User1st `CMS Use Cases`

January 1st, 2023

Ben Rosanes, Ariel Orbach, Xiaoyu (Aki) Gao, Vasile Nastas

Abstract. Usability and User Experience are critical aspects of software development, with accessibility being a fundamental component. User1st's integrated `u1 Toolkit`, a special designed Software Development Kit (SDK) used to cover various accessibility issues and different complexity, within many web solutions. Among these web solutions, User1st encountered different tehcnology stacks or profiles. One of the most common met represent the Content Management System (CMS). This whitepaper will present the use cases that were studied by User1st and applied among several CMS.

1. Overview

1.1. Content Management System

A content management system (CMS) is a web application or software that manages digital content, allowing multiple contributors to create, edit and publish from a single dashboard. Content in a CMS is typically stored in a database and displayed in a presentation layer based on a set of templates like a website. A CMS will unify multiple workstreams so that users can effectively collaborate on all types of digital content creation from anywhere. Companies typically rely on a dynamic CMS in order to effectively scale content creation without the need for technical investment.

Going over the market share [1,2] can be clearly observed that there are a lot of CMS frameworks, among them the most popular are:

- WordPress [3];
- Shopify [4];
- Magento [5];
- Umbraco [6];
- Joomla [7];

- etc.

1.2. CMS Accessibility

Any CMS is considered as a standard website from the legal point of view. As result as any website that provides content to public, the website built under any CMS framework must be accessible complaint. Going over the requirements usually each govermental institution or private oragnisation tries to go over and respect the Web Content Accessibility Guidelines (WCAG).

1.2.1 WCAG Compliance

The Web Content Accessibility Guidelines (WCAG) define three levels of compliance: A, AA, and AAA. Each level encompasses guidelines that must be met to ensure a website is accessible to all users. These conformance levels provide developers with a structured approach to achieving minimal, acceptable, and optimal accessibility, offering flexibility for complex or large websites to maintain a minimum level of compliance.

- Level A: Represents minimal compliance, addressing the most basic web accessibility features;
- Level AA: Represents acceptable compliance, addressing the most common and impactful barriers for users with disabilities;
- Level AAA: Represents optimal compliance, addressing the highest and most comprehensive level of web accessibility [8].

2.2. User1st Solutions

The `u1 Toolkit` comprises two solutions.

2.2.1 u1 Toolbar

The `u1 Toolbar` is a versatile widget designed for seamless integration into any web application, offering robust accessibility scanning capabilities. Framework-agnostic, it performs static analysis on individual web pages to identify accessibility violations. Beyond detection, `u1 Toolbar` provides actionable guidance to help developers address and resolve these issues, ensuring a more inclusive and accessible user experience.

2.2.2 u1 A11y

u1 A11y is a framework-agnostic library designed to facilitate semi-automatic accessibility remediation in web applications. This tool enables developers to build accessible components without extensive knowledge of accessibility standards.

2.3. Benefits

2.3.1 Enhance User Experience

Applications built with the `u1 Toolkit` offer a more inclusive user experience. By addressing accessibility issues, the toolkit helps create web applications that are easier to navigate and use for all individuals, leading to higher user satisfaction and engagement.

2.3.2 Improve Accessibility Compliance

The `u1 Toolkit` ensures that web applications adhere to global accessibility standards, making them accessible to a wider audience, including individuals with disabilities. This compliance not only broadens your user base but also helps you meet legal and regulatory requirements.

2.3.3 Increase Developer Productivity

By automating significant portions of the accessibility remediation process, the `u1 Toolkit` allows developers to concentrate on other critical development tasks. This automation leads to more efficient workflows and quicker project completion times.

3. User1st CMS Use Cases

3.1 WordPress

User1st was implemented within several Wordpress projects. The main and only point of access for User1st solution is integrate within the index.html. This approach is valid for any other CMS project, for now other public, open and tested solutions are not well-known.

In order to in integrate within a Wordpress solution any developer or user has to do the following steps:

1. Log in within the Wordpress site management dashboard.
2. From the admin panel, navigate to *Appeareance* sidebar menu option, choose and select *Theme File Editor*. This can be observed in the following Figure 3.1.

