only one “Address” label in a form - Only one line can use the label, the others will be read as generic text inputs.

Not only does this undermine the functionality of screen readers, but also other tools based on a similar framework such as Braille readers. Braille readers are designed around a similar principal to screen readers, however, rather than using audio, information is conveyed through a dynamic Braille display. For some users this is an easier way of navigating digital content as Braille displays can be read faster than screen readers can speak, allowing for a more efficient experience, much closer to that of using a screen.

I spoke with Jim McCafferty about his experiences of using digital systems as a vision impaired user. Jim has worked as a Transcriber and Proof Reader for the Scottish Braille Press as well being a Board Member of UKAAF. He has experience with screen readers, but prefers his Braille Reader. He demonstrated the speed and ease with which he can use it for a variety of tasks, from using websites to reading books and checking his emails.

However, though Braille Readers do not function in exactly the same way to screen readers, they still rely on correct use of web accessibility features.

We must also consider vision impaired users who do not necessarily need to use an alternative to a screen. For partially sighted users, using a screen may be preferable to the previously mentioned alternatives, provided the content is scaleable or adjustable. Beyond allowing for graceful scaling of content when zooming in and out, content adjustments can range from simply allowing for a high contrast viewing mode through to making plain text versions of pages available.

2.5.3 Users with motor difficulties

Some users will be unable to operate precision based input devices such as mice, so digital content needs to be navigable using just keyboard input. To keep it intuitive simple to follow, keyboard navigation is linear, following the flow of information from top to bottom².

The standard hotkeys for navigating forward and back through page elements are *tab* and *shift + tab*, however other options are available depending on the user’s ability. For users who cannot use keyboard keys, options such as

²The horizontal flow of information at each level will depend on the cultural norm.

pumps and pedals or even blowing or sucking on a straw can be an alternative for these actions.

For users with no ability to interact with a device through physical means, eye tracking systems are available. These work similarly to a mouse, and can be used in conjunction with an onscreen keyboard.

2.6 Existing Software Infrastructure

2.6.1 ARIA

ARIA (Accessible Rich Internet Applications) is a set of attributes for HTML elements which can be used to provide information to accessibility tools such as screen readers and Braille Readers. ARIA attributes are primarily aimed at non-native elements. These are page features which do not use a traditional HTML element, for example a list made from div elements rather than the HTML list element

These attributes can be used to override default reader behaviour. For example, a link could be defined like this:

```
<a href="https://en.wikipedia.org/wiki/Roman_Empire">Roman Empire</a>
```

A screen reader would read something to the effect of, “Link - Roman Empire”. However, the link could use the `aria-label` attribute like this:

```
<a href="https://en.wikipedia.org/wiki/Roman_Empire"
aria-label="Roman Empire Wikipedia page">Roman Empire</a>
```

The screen reader would read the `aria-label` rather than the link text which would be rendered on the page.

Another notable ARIA attribute is `aria-role`. This can be used to specify information about a non-native element’s purpose which is not obvious from the element itself. Valid roles include: *Nav*, *Banner*, *Feed*, *Document*, *Grid*, *List/ListItem*. For example if a page has a listing that does not utilise the HTML list element, the outer element of the listing could be given the *list* role and each item could be given the *listitem* role. This would indicate to the screen reader how to treat the element.

There are also ARIA attributes for indicating an elements state. Examples of state properties include:

- *aria-expanded* - An expandable element is expanded or collapsed.
- *aria-disabled* - An element such as a field or a button is disabled (cannot be used, may appear greyed out).
- *aria-required* - A field must be filled out to submit the form.

2.6.2 Keyboard Navigation

For users with motor difficulties and screen reader users, keyboard navigation is a priority. In order to facilitate this a clear tab order must be kept.

When the browser reads the web page DOM, it will work out the tab order of selectable elements automatically. However, correctness is not guaranteed as some elements such as menus and content at the side of the screen could break the flow.

To aid in maintaining good keyboard navigation, selectable elements have a “tabindex” attribute to guide the browser when generating the tab order of the page. This can be used to specify precedence of elements when navigating with a keyboard. For example, the tabindex attributes for a page could be:

- Navigation Bar links - 1
- Main Body elements - 2
- Footer links - 3

The order of elements with the same tabindex is decided based on the DOM so it is best to use the same value for whole sections to indicate the order of the page.

2.6.3 Skip to content button

A very useful feature for aiding keyboard navigation is the “Skip to Content” button, a button which jumps the keyboard focus to the beginning of the main body of the page. This means keyboard users do not need to cycle through elements such as menus, which are common to every page.

Most non-keyboard users are completely unaware of this feature, as it is only visible when selected (See Fig 2.1). The button is the first selectable element so will be the first option when tabbing through the page.

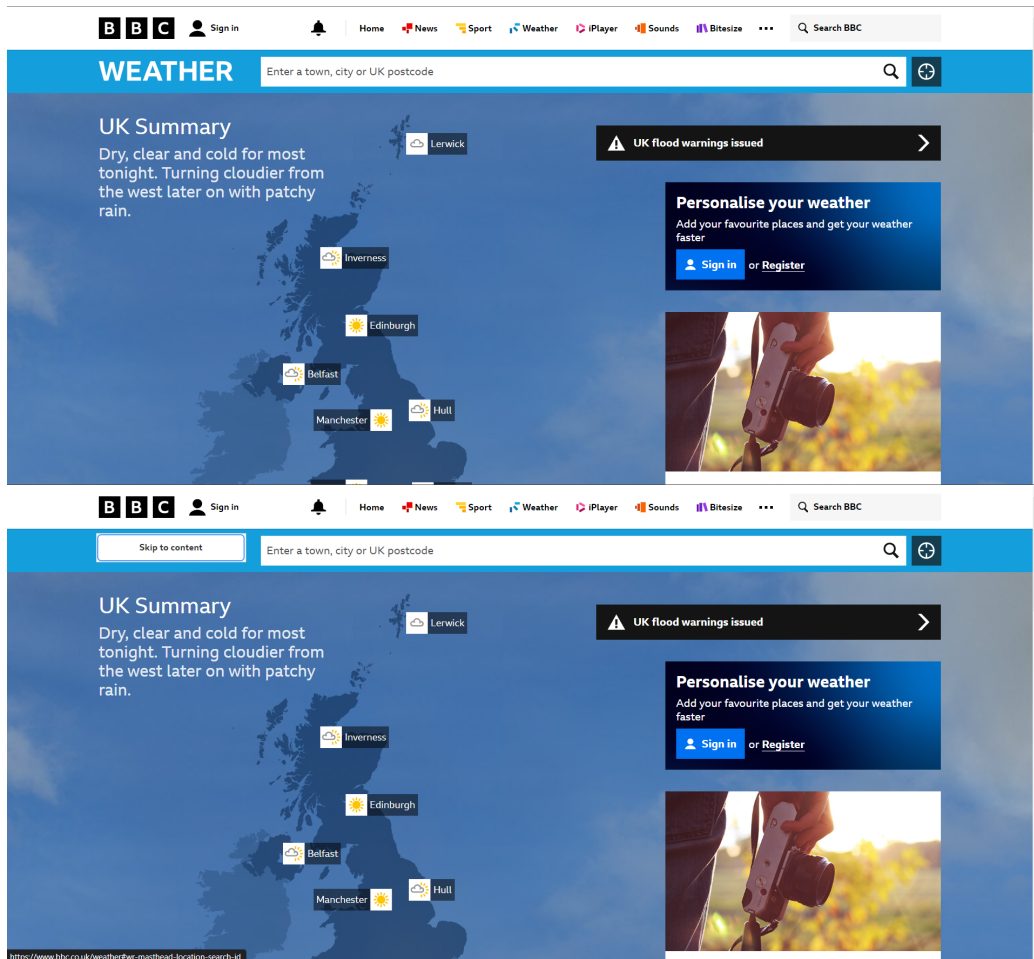


Figure 2.1: BBC Weather website with Skip to Content button hidden and shown.

2.7 Technical Tools

Here are details of the technical tools I researched in anticipation of developing of Well Formed.

2.7.1 React

The traditional structure of a web page is: an HTML document containing the elements to appear on the page; a CSS file containing styling for the page (sizing, colours, fonts, positioning, etc.); Javascript files for adding interactivity and dynamic content.

React is a Javascript library for dynamically generating and modifying web pages. This is ideal for web applications which rely on intensive interaction as it allows the page to be modified and updated in a programmatic way. The page is generated from *components*, small HTML sections which are put together. *Components* can be updated to change the rendered page in real time.

It is suggested on the React website that the library should be used in conjunction with a framework to keep project structure. For this there were a number of options, but two in particular stuck out; Next.js and create-react-app.

Next.js is a very popular framework as it has a great scope for large web applications with an emphasis in server side processing. Create-react-app in comparison, is a simpler, framework, providing the basics for developing an application in React.

2.7.2 Visual Studio Code

Visual Studio Code is an open-source code editor from Microsoft. It supports a variety of extensions, making it a powerful modular environment for working on the project. It has an easy to use source control system which can be integrated with GitHub.

2.7.3 GitHub

GitHub is a widely used code repository system which will be used for storing the project's source code. Client-side web projects such as this can be easily

accessed through GitHub's Pages feature, meaning the tool can be hosted for free.

Chapter 3

Requirements and Analysis

3.1 Problem Description and Analysis

Web forms are an especially important consideration for accessibility. When purely conveying information, poor accessibility can lead to confusion and misunderstanding; when user input is required, poor accessibility can be seriously detrimental to the user.

If fields are not clearly labeled, users, regardless of their ability, may input data incorrectly. For example, if a form has a text field called “Name”, it is unclear what data is expected. There is no indication as to whether they should give just their first name, their first and surname or their full name with middle names. The label must be as explicit as possible.

However, there is a much bigger issue if the source code behind the form is not written with accessibility in mind. There could be a text field with no connection between the label and the input. If some accessibility software such as a screen reader is required by the user, this will just be read as a generic blank text field. The user will have absolutely no indication of what input is expected.

For many people building a website themselves, learning the WCAG specification properly can be a large undertaking for what will generally be seen as a small aspect of website design. While the specification is written to be understandable, effective implementation requires additional reading from other sources. W3C provides a course covering how to implement WCAG compliant accessibility features, as well as giving insights from users who benefit from said features[1]. However, as valuable as this course is, it is

designed to take 16 - 20 hours to complete, with a recommendation of splitting this across four weeks. This is fine for a professional Web Developer, however, for someone with basic HTML and CSS skills, it is too much for just building a website for themselves.

I took this course for a summer placement where I was involved with accessibility features and when discussing my work with colleagues who had not taken the course, I often needed to explain not only what features I was implementing, but also why they needed to be implemented in a certain way. Courses like this are incredibly effective not just for explaining what needs to be done to meet the standard, they also give an insight into how the standard benefits users.