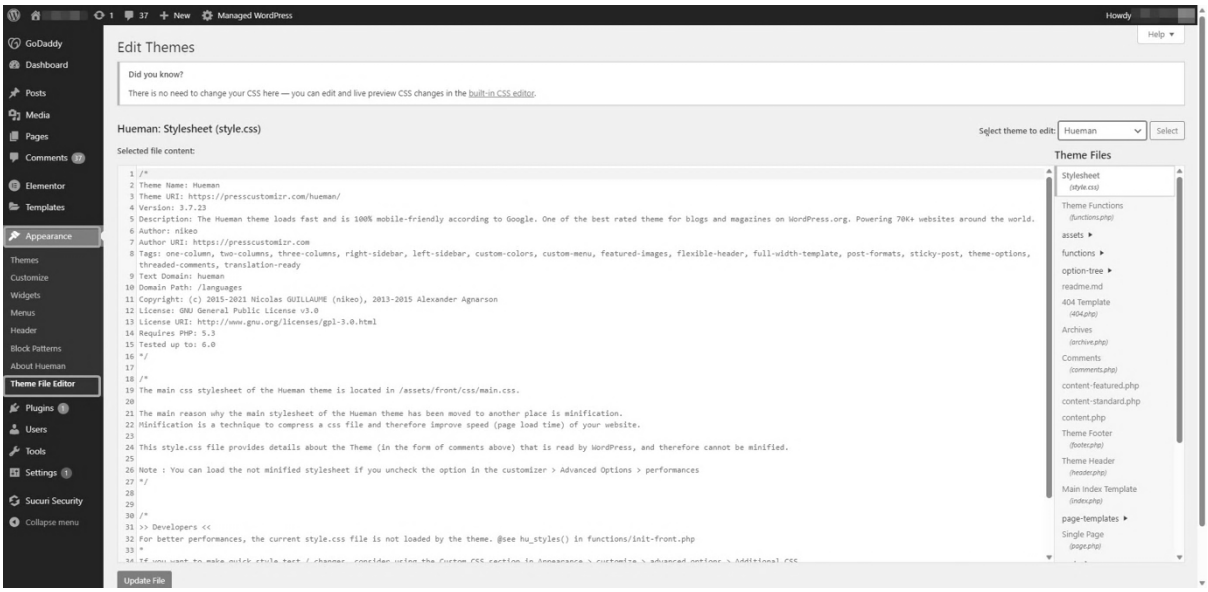

Figure 3.1. WordPress Theme File Editor

3. In the *Theme File Editor* section, need to locate the theme listbox and choose the theme you want to modify, afterwards click on the *Select* button. This is presented in the following Figure 3.2.

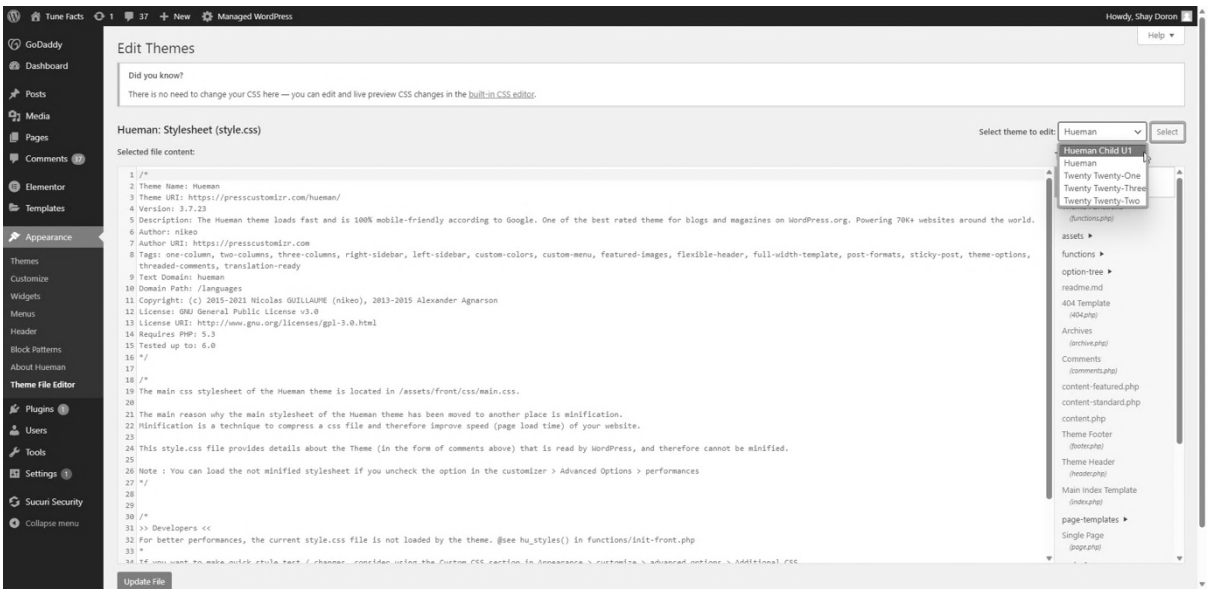

Figure 3.2. WordPress Theme File Editor Appearance

4. In the *Theme Files* sidebar, it is required to find and open the *Theme Header* **(header.php)** file.
5. Need to scroll down the code until the closing **<head>** tag.
6. Need to place the **u1Css <link>** just above the closing **<head>** tag.
7. Finally click the *Update File* to save changes, presented in the Figure 3.3.

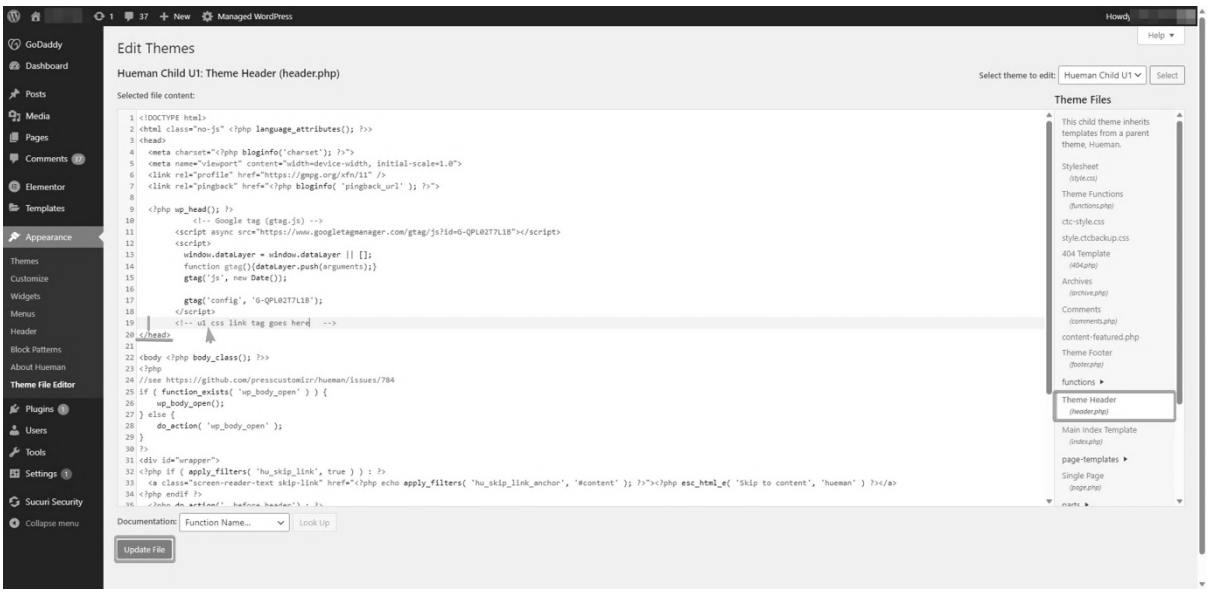

Figure 3.3. WordPress Theme File Editor Appearance

8. In the *Theme Files* sidebar, find and open the *Theme Footer* **(footer.php)** file.
9. Scroll down the code until you find the closing **<body>** tag (refer to the marker in the Figure 3.3).
10. Place the **u1Js <script>** just above the closing **<body>** tag.
11. Click *Update File* to save changes, presented in Figure 3.4.

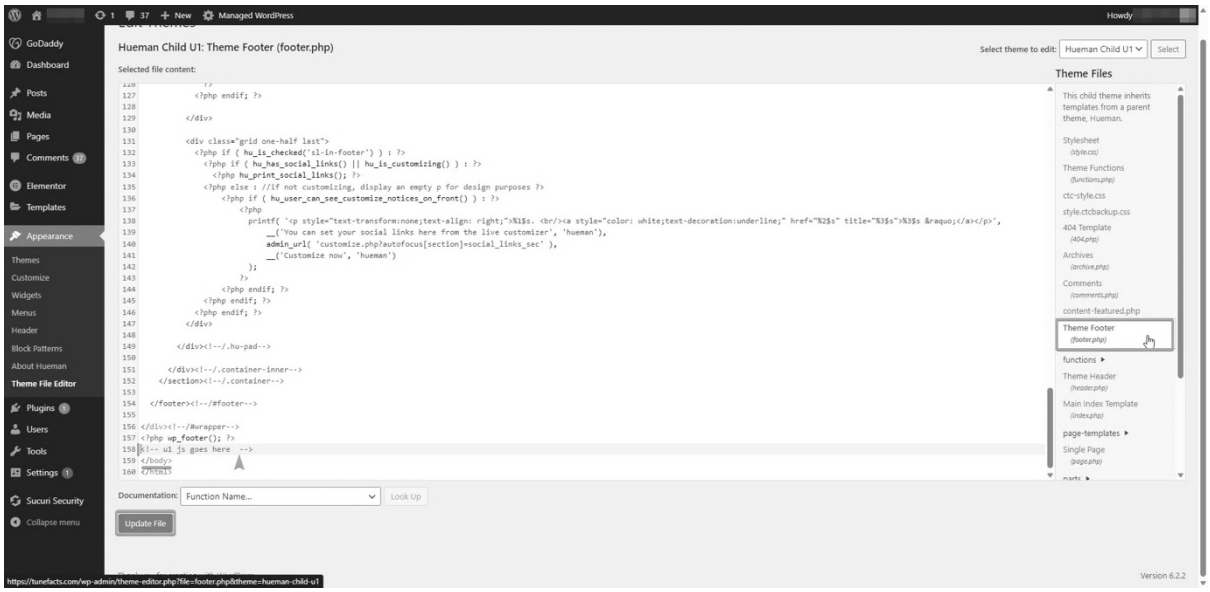

Figure 3.3. WordPress Theme File Editor Appearance

3.2 Shopify

Shopify represents another User1st successful use case. This CMS, as any other platform requires specific accessibility support in order to be comlpiant.