Another route they could take is using a website building tool. There are many of these; Wix, Squarespace and the GoDaddy Website Builder to name just a few. These tools are generally template reliant and employ an element based paradigm, usually through a drag and drop based interface. Users do not need any prerequisite web design knowledge to use these services as they are designed to be intuitive to users with basic computer literacy skills.

Wix's accessibility statement asserts that their features are WCAG 2.0 compliant[11] while Squarespace's explains they are constantly updating their products to meet requirements[12].

While website builders are an effective solution for anyone wanting to pick up and create, users with web design experience may not want to be confined by their template based nature. These tools become very clunky when trying to create something truly bespoke.

A further drawback is that users cannot just create a single element using these builders, they are confined to the builder for their whole website.

There is a third less structured approach a designer could take; post-hoc accessibility checking. Tools exist in the form of browser extensions such as Siteimprove Axe, Wave, and Tenon, which can analyse a web page's source code for accessibility errors. A developer can build a web page, then use one of these extensions to retroactively find and fix vulnerabilities. While these tools are useful for checking for errors before deployment, relying on them while building the website slows down development time considerably.

3.2 Solution Description

The purpose of this project is to fill the gap for users who have HTML/CSS experience yet no accessibility training. The proposition for this solution is Well Formed, an application which developers can use to automatically generate a web form with WCAG compliant accessibility features. This tool will take the form of a web application with a simple interface. Users will select and name the fields they want, then the system will generate an HTML form element supplemented with a CSS file for styling. The user can easily drop these into their website.

The output HTML element will itself be incomplete as the connections for sending the form data will have to be filled out by the user. The CSS file will contain either full styling or just basic styling with plenty of room for the user to customise further.

The following table shows the relative benefits and drawbacks of different accessibility approaches:

Method	Simple	Customisable
Using website builder	✓	
Learning by Spec		✓
Proposed Tool	✓	✓

3.3 User Story

A user story for the proposed tool:

“I am setting up a website containing information on the services provided by my small business. I have experience with building web pages with HTML and CSS and will create the website myself. I want the website to contain a customer feedback form which must be accessible. Since the website is a minor part of the business, I cannot spend time testing pages conform to accessibility expectations. A guaranteed accessible premade element would be ideal.”

This can be contrasted with a user story for a post-hoc approach using a browser extension based tool:

“I am building an online shop for my small business. I have HTML and CSS experience so I will create the website myself. I want the shop to be accessible to my customers and since the website is such a key element of the

business I can dedicate time to developing the accessibility features correctly. A tool which will verify my work for modification would be ideal.”

3.4 Functional Requirements

Functional User Requirements have IDs starting FUR while Functional System Requirements have IDs starting FSR.

ID	Title	Description	Priority
FUR1	The user must be able to add fields to a form.	On the form designer page the user will be able to click “+” button to add a new field to the form. A new field edit component will be added to the form designer.	Must
FUR1-1	The user must be able to enter the name of the field.	Each field edit component will have a name input where the user will edit the field’s name	Must
FUR1-2	The user must be able to select from a list of field types.	Each field edit component will have a drop-down menu containing the various field types for the user to choose from. The available types will be: Text, Date, Dropdown, Radio, Address, Email.	Must
FUR1-3	The field edit component must display different input controls for different types.	After the user has selected a field type, the edit component should provide the relevant input components to the field, e.g. selection options for radio/drop-down fields.	Must

FUR1-4	The user must be able to set a field as “Required”.	Each field edit component will have a checkbox allowing the user to set the field to require an input.	Must
FUR1-5	The tool should provide naming guidance.	The WCAG spec recommends what to avoid when naming fields, this guidance should be presented to the user [perhaps with a popup on the name field].	Should
FSR1	The system should have a Start Page.	The application should open to a start page to give some information on the tool before proceeding to allow the user to design a form.	Should
FUR2	The user must be able to remove fields from a form.	The field edit component will contain a “delete” button the user can click to remove the field from the form.	Must
FUR3	The user could be able to reposition fields in a form.	Each form component could have a control option to drag the component up and down the list. For keyboard users this could be achieved through selecting the move control and using the up and down arrow keys.	Could
FUR4	The application must provide the user with a form element when they finish designing.	The primary output of the tool is an HTML form element using the design specified by the user.	Must
FUR4-1	The form element must be accessible.	The output HTML form will be compliant to WCAG 2.2 (AAA Level).	Must

FUR4-2	The form element must be ready to be dropped into the user’s website.	The output form element must be complete enough to be added into a website and work (excluding required connections relevant to the site)	Must
FUR4-3	Any “loose ends”, e.g. js/php connections etc. should be clearly indicated in the form’s comments.	Though the target audience of the tool will have HTML experience, their level of experience is not guaranteed. They may need guidance to connect the form once it is added to their site.	Should
FUR4-4	The application should provide the user with accompanying CSS.	This CSS file should contain basic styles relevant to the elements of the form. This file should conform to the accessibility requirements with room for user customisation.	Should
FUR4-5	The form element should contain information on the accessibility features.	The HTML element and CSS file should contain comments explaining the accessibility considerations.	Should
FSR2	The element output page should contain button controls.	The buttons should perform a variety of functions.	Should
FSR2-1	The element output page should contain a “Copy” button.	The button should copy the form element to the user’s clipboard.	Should
FSR2-2	The element output page should contain an “Edit” button.	The button should take the user back to the design page for the current form.	Should
FSR2-3	The element output page should contain a “New Form” button.	The button should take the user to a blank design page.	Should

FUR5	The final form element could be rendered on the page.	The form could be displayed to the user as an embed in the tool's interface.	Could
------	--	--	-------

3.5 Non-Functional Requirements

Code	Title	Description	Priority
NFSR1	The system must be a web application.	The system should be accessible as a web application through a browser.	Must
NFSR1-1	The front end of the system must be written in React.	The application must be a React application written using the create-react-app framework.	Must
NFSR1-2	The system should run client-side.	The application should not require any server-side processing. React is a client-side library so the front end will be fine, however, the business layer (form generation code) should run client-side too.	Should
NFSR1-3	The system should be hosted through GitHub Pages.	The Pages feature of GitHub allows for a web page within a project to be hosted. Provided the application runs client-side, this is how it can be hosted without paying for a dedicated server.	Should

NFSR2	The system must support desktop browsers.	The system should be completely functional through common desktop browsers; e.g. Chrome, Firefox, Edge, Safari.	Must
NFSR3	The system could support mobile devices.	The system could be scalable to different screen sizes and device capabilities.	Could

3.6 Design Summary

Well Formed is proposed as a tool which can be used to generate WCAG compliant HTML form elements. There are a number of field types available to the user when designing a form; Text, Email Address, Date, Address, Dropdown, Radio Button.

It features a React based user interface for form design and delivery, and is available as a web application through a browser.

Chapter 4

Version 1 - Core Functionality

4.1 Version Breakdown

As the central function of the application is the system for generating the form elements, and therefore what the project hinges on, it was of primary concern to get it developed first. Fig. 4.1 shows how the Version 1 fits into the final application

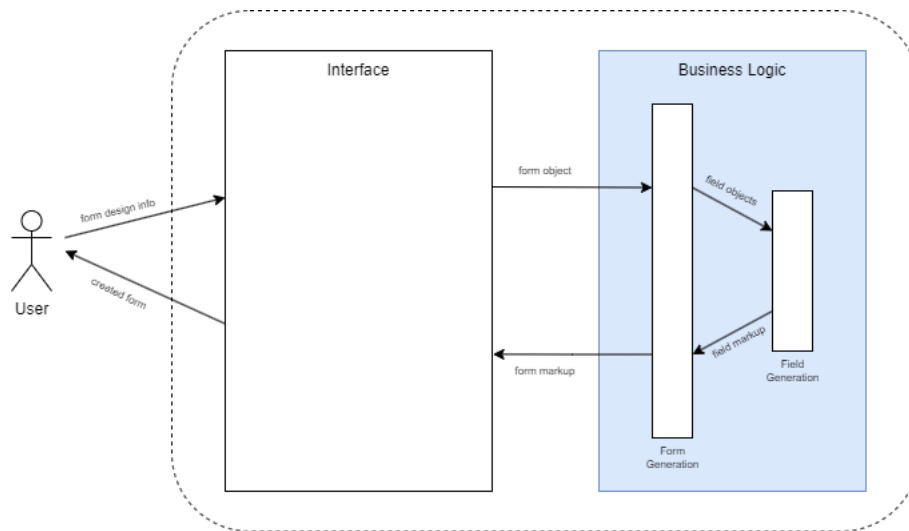


Figure 4.1: System diagram showing Version 1 section highlighted in blue.

The finished product of V1 is a simple Javascript program which takes an array of field objects and turns them into a form object. The class for

this form object implements a method which generates an HTML web form element based on the form's field objects and returns it as a string to the user. Fig. 4.2 shows a sequence diagram of this functionality.

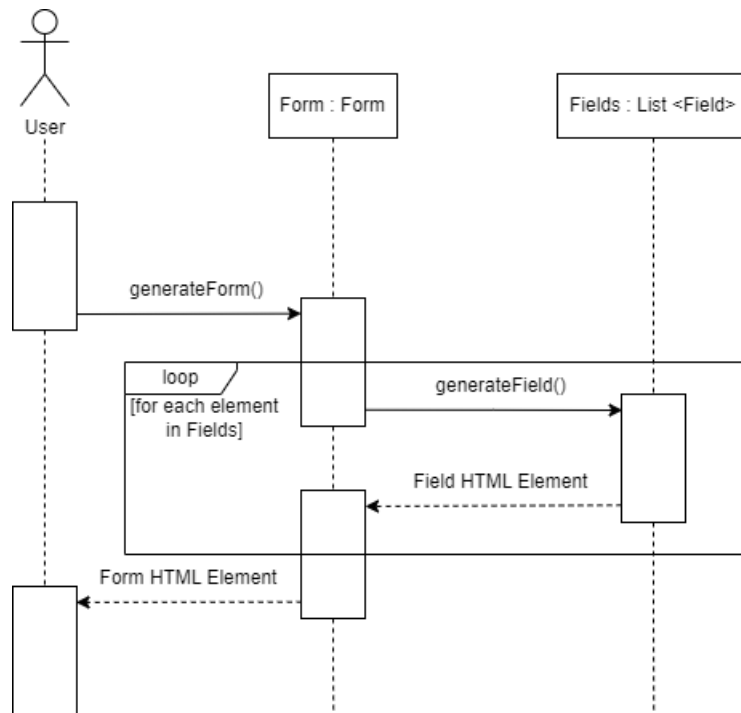


Figure 4.2: A sequence diagram for Version 1.