In order to in integrate within a Shopify solution any developer or user has to do the following steps:

1. Log in to your Shopify site management dashboard.
2. From the admin panel, navigate to the *Online Store* section in the bottom left menu and select *Themes* (Figure 3.4).

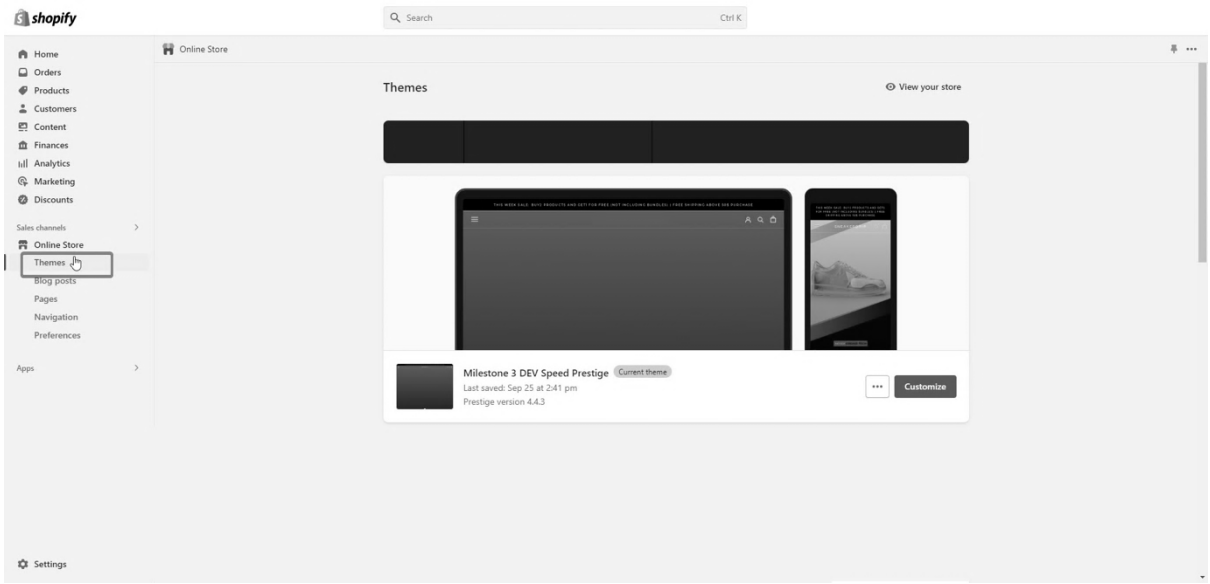

Figure 3.4. Shopify Themes

3. In the *Themes* section, locate the theme you want to modify and click on the more options button dropdown (...). Choose *Edit Code* from the options (Figure 3.5).

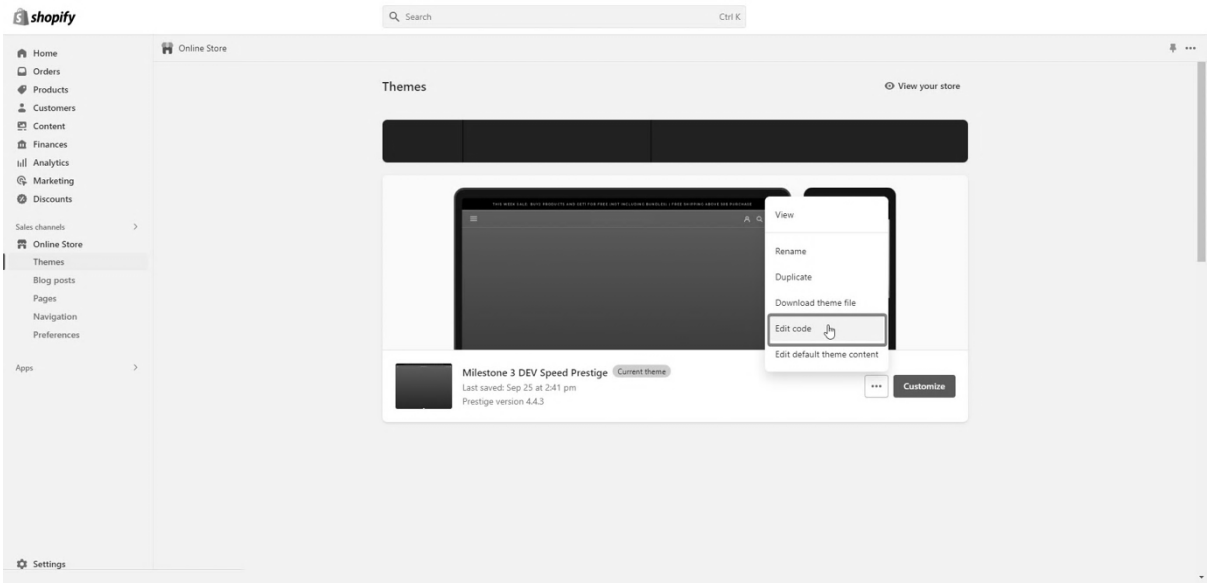

Figure 3.5. Shopify Themes

4. In the *Layout* section, find and open the **`theme.liquid`** file.
5. Scroll down the code until you find the opening **`<head>`** tag (refer to the first marker in the Figure 3.6).

6. Place the **`u1Css <link>`** just below the opening **`<head>`** tag.
7. Scroll further down the code until you find the closing **`</body>`** tag (refer to the second marker in Figure 3.6).
8. Place the JavaScript (js) **`<script>`** just before the closing **`</body>`** tag.

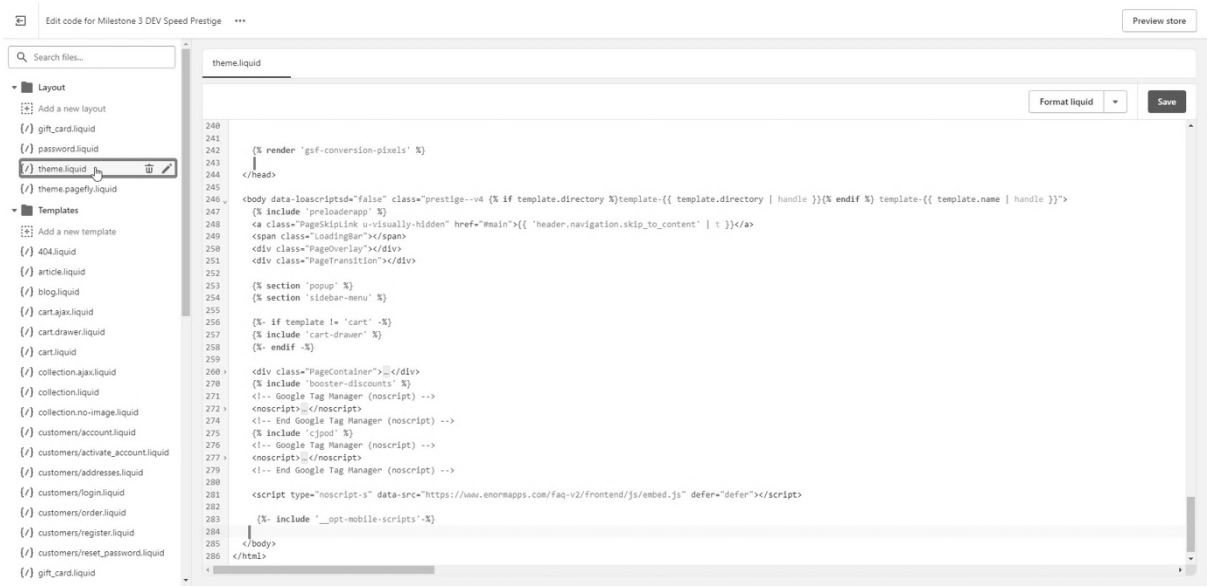

Figure 3.6. Shopify **`theme.liquid`** file

9. Save the changes by clicking the *Save* button located in the top right corner of the code editor, presented in the following Figure 3.7.

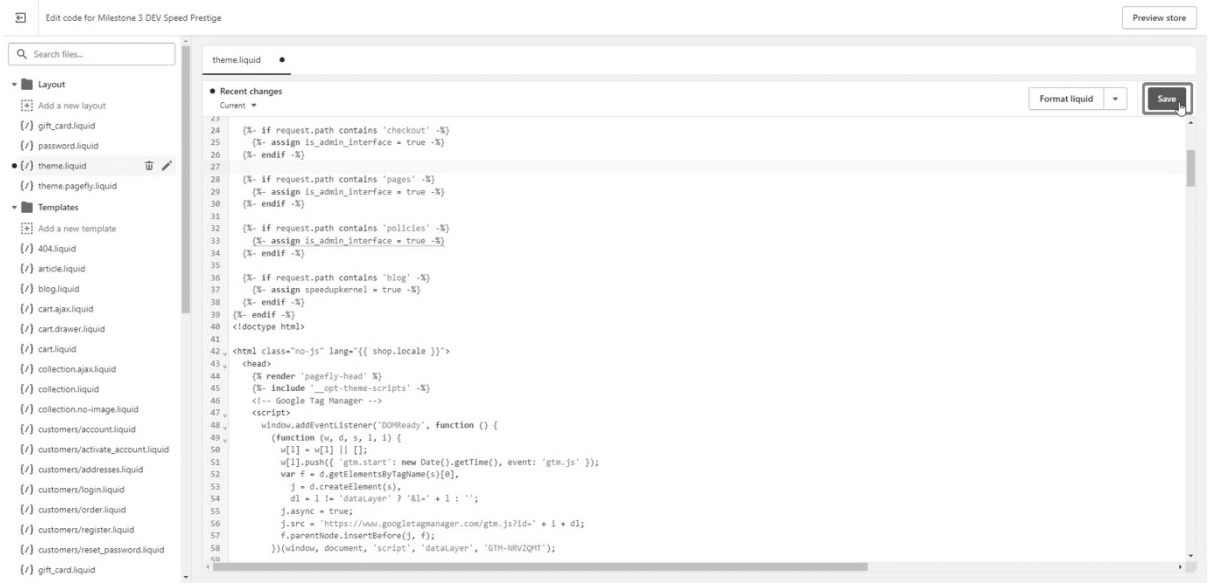

Figure 3.6. Shopify **`theme.liquid`** file

3.3 Magento

Magento represents an open source CMS, that is well known among other CMS like WordPress, Shopify etc. User1st integrated successful within this type of CMS solutions and completely supported accessibility needs.