As stated, this is just a simple program, so does not contain any sort of interface. The program contains a couple of files to test its functionality; a script and an HTML file. The script creates an array of field object, uses this to instantiate a form object. An HTML element is generated from this and printed to the console, as well as being written to the HTML with the relevant surrounding markup, `<html>...</html>`, etc.

From here, the second version, consisting of an interface in the form of a web application, could easily have V1 integrated into its business logic.

4.2 System Design

Fig. 4.3 shows the relationship of the form to field classes. In the code itself there is no polymorphism as, due to Javascript's dynamic typing, the super-class would essentially be redundant.

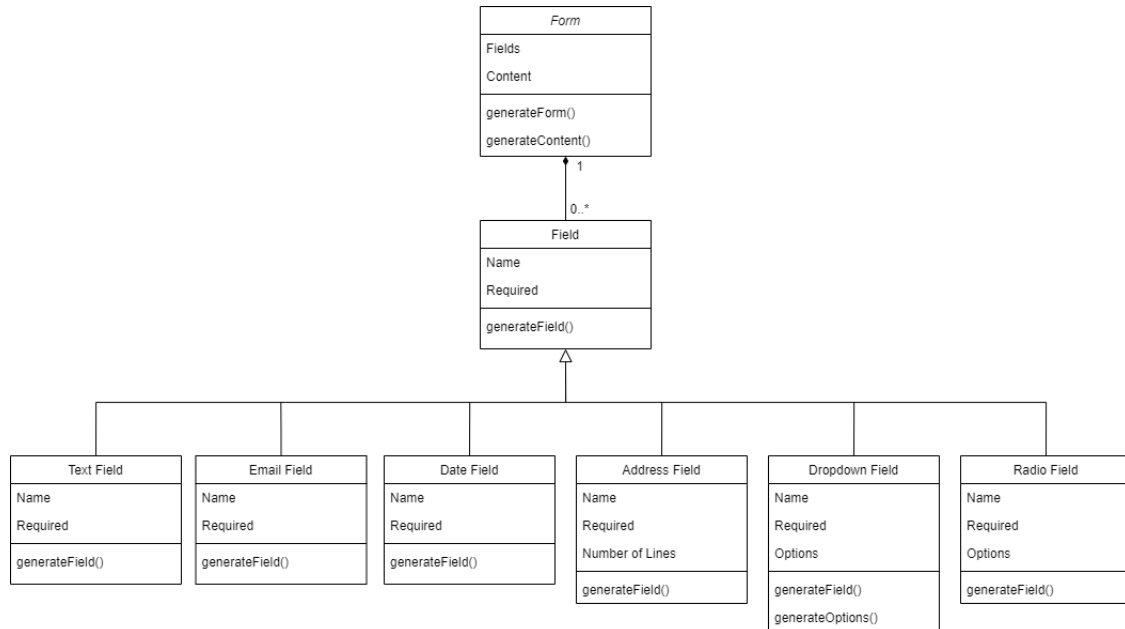


Figure 4.3: Class diagram.

Fig. 4.4 shows the relationship between files and how data flows between them when generating form elements.

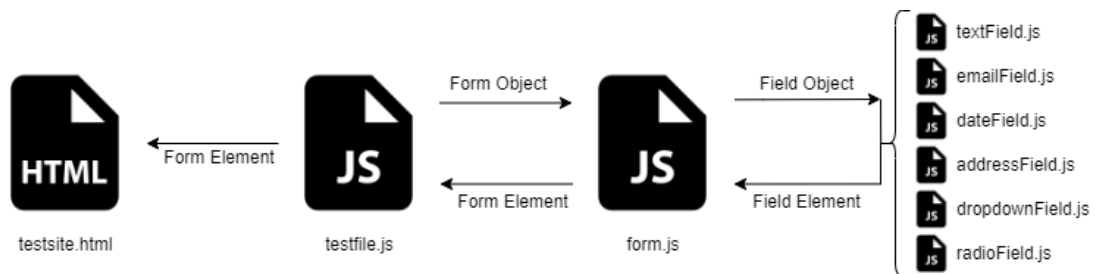


Figure 4.4: Data flow between files.

4.3 Scripting

When it came to planning out development, Javascript was chosen as the primary language for a number of reasons. The only other contender would have been Python as they are both lightweight languages, ideal for scripting. However, Javascript offers a more intuitive approach for object oriented programming, similar to more heavy duty languages such as Java or C#. Secondly, and more importantly, the plan for V2 was to develop the web application interface using React, a Javascript library, so writing 1 using Javascript would make the later integration considerably less complex.

Another potential option would have been Typescript. Typescript is a superset of Javascript which supports static typing, and would be compatible with React. From previous experience, my preference is towards statically typed languages, however, as I had never used Typescript before, and since nearly all React documentation is in Javascript, I deemed it safer to stick to the original choice.

4.4 Development

4.4.1 Initial Setup

While the React framework would not come into play until the second version, I first wanted to setup the boilerplate application so I could develop V1 with an idea of the file structure I would later be working with. Little was done with the React code at this point beyond changing the speed of the spinning logo on the default page to check my understanding of the application structure.

I then initialised a GitHub repository for code backup and version control and pushed the default React app code to it.

4.4.2 Basic Classes

Next, I defined the V1 class structure, creating classes with empty methods. Each field type has a class with a simple constructor method and a method to generate the HTML string for the field element. At this stage, this method was left empty.

Similarly, the class for the parent form element was created with a simple constructor and empty generate method.

4.4.3 Generation

The first piece of generation code to focus on was the Form element as this acts as a container for the field elements. The generateForm method creates a string containing the opening tag for the form element, then goes through each field and concatenates the generated field strings. Finally, the form element closing tag is concatenated.

As none of the field generation methods had been developed, a placeholder return was added to the text field generation method for testing. For example, a form object such as this:

```
1 new Form([new TextField("First Name"),
2           new TextField("Last Name")]);
```

Would produce this generated form:

```
1 <form>
2     First Name PLACEHOLDER
3     Last Name PLACEHOLDER
4 </form>
```

From here I started work on generation of field elements. The first were text, email and date fields as these all have a similar output; just a label and a single input field. The difference between these fields is the “type” attribute on the input element:

Text	Pure text content
Email	An email address containing an @ symbol (RegEx: /[a-zA-Z]+@[a-zA-Z]+)/)
Date	Formatted date (DD/MM/YYYY)

In theory, emails and dates can just use a simple text field, but an important stipulation in the WCAG spec is to use specific and correct markup whenever possible. Not only do these types provide some basic input validation, tools such as screen readers rely upon them to give users guidance on how to input data to the field.

Each field has a boolean “required” attribute and, when generated, this attribute is given (or not given) to the input element.

At this point I also added a submit button to the form class’s generation function, allowing a destination to be set for the form data.

If we go back to the previous example, now considering the required attribute:

```

1 new Form([new TextField("First Name", true),
2           new TextField("Last Name", true)]);

```

This output is generated¹:

```

1 <form>
2   <label for="First Name">First Name</label><span>*</span>
3   <input type="text" id="First Name" name="First Name"
4   required>
5   <label for="Last Name">Last Name</label><span>*</span>
6   <input type="text" id="Last Name" name="Last Name"
7   required>
8   <input type="submit">
9 </form>

```

The next field type I worked on was the address type. Usually these would just be made with text fields, but there are a couple of potential issues that can arise. Primarily, there could be an inclination not to give the lines labels, as shown in Fig. 4.5.

Residence Address

Figure 4.5: Address field without line labels.

While this is not an issue for most visual users as they can infer each field represents an address line, for those using descriptive technologies such as screen or braille readers these will be read as anonymous text fields. To solve this, the address field type I created generates a set number of address lines, each with a label, as shown in Fig. 4.6.

Another issue however is these labels are still not as clear as they could be. They will just read as “Address Line [X]” without any context of the

¹Class names are omitted.

Residence

Address Line 1	<input type="text"/>
Address Line 2	<input type="text"/>
Address Line 3	<input type="text"/>

Figure 4.6: Address field without line labels.

expected input (home address, work address, etc.).

The most obvious solution to this would be to add the field name to the beginning of each line label, e.g. Residence Address Line 1, Residence Address Line 2, etc. But this makes redundant visual clutter, so I decided to employ the `aria-label` attribute. This keeps the connection between the label and input elements while giving a dedicated description for assistive technologies.

For an address line in the previous example, the final HTML would look like this:

```
1 <label for="Address Line 1">Address Line 1</label>
2 <input type="text" id="Address Line 1" name="Address Line
  1" aria-label="Residence Address Line 1">
```

For required fields, only the first two lines are given the required attribute as users will rarely use every line.

The final two field types I developed were drop downs and radio button selections. These were similarly straightforward to implement, each taking a name and an array of options.

4.4.4 Completion

As V1 has no interface, a file called `testfile.js` was created to demonstrate its functionality. This file creates an example form object and generates a form element from it. This form element is printed to the console and saved (with surrounding HTML tags) to a file called `testsite.html`.

The following is the standard form generated by `test.js`. It exhibits each available field type (class names and ids are omitted):

```

1 <form>
2   <label for="First Name">First Name</label><span>*</span>
3   <input type="text" name="First Name" required>
4
5   <label for="Last Name">Last Name</label><span>*</span>
6   <input type="text" name="Last Name" required>
7
8   <label for="Email Address">Email Address</label>
9   <input type="email" name="Email Address">
10
11  <label for="Date of Birth">Date of Birth</label><span>*</span>
12  <input type="date" name="Date of Birth" required>
13
14  <label>Residence</label>
15  <label for="Residence Address Line 1">Address Line 1</label><span>*</span>
16  <input type="text" name="Residence Address Line 1" aria-label="Residence Address Line 1" required>
17  <label for="Residence Address Line 2">Address Line 2</label><span>*</span>
18  <input type="text" name="Residence Address Line 2" aria-label="Residence Address Line 2" required>
19  <label for="Residence Address Line 3">Address Line 3</label>
20  <input type="text" name="Residence Address Line 3" aria-label="Residence Address Line 3">
21
22  <label for="Preferred colour">Preferred colour</label>
23  <select name="Preferred colour">
24    <option value="red">red</option>
25    <option value="blue">blue</option>
26    <option value="green">green</option>
27  </select>
28
29  <fieldset>
30    <legend>Pick One<span aria-hidden="true">*</span></legend>
31    <input type="radio" name="Pick One" required>
32    <label for="Option 1">Option 1</label>
33    <input type="radio" name="Pick One" required>
34    <label for="Option 2">Option 2</label>
35    <input type="radio" name="Pick One" required>
36    <label for="Option 3">Option 3</label>
37  </fieldset>

```

```
38 <input type="submit">
39 </form>
```

The rendered result of this is shown in Fig. 4.7.