In order to in integrate within a Shopify solution any developer or user has to do the following steps:

1. Define a Global Script in RequireJS. Use Magento's **`requirejs-config.js`** (or create one if not defined) in *app/design/frontend/<Vendor>/<Theme>/requirejs-config.js* to define a global script (Figure 3.7).

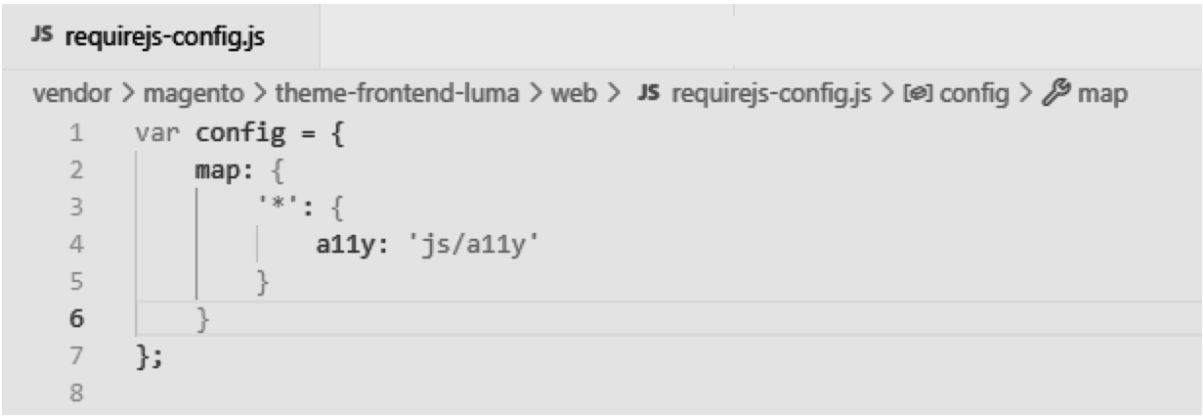

Figure 3.7. Magento **`requirejs-config.js`**

2. Create the Custom JavaScript File and place the custom JavaScript file in the *theme's web/js* directory *app/design/frontend/<Vendor>/<Theme>/web/js/a11y.js* (Figure 3.8).

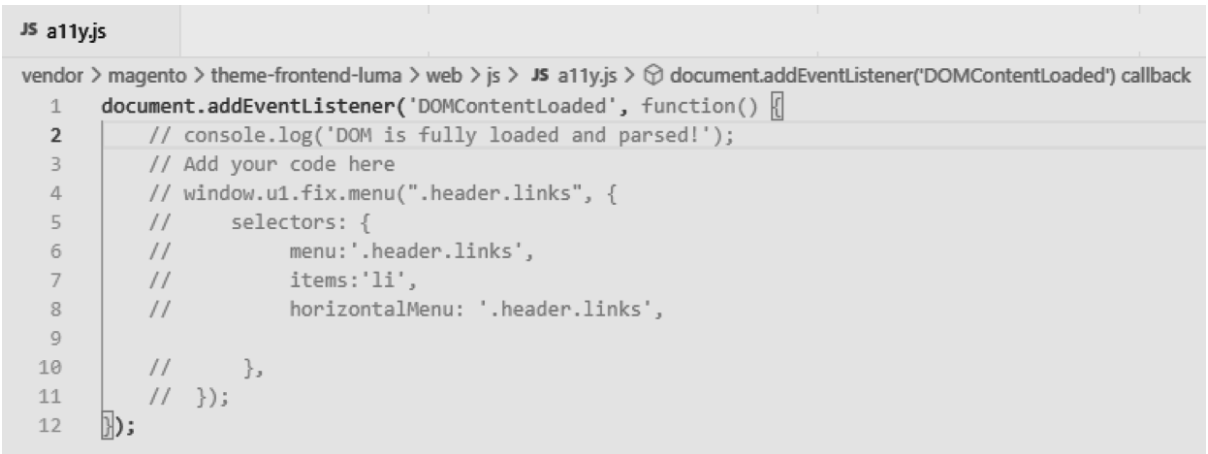

Figure 3.8. Magento **`a11y.js`**

3. Load the script globally by editing **`default.xml`** (Figure 3.9).

Figure 3.9 Magento **`default.xml`**

3.4 u1 Showcases

This section will provide a list of showcases that can be applied with the help of u1. u1 offers a11y fix via u1 attributes or manual selectors.

u1 attributes

u1 attributes represent the shortest and fastest way to fix a11y issues. In case there is a custom component with access to its `HTML`, it will be required simply to apply u1 attribute according to the official documentation, otherwise it will be required to apply manual selectors.

u1 Manual Selectors

Manual selectors provide a `CSS` selector as the first property to wait for an element in the `DOM`. The second property is an object containing further selectors or options. Unless specified, use relative selectors to the container element. This method is useful when you prefer `CSS` selectors or when working with UI libraries lacking `HTML` access. Adding classes and IDs can create more robust selectors.