The rendered form contains the following elements:

- First Name***: A text input field.
- Last Name***: A text input field.
- Email Address**: A text input field.
- Date of Birth***: A date input field with a placeholder 'dd/mm/yyyy' and a calendar icon.
- Residence**: A section header.
- Address Line 1***: A text input field.
- Address Line 2***: A text input field.
- Address Line 3**: A text input field.
- Preferred colour**: A dropdown menu with 'red' selected.
- Pick One***: A radio button group with three options: 'Option 1', 'Option 2', and 'Option 3'.
- Submit**: A button.

Figure 4.7: Output form element.

4.4.5 Styling

At this stage, styling was minimal. I created a simple stylesheet defining only rules for colouring the asterisks and spacing elements on the page. The styling is present in the example form above.

4.5 Version 1 Summary

Well Formed Version 1 demonstrates the primary functionality of the application; form generation. It defines a class framework for fields and forms, and can create a full HTML form element from a form object and its associated field objects. While it does not have a proper interface, it can be used through code, almost like a library. This structure is important for V2 as it does not just extend V1, but encapsulates it completely.

Chapter 5

Version 2 - Web Application

5.1 Version Breakdown

Version 2 is a web application which encapsulates V1 as its business logic. The two primary functions of this application are to provide an interface for the user to design forms, and an interface for the system to deliver form elements to the user.

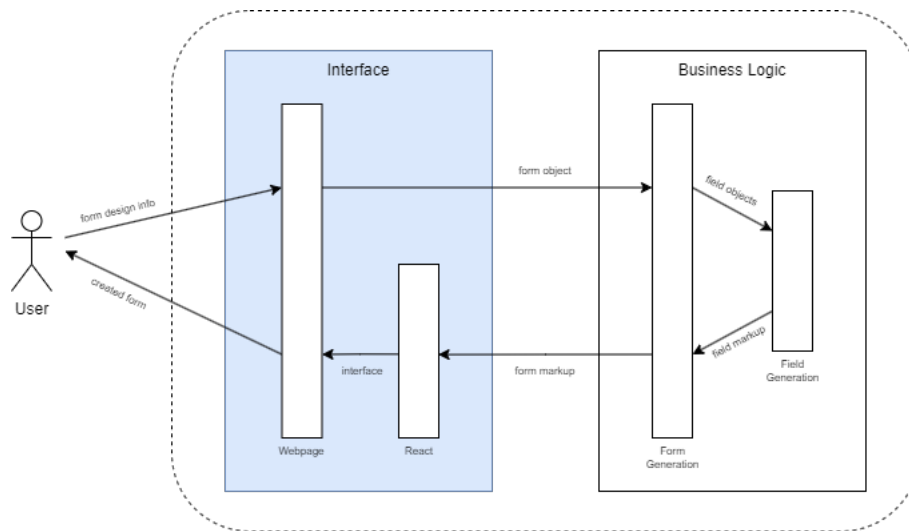


Figure 5.1: System diagram showing relevant area for Version 2.

The finished product of V2 is built using the React library for Javascript, with a simple to understand and use UI. As this project is about promoting

and assisting the development of accessibility on the web, it is important that the UI conforms to the WCAG accessibility standard.

5.2 System Design

Fig. 5.2 shows how the UI breaks down by component.

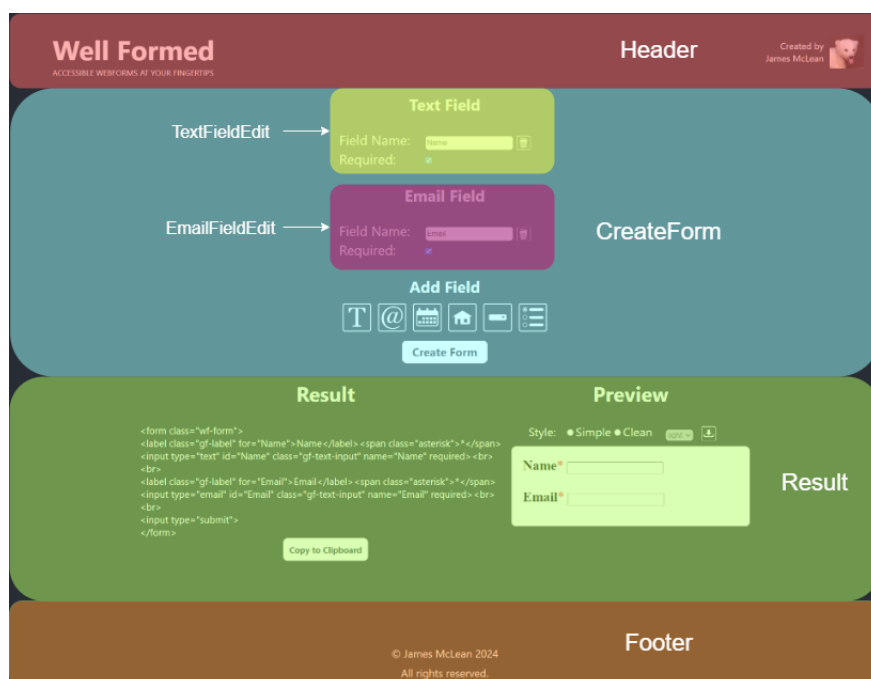


Figure 5.2: Component Breakdown.

Fig. 5.3 shows the relationships between the files for generating the application's user interface.

5.3 Library and Framework

5.3.1 React

React is a Javascript library for creating web applications and dynamic web-pages.

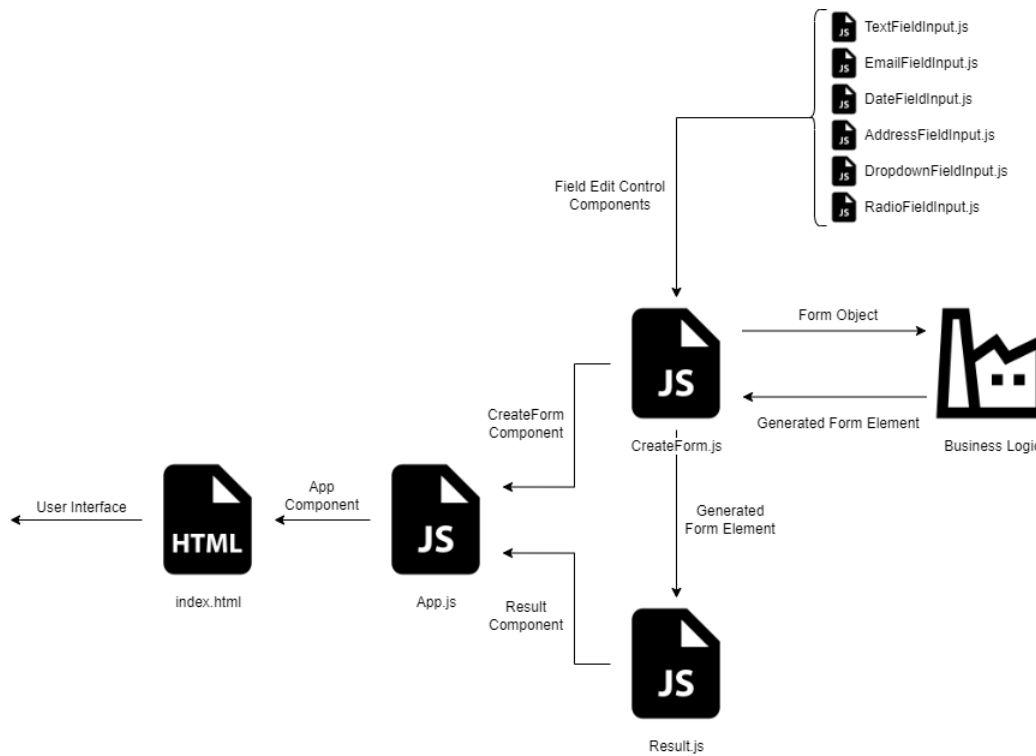


Figure 5.3: Data flow between files.

Pages are made up of components, functions which return snippets of HTML. Components can be loaded in a programmatic way, allowing for a much more effective approach to GUIs than static HTML pages. As V2 builds on V1 by encapsulating it into a web application, React was the clear choice as it is purely concerned with front end.

5.3.2 create-react-app

As React is just a library, it is highly recommended to use it in conjunction with a framework. Initially one stuck out due to its popularity; Next.js. This popularity is due to its great range of functionality, especially its support for server-side processing. However, all that was needed from the framework was a base file structure to work with, so these features were unnecessary.

After some more research, I decided on create-react-app, a very simple framework providing the basic components for an application without any

major configuration required.

5.4 Development

5.4.1 Setup and boilerplate modification

As stated in 4.4.1, the application framework was initialised before development on V1 was started, so when it came to starting on V2, the boilerplate was already in place.

The first modification I made to the default page was to make sure the front end could be connected to V1 (henceforth referred to as the “business logic”). React components are not able to access files outside their main directory, so the business logic could not be accessed as it was in its own directory. I could have moved these files into the React directory, but I wanted to keep a separation between the interface and the form generation.

To achieve this, the business logic was turned to a package which was loaded as a dependency for the React application. This way, the form generation could be treated like a library by the React components.

Once this connection had been made, I tested it by creating a “login screen” on the default page, rendering a form element with “username” and “password” fields, as shown in Fig. 5.4.

5.4.2 Create Form Area

Following the connection of the front-end to the business logic, I focused on the form design section of the application. This started with the creation of a main component, `CreateForm`, to store the parts of the interface for adding and editing fields.

This component consists of a list of field edit controls, areas for modifying field information, and “Add Field” buttons.

Initially, `CreateForm` was made with a single generic “Add Field” button which generated a basic field edit control. This basic field edit control corresponded to text fields (and would eventually be extended to email and date fields due to their identical Name + Required schema).

React components can store data in a state variable, so when the “Add Field” button is clicked, a new field edit control is added to an array in a state variable called `formFields`. Within `CreateForm`’s HTML, `formFields` is

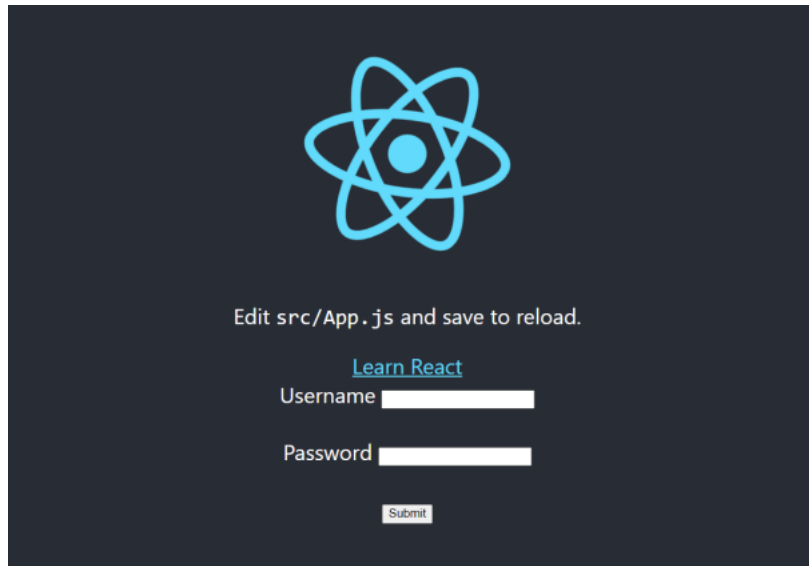


Figure 5.4: Default page with login screen.

called as an element, rendering its components (field edit controls) on the page:

```

1   {formFields}
2   <button type="button" title="Add Field" onClick={...}></
   button>

```

Fig. 5.5 shows an example of this basic version of the interface.

From here the field information needed to be tracked so it could be sent to the business logic and turned into a form element.

Preferably each field's data would be have been kept in a state variable in its respective field edit control component, with this information being collated before form generation. However, due to React's rules, as the field edit control components are held within a state variable, they cannot themselves have state variables.

To get around this, the field data is stored in a state array in the createForm component. Each field edit control component has access to this array and with the addition of an event listener, will update it with its field information whenever the user makes a change.

The added benefit of this is that fields will not register for generation unless they have been modified, meaning completely blank fields will not appear in the final form element.

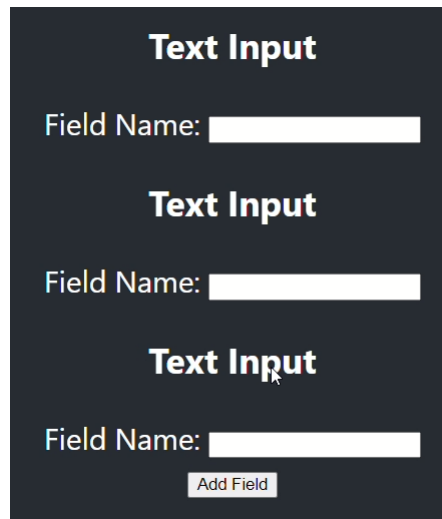


Figure 5.5: CreateForm interface with three edit controls.

I added a “state” button to the UI to print the form data state array to the console, and a “Create Form” button to generate a form and print it to the console.


Next, the main focus was developing the field edit controls. Text, Email and Date field controls were simply modified from the basic field edits, with a required option added. See Fig. 5.6. Also at this point, a delete button was added to remove a field from the saved form data and delete the field edit control component from the UI.

Following from this, Address field edit controls were developed. These build on the previous simple components by adding an input for selecting the number of address lines. Initially this was just a typed input, however as the user should select a value between 3 and 7, input validation becomes complex, requiring warnings on the UI. To solve this, the input was changed to a slider to enforce limits. See Fig. 5.7.

The final field edit components were Dropdown and Radio Button controls. These fields need a way for users to create a list of options and a couple of approaches to this were considered. Initially I had wanted to implement an extendable list of text inputs as shown in Fig. 5.8.


However, this would require the use of state variables within field edit components and as previously mentioned in regard to storing form data, this is not allowed. Instead I opted to use a basic HTML text area, with options

Text Field

Field Name: 


Required: ☐

Email Field

Field Name: 

Required: ☐

Date Field

Field Name: 

Required: ☐

Figure 5.6: Text, Email and Date Field Edit Controls.

separated by a return. There are a couple of benefits to this. Firstly, it was a lot simpler to implement, but more importantly, it is better from an accessibility perspective.

The initial extendable list approach would be considered dynamic content and therefore it would be difficult to assess its effectiveness with page description tools such as screen readers. The text area, however, is static, avoiding keyboard focus changes and a cluttered UI.


The implemented design of these controls is shown in Fig. 5.9.

5.4.3 Delivery Area

The second part of the user interface is the delivery area. This comprises two parts, a “Result” section, for showing the generated HTML form element as text, and a “Preview” section, for showing a rendering of the form in the page.

The delivery area component (called Result in the code) is loaded below the Create Form area on the interface. Within it are separate areas for the result section and the preview section.


Address Field

Field Name: 

Address Lines: 4

Required: ☐

Figure 5.7: Address Field Edit Component with slider

Field 2 

Title Required ☒ Type

Option 1

Option 2

Option 3

Figure 5.8: An early mock up showing an extendable list of option inputs.

5.4.3.1 Result Section

The result section is very simple, containing only an area for the generated HTML to be shown to the user and a button to copy the text to the user’s clipboard.

During the development of the Create Form area, the “Create Form” button would print the form element to the console, so when it came to implementing the result section all that needed to be changed was the destination of the text. The following code shows how this was achieved by finding and modifying the resultField element.

```
1 var resultField = document.getElementById("resultField");
2 resultField.innerText = newForm;
```

The “Copy to Clipboard” button runs the asynchronous function copyContent which I modified from an example on FreeCodeCamp[13].

```
1 // Copy content to user's clipboard
2 const copyContent = async () => {
3   try {
```

Figure 5.9: Edit control design for Dropdown and Radio fields.

```

4      var text = document.getElementById("resultField")
      .innerText;
5      await navigator.clipboard.writeText(text);
6      console.log("Content copied to clipboard");
7    }
8    catch (e) {
9      console.log(e);
10   }
11 }

```

Result

```

<form class="wf-form">
<label class="gf-label"
for="Name">Name</label>
<input type="text" id="Name" class="gf-
text-input" name="Name"> <br> <br>
<label class="gf-label"
for="Email">Email</label>
<input type="email" id="Email"
class="gf-text-input" name="Email">
<br> <br>
<input type="submit">
</form>

```

Copy to Clipboard

Figure 5.10: The Result Area showing a simple form element.

5.4.3.2 Preview Section

The preview section renders the generated form element in the page and provides styling options to the user.

Showing the form element on the page works similarly to how the result section shows the element's HTML, however instead of changing the inner-Text attribute of the area's div, the innerHTML attribute is set. This causes the string being passed in to be treated as pure HTML.

```
1   var previewField = document.getElementById("previewField")
   );
2   previewField.innerHTML = newForm.replace("<input type='
submit'>", "");
```

Note the removal of the submit button from the preview. This is done to avoid the page reloading if the user clicks the submit button.

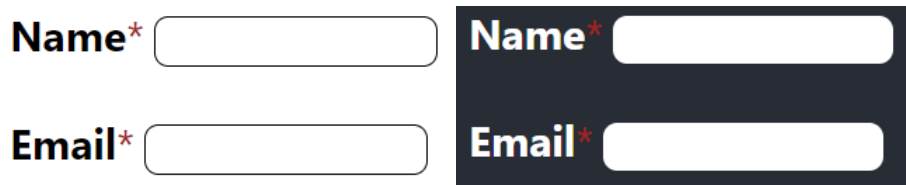
I designed two stylesheets to be available through the application. The first of these is called Simple and only applies some basic rules to be used in conjunction with pre-existing styling on a website.



A form with two input fields. The first field is labeled "Name*" in a bold, black, serif font, followed by a rectangular input box. The second field is labeled "Email*" in a bold, black, serif font, followed by a rectangular input box.

Figure 5.11: A form with Simple styling.

The other stylesheet, Clean, comes in light and dark variants. It offers heavier rules which maintain a sleeker design with a sans-serif font, rounded field input boxes and radio buttons with transitions.



Two versions of a form side-by-side. The left version is the 'light' theme, showing "Name*" and "Email*" labels in a bold, black, sans-serif font, each followed by a rounded rectangular input box. The right version is the 'dark' theme, showing the same labels in a bold, white, sans-serif font, each followed by a rounded rectangular input box, all set against a dark background.

Figure 5.12: A form with Clean styling (light and dark themes).

This styling is controlled in the preview by a menu which first allows Simple or Clean to be selected, then if Clean is selected, a dropdown is

available to select light or dark theme. The style menu also provides a button to download the selected stylesheet.

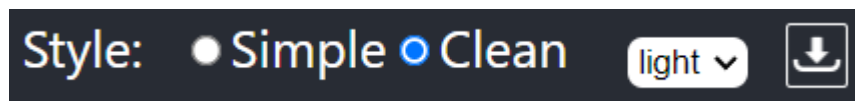


Figure 5.13: Style menu

Fig. 5.14 shows the full preview section.

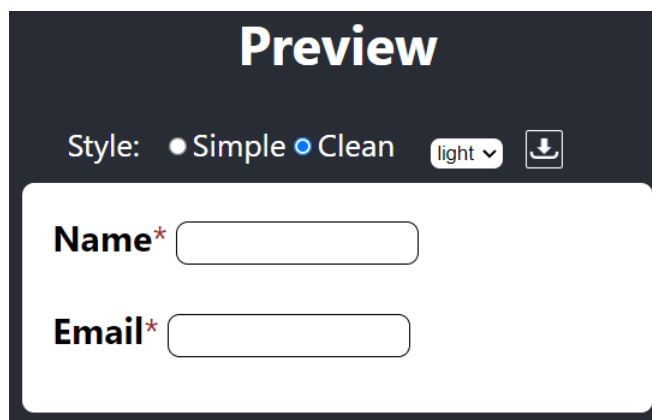


Figure 5.14: The preview section of the Delivery Area with Clean styling selected.

5.4.4 Branding and Further Visual Additions

Originally the name of the application was “Good Form” and this can still be seen in many class names in generated form elements; “gf-[class name]” etc. However, during development I decided to change this name to “Well Formed”, a similar, but more obvious play on words. This is seen in a couple of classes added later; “wf-[class name]” etc.

As the interface developed I created icons for each field with a visual representation of the field type (see Fig. 5.15). These are used for the “Add Field buttons”. I also created a delete icon for field edit controls and a download icon for the style selection menu in the preview section.



Figure 5.15: Icons for field types (Text, Email, Date, Address, Dropdown, Radio).

5.5 Version 2 Summary

Well Formed Version 2 develops the basic functional nature of V1 into a complete web application. It adds a design interface allowing the user to create a form in an intuitive way, and a display interface providing styling options, so the user can refine their form. The use of React allows the application to be dynamic, meaning the flow from design to refinement to final product is natural.

Chapter 6

Testing and Design Refinements

6.1 Version 1 Testing

The functionality testing for V1 was an uncomplicated process. Due to the linear nature of the system, it was a question of testing each field type. As this part of the project will only ever receive properly formatted objects as an input, and since most testing of this nature had been done informally through the development, there was not much need for a formal test beyond generating a form with each field type.

I used the previously mentioned `testFile.js` (see 4.4.4) and its resulting webpage, `testsite.html`, to create this form, which was then used for accessibility testing.

To assess the accessibility of the produced forms, I manually tested a form against a number of test cases I had devised based on the relevant WCAG guidelines. After this I ran the Site Improve extension on the page. This browser extension automatically picks up any accessibility compliance errors as well as best practice issues.