3.4.1 Button Use Case

A button is an independent component which is responsible for performing a functionality on the same page the user is visiting. If the button is not a native button,

it is not accessible. To make a non-native button accessible, it has a proper description. This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.10 and 3.11.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Button Example</title>
</head>

<body>
    <div onclick="onClick()" class="test-btn" u1-button>
        <img src="path/to/image.png" />
    </div>

    <script>
        function onClick() {
            // Do something
        }
    </script>
</body>
</html>
```

Figure 3.10 Button fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Button Example</title>
</head>

<body>
    <div onclick="onClick()" class="test-btn">
        <img src="path/to/image.png" />
    </div>

    <script>
        function onClick() {
            // Do something
        }
        window.u1?.fix.button(".test-btn", {
```

```
            selectors: {
                element: ".test-btn",
            },
        });
    </script>
</body>
</html>
```

Figure 3.11 Button fixed using u1 manual selectors

3.4.2 Link Use Case

135 Accessible links redirect users to a new page or view, opening in the same or a new tab. Activated by the Enter or Spacebar key, they require clear, screen reader-detectable descriptions of the destination.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.12 and 3.13.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Button Example</title>
</head>

<body>

    <button onclick="navigate('/accessibility-
statement')"    class="test-link" u1-link>
        Accessibility Statement
    </button>

    <script>
        function navigate(url) {
            window.location.href = url;
        }
    </script>

</body>
</html>
```

Figure 3.12 Link fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Link Example</title>
</head>

<body>
    <button onclick="navigate('/accessibility-statement')"
class="test-link">
        Accessibility Statement
    </button>

    <script>
        function navigate(url) {
            window.location.href = url;
        }

        window.u1?.fix.link('.test-link', {
            selectors: {
                element: '.test-link',
            },
        });
    </script>
</body>
</html>
```

Figure 3.13 Link fixed using u1 manual selectors

3.4.3 Radio Use Case

Radio buttons come as a group. Only one radio button can be selected in a radio group. If the component is not a native input of radio type, it will most likely need our attention. Each radio which is visible to the user will need a proper description and keyboard navigation.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.14 and 3.15.

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <title>Radio Example</title>
</head>

<body>
    <div class="radio-group-container" u1-radio u1-group>
        <div class="radio" onclick="setSelected(0)" u1-
button>Option 1</div>
        <div class="radio" onclick="setSelected(1)" u1-
button>Option 2</div>
        <div class="radio" onclick="setSelected(2)" u1-
button>Option 3</div>
    </div>

    <script>
        function setSelected(index) {
            const radios = document.querySelectorAll('.radio');
            radios.forEach((radio, i) => {
                if (i === index) {
                    radio.classList.add('selected');
                    radio.setAttribute("u1-checked", "true");
                } else {
                    radio.classList.remove('selected');
                    radio.setAttribute("u1-checked", "false");
                }
            });
        }
    </script>
</body>
</html>
```

Figure 3.14 Radio fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Radio Example</title>
</head>

<body>
    <div class="radio-group-container">
        <div class="radio" onclick="setSelected(0)">Option 1</div>
        <div class="radio" onclick="setSelected(1)">Option 2</div>
        <div class="radio" onclick="setSelected(2)">Option 3</div>
```

```
    </div>
    <script>
        function setSelected(index) {
            const radios = document.querySelectorAll('.radio');
            radios.forEach((radio, i) => {
                if (i === index) {
                    radio.classList.add('selected');
                } else {
                    radio.classList.remove('selected');
                }
            });
        }

        window.u1?.fix.radio('.radio-group-container', {
            selectors: {
                radioGroup: '.radio-group-container',
                radioButton: '.radio',
                checkedState: '.radio.selected',
                uncheckedState: '.radio:not(.selected)',
            }
        });
    </script>
</body>
</html>
```

Figure 3.15 Link fixed using u1 manual selectors

3.4.4 Checkbox Use Case

Checkbox is an element a user can select and deselect. They often come in groups and the user can select one checkbox or more. Or they can come as singles. If your element is not a visible native input of type checkbox, it will probably need our attention. In order for a checkbox to be accessible, it will need a proper description and an option to check it using the keyboard.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.16 and 3.17.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Checkbox Example</title>
</head>
```

```
<body>
    <div class="checkbox-container">
        <div class="checkbox" onclick="toggleChecked()" u1-
checkbox>
            <input type="checkbox" id="checkboxInput"
onchange="updateCheckbox()"
            u1-exclude />
            <label for="checkboxInput">Agree To Terms</label>
        </div>
    </div>