6.1.1 Version 1 Manual Accessibility Testing

Primarily, the accessibility testing for V1 consisted of formally written test cases for the generated form element. These tests were split into three categories:

- *Keyboard* - The navigation of elements must be intuitive. [4 tests]
- *Screen Reader* - The screen reader must describe elements in a clear way. [6 tests]
- *Colour* - The colour of elements must be visually clear to all sighted users. [3 tests]

Of these test cases, only two were not met from the outset. This in itself is a testament to the benefits of the accessibility by design approach, especially as both of these issues were screen reader problems which are difficult to predict during development.

The first of these failed tests was *FT14 - Address fields are described with the name of the field before each line and the correct line number*. The problem occurring was that the screen reader would not read the number of the address line. The fix for this was as simple as moving a misplaced closing quote.

The second test fail required more in depth modification to the system. This was for *FT15 - Radio fields have their labels read when entering, each option is clearly read with its index in the list*. The issue that arose was that the radio buttons would not read the name of the field and just the button labels. This is because of how radio elements are grouped together; the group is treated as a single field, but the label given this field was ignored by the screen reader. After some research I discovered that in order to have the screen reader describe the field with its label correctly, the radio buttons needed to be surrounded in a fieldset element.

This shows a problem with many web development resources as the importance of fieldset elements is rarely highlighted. Neither W3Schools[14] nor GeeksforGeeks[15], both popular code education websites, mention fieldsets on their HTML Radio Button pages. MDN Web Docs[16] does contain them in code examples, but does not underline their function.

6.1.2 Version 1 Extension Accessibility Testing

After manually testing the accessibility of the generated form components, I ran the Site Improve browser extension (see 6.1) on the testsite page. This raised one minor colour problem I had initially missed. The red colour of the asterisks for required fields did not meet the required colour ratio on the Clean styling with dark theme selected. For normal text the contrast ratio should be at least 7:1 for AAA WCAG compliance (Guideline 1.4.6), but

for white or dark grey/black backgrounds, this is difficult to achieve with primary colors such as red.

However, even though the asterisks are text based elements, they are not necessarily textual content. As they are symbolic visual indicators rather than readable text, they can be considered graphical user interface components, therefore only need a colour contrast ratio of 3:1 (Guideline 1.4.6).

6.2 Version 2 Testing

Testing for V2 was considerably more involved than that of V1. This is due to the different approach to accessibility. As previously stated, the form elements from V1 were accessible by design, meaning they really only needed a couple of touch ups. V2 by contrast was designed with accessibility as a secondary concern. This was due to my lack of experience with React pushing the focus towards just getting the UI working, but it highlights the difference in effectiveness between the approaches.

As with the V1, there was little requirement for formal input testing, so the focus of V2's testing was WCAG compliance of the user interface.

6.2.1 Version 2 Manual Accessibility Testing

The manual testing for V2 as with previous, was primarily focused around formally written test cases, this time for the UI, rather than the system output. Again, these were split into the same three categories (Keyboard [5 tests], Screen Reader [9 tests], Colour [2 tests]).

However, unlike the V1 testing, many more issues were raised. Of the sixteen test cases, only six passed first time, two required minor tweaks, and the remaining eight required full fixes.

6.2.1.1 Minor Issues and Tweaks

The minor issues were primarily to do with how the screen reader describes buttons. Firstly, *AT17 - Each "add field" button is clearly described*, had result the of buttons being described as graphic buttons with image descriptions. This is due to the buttons' icons being treated as images within the button elements. As the icons are redundant for non-visual users, this is just

confusing. To sort this, I gave the images the aria-hidden role, so accessible technologies ignore them and treat the buttons normally.

The second tweak, *AT18 - The preview theme selection fields are described clearly*, was a similar problem as the download button had the same issue with the image causing a confusing description. As well as this, the radio buttons for selecting the style had the same issue as the radio buttons in V1 (ignoring the field name and just reading the option labels). Like with V1, this was a case of putting the options in a fieldset and making some CSS adjustments.

6.2.1.2 Major Issues and Fixes

Major issues were found with the following test cases

- *AT04* - The first element on a field edit is focused on creation
- *AT05* - The result area is a focusable element
- *AT11* - All elements in text field edits are clearly described
- *AT12* - All elements in email field edits are clearly described
- *AT13* - All elements in date field edits are clearly described
- *AT14* - All elements in address field edits are clearly described
- *AT15* - All elements in dropdown field edits are clearly described
- *AT16* - All elements in radio field edits are clearly described

The following table shows the issues, and their relevant solution:

Test Case	Issue	Fix
AT04	Focus remains on the create field button	Created a set focus function to find and focus on the first input of the field edit
AT05	Element is not focusable	Added tabindex attribute to element
AT11	Inputs have no description	Connected labels and added relevant ARIA attributes
AT12		
AT13		
AT14		
AT15		
AT16		

The first two issues concern keyboard focus. The fix for *AT04* was to create a function which would be called as part of the click functionality of the create field buttons. This function finds the first input element of the newly loaded field edit component (the “Field Name” input) in the DOM, and overrides the current keyboard focus to it.

For *AT05*, the element containing HTML code for the generated form must be focusable. Generally the only focusable elements are ones with some sort of interaction, but due to this being a dynamic piece of text, I decided it was important for it to be part of the keyboard navigation.

The remaining issues were due to missing label connections on the field edit controls. This is not difficult to fix, but it took time as it needed to be sorted for each field control.

6.2.2 Version 2 Extension Testing

As with V1, I ran the Site Improve browser extension over V2 to see if it could pick up any imperfections missed by the manual testing. The extension flagged three issues (see Fig. 6.1).

Of these three issues, only one concerns a WCAG guideline (1.4.8 Visual Presentation). In the footer of the application (licensing and attribution information) the text does not have a compliant line height. To fix this, I added a vertical spacing CSS rule.

The other problems highlighted are to do with best practice rather than any guideline infringement. Firstly, the headings in the page do not follow the correct hierarchical flow; h1 followed by h2 etc. As this does not affect the function or aesthetic value of the application I decided to ignore it.

The second best practice issue focuses on the lack of ARIA landmarks. These are attributes to break the page into sections, however they are quite prescriptive. There is a set of recognised landmark roles which are effective for a traditionally structured webpage. In the case of Well Formed, these landmark roles do not correlate to the separations in the interface well. One of the key tenets of the WAI-ARIA is that no use of ARIA is better than bad use of ARIA, so I decided it best not to try to add these attributes.

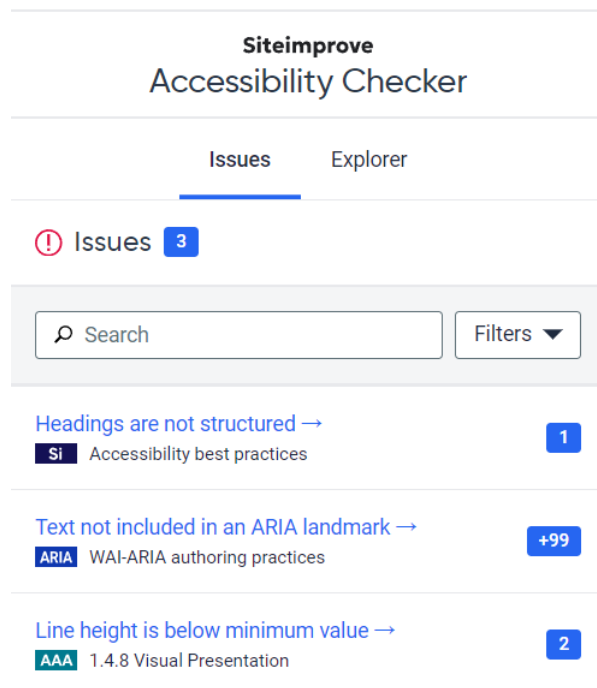


Figure 6.1: The Site Improve browser extension.

6.3 Testing Summary

The difference in testing for each version shows the relative merits and drawbacks of two approaches. Firstly, the accessibility by design approach of V1 led to much faster testing with many fewer problems to fix. Conversely, the retrospective approach employed for V2, with accessibility features being added after the functionality of the application had been developed, led to a much longer testing period with many more issues to solve.

I feel this really demonstrates the value of an application such as Well Formed. It provides an element guaranteed to meet WCAG compliance, cutting out the requirement for extensive fixes on a full page.

Chapter 7

Evaluation

Beyond the development testing, the evaluation of the application would be trialling it with users. Ideally this would have to be done with a range of users from a range of communities, e.g. blind, colour blind and vision impaired users; deaf users; users with motor impairments; etc. to evaluate the effectiveness of created forms in a real world context. This however, would require a great deal more time than was available so was beyond the scope of the project.

However, for the application itself, the primary target is developers. In order to get valuable feedback, those testing the system would have to be regularly involved with web development, but not necessarily very familiar with the WCAG standard. A meeting with members of the Heriot-Watt Web Applications Team was arranged for demonstrating the Well Formed application and discussing its effectiveness in an applied context.

Before the meeting I was informed that the team employs a post-hoc approach to accessibility, using the Siteimprove browser extension¹ to check pages for errors during development. They also subscribe to the Siteimprove Web Governance Service which regularly checks deployed pages for accessibility issues, broken links, typos, etc.

7.1 Evaluation Description

The evaluation consisted of:

¹This informed my choice of Site Improve for the extension based testing in Chapter 6

- A presentation explaining what Well Formed is, its approach and how to use it (slides provided in Appendix D).
- An example bad form with issues that would not be picked up by just an extension compared with a form made using Well Formed.
- A demonstration of the application allowing the participants to get proper use of the system by making forms themselves.
- Discussion about the application's effectiveness.

The evaluation took place in a meeting room in Heriot-Watt University on 28th April 2024, lasting around an hour. Myself, two members of the Heriot-Watt Web Applications team, and my supervisor, Andrew Ireland, were present.

7.2 Evaluation Results

7.2.1 Presentation

The presentation consisted of:

- A description of the application itself, explaining what it is and the context in which it should be used.
- A description of the accessible by design approach.
- A user guide explaining:
 - Form Design
 - Each field type
 - Form delivery
 - Form Styling

The presentation went well, with neither participant asking for clarification on anything. This was full introduction to the application, and considering the presentation was only eleven slides long, it shows the system is intuitive and easy to understand.

7.2.2 Bad Form Example

The bad form example I had devised contained three errors:

- A typo causing a label and relative field to be disconnected - Picked up by Site Improve Extension

- An address field where each line only has a “Line [x]” label - Semantic issue not picked up by Site Improve but avoided with Well Formed
- A radio field where only the option labels are read by the screen reader and the main field label is ignored - Screen reader issue not picked up by Site Improve but avoided with Well Formed

After going through the bad example form, I showed a similar form made with Well Formed and demonstrated, both with the Site Improve Extension and with a screen reader how these issues were avoided.

7.2.3 Demonstration

As the presentation only showed the interface in sections, I started the demonstration by giving a proper overview of the interface in full. I then passed the application over to the participants to try.

After some discussion about the icons for each field type, they began to create a form which they continued to tweak and modify throughout the session. They tried every aspect of the application, from creating a form and styling it, to testing the clipboard and download capabilities. The form created is shown in Fig. 7.1.

7.2.4 Discussion

One of the first points brought up was the potential to use the system as a basis for creating React components. We talked about how the form generation section of the system could be reworked to act as a library which could be used in conjunction with React to create forms as components. This would allow forms to be programmatically designed by an application and generated by Well Formed.

From this we discussed the potential for branching forms, i.e. forms which change based on previous answers. This is not something I had considered when designing Well Formed and could be an interesting expansion for the application’s capabilities. It would require an overhaul of the design interface as well as the necessary changes to the form generation code.


The participants asked about how the application would be kept up to date as the WCAG Standard is updated. This would require me to modify the system when a new version of the standard is released, however as form guidelines have remained fairly unchanged since WCAG 2.0 (2008) if an update to the application is required, it is unlikely to be very large.

Programme title*


GCM code*

Campus*

- ☐ Edinburgh
- ☐ Borders
- ☒ Orkney
- ☐ Dubai
- ☐ Malaysia

Entry date* 

Programme leader

Year 

Heriot-Watt email address

Figure 7.1: Form created by participant.

I was also asked about potential new fields which could be added in future. I talked about how I had missed out checkbox fields until too late in development. This would be a simple field to implement, just needing a field name.

As the application is full usable with just keyboard inputs, an interesting question posed was whether a form could be created with just a mouse. All controls are clickable and the any typed inputs should be accessible via an on-screen keyboard.

Another question raised was whether fields could be reordered using the designer. This would be a useful feature, though finding an accessible way to achieve this would be a challenge as it would be very dynamic content.

Overall the participants seemed impressed by the application, stating it went further than they would generally consider when applying accessibility features. The use of the screen reader to test the forms was something they pointed out as being outside their approach, so the elimination of this aspect

added even more to the application's effectiveness.

I feel satisfied that Well Formed meets the intended expectations for a system of its kind, and I am pleased with the feedback I received. It shows the application has great potential for further expansion.

Chapter 8

Conclusion

8.1 Requirements Conformance

This section revisits the requirements defined in 3.4 and 3.5, showing to what degree they were met. Each requirement is colour-coded: green for fully met; amber for partially met; red for not met. As well as this, requirements made redundant by design changes are coloured grey.

ID	Title	Description	Conformance Notes
FUR1	The user must be able to add fields to a form.	On the form designer page the user will be able to click “+” button to add a new field to the form. A new field edit component will be added to the form designer.	There is not a generic “+” button; each field type has its own button to avoid the page being overly dynamic for keyboard users.
FUR1-1	The user must be able to enter the name of the field.	Each field edit component will have a name input where the user will edit the field’s name	

FUR1-2	The user must be able to select from a list of field types.	Each field edit component will have a drop-down menu containing the various field types for the user to choose from. The available types will be: Text, Date, Dropdown, Radio, Address, Email.	The user can select a field type but with unique “Add Field” buttons, rather than a dropdown within the field edit component itself.
FUR1-3	The field edit component must display different input controls for different types.	After the user has selected a field type, the edit component should provide the relevant input components to the field, e.g. selection options for radio/drop-down fields.	
FUR1-4	The user must be able to set a field as “Required”.	Each field edit component will have a checkbox allowing the user to set the field to require an input.	
FUR1-5	The tool should provide naming guidance.	The WCAG spec recommends what to avoid when naming fields, this guidance should be presented to the user [perhaps with a popup on the name field].	It was difficult to find a way to do this that would keep the interface clean, while also avoiding making the page too complicated for non-sighted users.
FSR1	The system should have a Start Page.	The application should open to a start page to give some information on the tool before proceeding to allow the user to design a form.	This is something I simply did not get around to making. I gave it too high a priority, it should have been a “could” requirement.
FUR2	The user must be able to remove fields from a form.	The field edit component will contain a “delete” button the user can click to remove the field from the form.	

FUR3	The user could be able to reposition fields in a form.	Each form component could have a control option to drag the component up and down the list. For keyboard users this could be achieved through selecting the move control and using the up and down arrow keys.	The design area took a long time to get working with statically positioned field edit components, so achieving this in a way that would be usable with a mouse or a just a keyboard would have taken too much time. I would like to find a way to add this in future as I believe it could be very useful.
FUR4	The application must provide the user with a form element when they finish designing.	The primary output of the tool is an HTML form element using the design specified by the user.	
FUR4-1	The form element must be accessible.	The output HTML form will be compliant to WCAG 2.2 (AAA Level).	
FUR4-2	The form element must be ready to be dropped into the user's website.	The output form element must be complete enough to be added into a website and work (excluding required connections relevant to the site)	
FUR4-3	Any "loose ends", e.g. js/php connections etc. should be clearly indicated in the form's comments.	Though the target audience of the tool will have HTML experience, their level of experience is not guaranteed. They may need guidance to connect the form once it is added to their site.	I decided this would not be especially helpful for most users as there are a number of ways they could want to treat form data.
FUR4-4	The application should provide the user with accompanying CSS.	This CSS file should contain basic styles relevant to the elements of the form. This file should conform to the accessibility requirements with room for user customisation.	

FUR4-5	The form element should contain information on the accessibility features.	The HTML element and CSS file should contain comments explaining the accessibility considerations.	A single comment is provided at the top of the element which references the relevant guidelines.
FSR2	The element output page should contain button controls.	The buttons should perform a variety of functions.	
FSR2-1	The element output page should contain a “Copy” button.	The button should copy the form element to the user’s clipboard.	
FSR2-2	The element output page should contain an “Edit” button.	The button should take the user back to the design page for the current form.	The final application has the design and delivery sections on the same interface so a form can be modified and reloaded without having to switch pages.
FSR2-3	The element output page should contain a “New Form” button.	The button should take the user to a blank design page.	As the design and delivery are on the same page so this is somewhat redundant in its current form. The page can just be reloaded in the browser to start with a blank form, but this functionality could be available through a “Clear Form” button in future.
FUR5	The final form element could be rendered on the page.	The form could be displayed to the user as an embed in the tool’s interface.	

8.2 Non-Functional Requirements

Code	Title	Description	Conformance Notes
NFSR1	The system must be a web application.	The system should be accessible as a web application through a browser.	
NFSR1-1	The front end of the system must be written in React.	The application must be a React application written using the create-react-app framework.	
NFSR1-2	The system should run client-side.	The application should not require any server-side processing. React is a client-side library so the front end will be fine, however, the business layer (form generation code) should run client-side too.	
NFSR1-3	The system should be hosted through GitHub Pages.	The Pages feature of GitHub allows for a web page within a project to be hosted. Provided the application runs client-side, this is how it can be hosted without paying for a dedicated server.	For my demonstrations of the system during the project, I was able to just use a localhost server on my own machine.
NFSR2	The system must support desktop browsers.	The system should be completely functional through common desktop browsers; e.g. Chrome, Firefox, Edge, Safari.	
NFSR3	The system could support mobile devices.	The system could be scalable to different screen sizes and device capabilities.	In theory the application is functional on mobile devices, though it has not been optimised for smaller screen sizes.

8.3 Future Work

There are a number of directions the project could be taken in future. From a basic functional standpoint, there are two possible extensions that could be made to form generation. Firstly adding more field types. The main field types are all covered bar one; check boxes. This would be simple to implement, but had missed my consideration until after I had finished on V1, and wanting to push on with V2, I decided it best not to go back. Beyond this, there is scope for more complex custom fields similar to the address field.

Another expansion to the form generation, mentioned in 7.2.4, would be that of branching forms. These are forms where inputs to fields will affect the available fields further down the form. Not only would this require changes to the form generation code, the design interface would also need to be redesigned to accommodate this more complex form type. The branching paths of forms could be defined using a flowchart system, either directly integrated into the existing UI, or adjacent to it.

The integration of Well Formed into a web framework such as React was another point of discussion during the evaluation session (7.2.4). This would be an extension of V1 rather than the full application, adding an interface for it to be used as a library to return forms as page components. This could allow developers to call a form within another component, specifying field information and desired styling options.

At the more ambitious end of the spectrum, Artificial Intelligence could be employed for detecting semantic problems. This could be effective in avoiding ambiguity in field names, suggesting more descriptive alternatives. If more complex field types are added, AI could be used to advise on the appropriateness of a field type, not just in relation to the given field name, but also based on the context of the rest of the form.

Appendix A

Testing Log

The following is the testing log detailing the test cases and fixes covered in Chapter 6.

Form Testing

Keyboard Navigation

Test Case	Expected Behaviour	Initially Met	Eventually Met
FT01	All relevant elements are focusable	✓	n/a
FT02	Tab order naturally follows the order of elements	✓	n/a
FT03	Radio options are navigable separately from the rest of the form (i.e. navigable without moving to surrounding fields)	✓	n/a
FT04	Dropdown selections are navigable separately from the rest of the form (i.e. navigable without moving to surrounding fields)	✓	n/a

Screen Reader

Test Case	Expected Behaviour	Initially Met	Eventually Met
FT11	Text fields are clearly described	✓	n/a
FT12	Email Address fields are clearly described	✓	n/a
FT13	Date fields are clearly described	✓	n/a
FT14	Address fields are described with the name of the field before each line and the correct line number	□	✓
FT15	Radio fields have their labels read when entering, each option is clearly read with its index in the list	□	✓
FT16	Dropdown select fields have their labels read when entering, each option is clearly read with its index in the list	✓	n/a

Colour

Test Case	Expected Behaviour	Initially Met	Eventually Met
FT21	Simple styling has a text-background colour ratio of at least 7:1	✓	n/a
FT22	Light Clean styling has a text-background	✓	n/a

	colour ratio of at least 7:1		
FT23	Dark Clean styling has a text-background colour ratio of at least 7:1	✓	n/a

Issues

- FT14 - Address line number is not read
 - Fix - misplaced closing quote
- FT15 - Label is not read when entering field
 - Fix - changed how radio fields are generated
 - Now the options are within a fieldset element
 - This has a legend instead of a label
 - Made the appropriate CSS updates

Application Testing

Keyboard Navigation

Test Case	Expected Behaviour	Initially Met	Eventually Met
AT01	All relevant elements are focusable	✓	n/a
AT02	Tab order naturally follows the order of elements	✓	n/a
AT03	All edits are intractable correctly	✓	n/a
AT04	The first element on a field edit is focussed on creation	□	✓
AT05	The result area is a focusable element	□	✓

Screen Reader

Test Case	Expected Behaviour	Initially Met	Eventually Met
AT11	All elements in text field edits are clearly described	□	✓
AT12	All elements in email field edits are clearly described	□	✓
AT13	All elements in date field edits are clearly described	□	✓
AT14	All elements in address field edits are clearly described	□	✓

AT15	All elements in dropdown field edits are clearly described	<input type="checkbox"/>	✓
AT16	All elements in radio field edits are clearly described	<input type="checkbox"/>	✓
AT17	Each “add field” button is clearly described	✓	✓
AT18	The preview theme selection fields are described clearly	✓	✓
AT19	The rendered preview form retains the accessibility features it is created with (see Form Testing above)	✓	n/a

Colour

Test Case	Expected Behaviour	Initially Met	Eventually Met
AT21	Main interface has a foreground-background colour ratio of at least 7:1	✓	n/a
AT22	Preview area has a foreground-background colour ratio of at least 7:1	✓	n/a

Tweaks

- AT17 - Buttons are read as graphic buttons with image descriptions (unnecessary and complicating); this area being in a header makes the screen reader read it as a banner region (also unnecessary as regions are not relevant)
 - Added aria-hidden to img elements
 - Changed the area from a header element to a div element
- AT18 - Radio buttons are read, but not with proper label; download button is just described as “Download” (and also as a graphic button); theme select has no description
 - Put radio buttons into a fieldset element
 - Did all the necessary CSS modification to hide the legend etc.
 - Gave download button an aria label and made the img aria hidden
 - Added an aria-label to theme select field

Issues

- AT05 - Element is not focusable
 - Fix - Added tabindex attribute to element
- AT04 - Focus remains on the create field button
 - Fix - Created a set focus function
 - This function finds the first element and sets the focus
 - It is called when the element loads
- AT11 - Inputs have no description
 - Connected labels and added relevant ARIA attributes

- AT12 - Inputs have no description
 - Connected labels and added relevant ARIA attributes
- AT13 - Inputs have no description
 - Connected labels and added relevant ARIA attributes
- AT14 - Inputs have no description
 - Connected labels and added relevant ARIA attributes
- AT15 - Inputs have no description
 - Connected labels and added relevant ARIA attributes
- AT16 - Inputs have no description
 - Connected labels and added relevant ARIA attributes

Appendix B

PLES

B.1 Social Implications

My project concerns an area which targets potentially vulnerable users. Careful considerations have been made in regard to language used throughout documents as well as the social implications of the project. This project is intended to have a positive impact on the development of accessibility on the web.

B.2 Consent Form

This is a copy of the consent form which was prepared for an interview with Jim McCafferty on his experiences as a transcriber and proofreader for the Scottish Braille Press, as well as his experiences with digital systems as a vision impaired user.

A Generator for Accessible HTML Forms

Study Consent Form

The importance of accessibility in computer systems cannot be understated. Errors in a system's accessibility can cause the system to be useless at best and actively detrimental at worst. This is why hearing the insights of users of these systems is key in developing them in an effective way. The purpose of this study will be to discuss the requirements and potential pitfalls of computer accessibility, with the view of gaining background knowledge for the development of an accessible web form generation tool.

The activity will comprise of an interview with the participant, the purpose of which will be to:

- Discuss the participant's experience with using various accessible computer interaction systems.
- Discuss the participant's expertise working in the wider field of accessibility.
- Gain background knowledge to help with design considerations for the final system.

By signing this form, I assert:

- I am willing to take part in the study and know I can withdraw at any time.
- I consent to having the information they provide recorded and used in the final submission.
- I confirm I do not have any health issues that will prevent me from providing informed consent.
- I am happy to be named in the final report: Yes/No (Delete as appropriate)

Participant's Name	Participant's Signature

Researcher's Name	Researcher's Signature

Appendix C

User Guide

This is the user guide provided through the README file in Well Formed project.

Well Formed - Accessible Webforms at your fingertips

James McLean

What is Well Formed

Well Formed is an application for generating accessible web forms. It features an easy to use interface to design forms and produces HTML form elements ready to be dropped into your web page. These elements are WCAG AAA compliant so will be compatible with screen readers and other descriptive accessibility tools.

Set up

Prerequisites:

- Node.js

Download the application folder and navigate to it in the file explorer or open it in a development application such as VSCode. Open a terminal window and run:

```
npm start
```

Open a browser tab and navigate to localhost:3000.

Using the application

Add a field to a form by clicking one of the add field icons. The field types available are:

- Text
- Email Address
- Date
- Address
- Dropdown
- Radio Select

Once the form is completed, click create form to generate the accessible form element from your design. The element's markup will be presented along with a rendering of the form in the page. From here styling can be selected. There are two available styles:

- Simple - basic styling which should not conflict with preexisting CSS rules.
- Clean - heavier styling to give forms a sleeker clean look (available in light and dark).

These style sheets can be downloaded from the selection menu.

Appendix D

Presentation

This is the presentation I produced for the demonstration session with the Heriot-Watt Web Applications Team.

Well Formed

ACCESSIBLE WEBFORMS AT YOUR FINGERTIPS

James McLean

What is Well Formed?

- Application for creating accessible web forms
- Easy to use interface
- Drop into a webpage
- WGAG 2.2 compliant to Level AAA

Approach

- Accessible by design
- Avoid lengthy accessibility testing
- Room for customisation

Form Design

- Six field types
 - Text
 - Email Address
 - Date
 - Address
 - Dropdown
 - Radio Selection



Simple Field Types


- Text
 - Basic text field
- Email Address
 - Input requiring “@”
- Date
 - Enforced dd/mm/yyyy format
 - Browser will likely provide calendar UI

Field Name: 

Required: ☐


Name


Email

Date of Birth 

Address Field

- Select number of address lines
 - 3-7 Lines
 - First two will be required
- Accessibility
 - Enforced line numbers
 - Screen reader identifies field name and line number

Field Name: 

Address Lines:  4


Required: ☐

Residence

Address Line 1 *	<input type="text"/>
Address Line 2 *	<input type="text"/>
Address Line 3	<input type="text"/>
Address Line 4	<input type="text"/>

Selection Fields

- Two types
 - Dropdown
 - Radio button
- List options in text field
- Accessibility
 - Radio button employs fieldset

Field Name: 

Field Options:

Required: ☐

Preferred colour

Pick One*

☐ Option 1

☐ Option 2

☐ Option 3

red
blue
green

Result

- Generated markup is displayed
- This can be copied to clipboard

```
<form class="wf-form">
<!-- This accessible form element was created using Well Formed
It is WCAG 2.2 compliant to AAA standard, meeting the following success criteria in particular:
1.3.5 - Identify Input Purpose (AA)
1.4.6 - Contrast Enhanced (AAA)
2.1.2 - No Keyboard Trap (A)
2.1.3 - Keyboard (No Exception) (AAA)
2.4.3 - Focus Order (A)
2.4.6 - Headings and Labels (AA)
2.4.7 - Focus Visible (AA)
2.5.3 - Label in Name (A)
Well Formed © James McLean 2024
All rights reserved.-->
<label class="gf-label" for="Name">Name</label> <span class="asterisk">*</span>
<input type="text" id="Name" class="gf-text-input" name="Name" required> <br> <br>
<label class="gf-label" for="Email">Email</label> <span class="asterisk">*</span>
<input type="email" id="Email" class="gf-text-input" name="Email" required> <br> <br>
<label class="gf-label" for="Date of Birth">Date of Birth</label>
<input type="date" id="Date of Birth" class="gf-text-input" name="Date of Birth"> <br> <br>
<input type="submit">
</form>
```

Copy to Clipboard


Simple Theme

- Applies basic rules
 - Label and field positioning
 - Asterisk colouring and positioning
- Leaves room for outside styling

Style: ☒ Simple ☐ Clean light ▾ 

Name*

Email*

Date of Birth 

Clean

- Two themes
- Applies a more polished look
- Keeps at least 7:1 colour ratio

Light

Style: ☐ Simple ☒ Clean light 

Name*

Email*

Date of Birth 

Dark

Style: ☐ Simple ☒ Clean dark 

Name*

Email*

Date of Birth 

Demo and Questions

Bibliography

- [1] W3C/edX. *Introduction to Web Accessibility - Foundations Course*. URL: <https://learning.edx.org/course/course-v1:W3Cx+WAI0.1x+3T2019/home>. (accessed: 25.05.2023).
- [2] W3C. *Web Content Accessibility Guidelines 1.0*. URL: <http://www.w3.org/TR/WAI-WEBCONTENT>. (accessed: 25.09.2023).
- [3] Nick Awad. *Influencing the Trends of Digital Inclusion: An Interview with Jonathan Hassell*. 2023. URL: <https://www.accessibility.com/blog/influencing-the-trends-of-digital-inclusion-an-interview-with-jonathan-hassell>. (accessed: 14.11.2023).
- [4] W3C. *Web Content Accessibility Guidelines 2.0*. URL: <https://www.w3.org/TR/WCAG20/>. (accessed: 02.10.2023).
- [5] W3C. *Web Content Accessibility Guidelines 2.1*. URL: <https://www.w3.org/TR/WCAG21/>. (accessed: 18.11.2023).
- [6] W3C. *What's New in WCAG 2.1*. URL: <https://www.w3.org/WAI/standards-guidelines/wcag/new-in-21/>. (accessed: 18.11.2023).
- [7] W3C. *Web Content Accessibility Guidelines 2.2*. URL: <https://www.w3.org/TR/WCAG22/>. (accessed: 19.11.2023).
- [8] W3C. *What's New in WCAG 2.2*. URL: <https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/>. (accessed: 19.11.2023).
- [9] The DAISY Consortium/NISO. *ANSI/NISO Z39.86-2005 (R2012) Specifications for the Digital Talking Book*. URL: <https://daisy.org/activities/standards/daisy/daisy-3/z39-86-2005-r2012-specifications-for-the-digital-talking-book/>. (accessed: 21/11/2023).
- [10] UK Association of Accessible Standards. *Standards: HTML*. URL: <https://www.ukaaf.org/standards/#html>. (accessed: 26/11/2023).

- [11] Wix. *Web Accessibility - Make Your Wix Website Accessible*. URL: <https://www.wix.com/accessibility>. (accessed: 01.11.2023).
- [12] Squarespace. *Squarespace Accessibility & Compliance*. URL: <https://www.squarespace.com/accessibility>. (accessed: 01.11.2023).
- [13] Joel Olawanle. *How to Copy Text to the Clipboard with JavaScript*. URL: <https://www.freecodecamp.org/news/copy-text-to-clipboard-javascript/>.
- [14] W3Schools. *HTML `<input type=radio>`*. URL: https://www.w3schools.com/tags/att_input_type_radio.asp.
- [15] GeeksforGeeks. *HTML `<input type=radio>`*. URL: <https://www.geeksforgeeks.org/html-input-typeradio/>.
- [16] Mozilla. *`<input type=radio>`*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/radio>.