    <script>
        let checked = false;

        function toggleChecked() {
            const checkboxInput =
document.getElementById('checkboxInput');
            checked = !checked;
            checkboxInput.checked = checked;
            updateCheckbox();
        }

        function updateCheckbox() {
            const checkboxContainer =
document.querySelector('.checkbox');
            if (checked) {
                checkboxContainer.classList.add('selected');
                checkboxContainer.setAttribute('u1-checked',
"true");
            } else {
                checkboxContainer.classList.remove('selected');
                checkboxContainer.setAttribute('u1-checked',
"false");
            }
        }
    </script>
</body>
</html>
```

Figure 3.16 Checkbox fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <title>Checkbox Example</title>
</head>

<body>
    <div class="checkbox-container">
        <div class="checkbox" onclick="toggleChecked()">
            <input type="checkbox" id="checkboxInput"
onchange="updateCheckbox()" />
            <label for="checkboxInput">Agree To Terms</label>
        </div>
    </div>

    <script>
        let checked = false;
        function toggleChecked() {
            const checkboxInput =
document.getElementById('checkboxInput');
            checked = !checked;
            checkboxInput.checked = checked;
            updateCheckbox();
        }
        function updateCheckbox() {
            const checkboxContainer =
document.querySelector('.checkbox');
            if (checked) {
                checkboxContainer.classList.add('selected');
            } else {
                checkboxContainer.classList.remove('selected');
            }
        }

        window.u1?.fix.checkbox('.checkbox-container', {
            selectors: {
                element: '.checkbox',
                checkedState: '.checkbox.checked',
                uncheckedState: '.checkbox.not:(.checked)',
                exclude: '.checkbox input',
            },
        });
    </script>
</body>
</html>
```

Figure 3.17 Checkbox fixed using u1 manual selectors

3.4.5 Accordion Use Case

Accordions are common design structures used to organize and hide content to not overwhelm the user. Clicking the expandible buttons will reveal a new section associated with it. Accordion buttons should have a description and a keyboard navigation option.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.18 and 3.19.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Accordion Example</title>
</head>

<body>
    <div class="accordion-container" u1-accordion>
        <div class="accordion-btn" onclick="toggleExpanded()" u1-
header>
            What is Accessibility?
        </div>
        <div class="accordion-content" id="accordionContent" u1-
content>
            The ability to access and benefit from some system or
entity.
        </div>
    </div>

    <script>
        let expanded = false;
        function toggleExpanded() {
            const accordionContent =
document.getElementById('accordionContent');

            if (expanded) {
                accordionContent.classList.remove('visible');
            } else {
                accordionContent.classList.add('visible');
            }

            expanded = !expanded;
        }
```

```
    </script>
</body>
</html>
```

Figure 3.18 Accordion fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Accordion Example</title>
</head>

<body>
    <div class="accordion-container">
        <div class="accordion-btn" onclick="toggleExpanded()">
            What is Accessibility?
        </div>
        <div class="accordion-content" id="accordionContent">
            The ability to access and benefit from some system or
entity.
        </div>
    </div>

    <script>
        let expanded = false;
        function toggleExpanded() {
            const accordionContent =
document.getElementById('accordionContent');

            if (expanded) {
                accordionContent.classList.remove('visible');
            } else {
                accordionContent.classList.add('visible');
            }

            expanded = !expanded;
        }

        window.u1?.fix.accordion('.accordion-container', {
            selectors: {
                headerSelector: '.accordion-btn',
                contentSelector: '.accordion-content',
```

```
            },
            headingLevel: 3,
        });
    </script>
</body>
</html>
```

Figure 3.19 Accordion fixed using u1 manual selectors

3.4.6 Form Use Case

Any non-native form will require our attention. Accessible forms must include required fields and specific errors for the inputs. If you have any non native components inside the form such as listbox, checkbox, etc. - please make sure they are being handled by U1 separately.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.20 and 3.21.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Form Example</title>
</head>

<body>
    <div class="form-container" u1-form>
        <h2 id="form-title" u1-label>Contact Us</h2>

        <label for="name-field">Name</label>
        <input placeholder="Name" id="name-field" u1-required />
        <span class="error" u1-error>Please name this
field</span>

        <label for="phone-field">Phone Number</label>
        <input placeholder="Phone number" id="phone-field" u1-
required />
        <span class="error" u1-error>Please enter phone
field</span>

        <button type="button" onclick="submitForm()" u1-
```

```
submit>Submit</button>
    </div>

    <script>
        function submitForm() {
            // form submission logic
        }
    </script>
</body>
</html>
```

Figure 3.20 Form fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Form Example</title>
</head>
<body>
    <div class="form-container">
        <h2 id="form-title">Contact Us</h2>

        <label for="name-field">Name</label>
        <input placeholder="Name" id="name-field" />
        <span class="error">Please name this field</span>

        <label for="phone-field">Phone Number</label>
        <input placeholder="Phone number" id="phone-field" />
        <span class="error">Please enter phone field</span>

        <button type="button" class="submit"
onclick="submitForm()">Submit</button>
    </div>
    <script>
        function submitForm() {
            // form submission logic
        }
        window.u1?.fix.form('.form-container', {
            selectors: {
                form: '.form-container',
                submitButton: '.submit',
                formLabelAbsolute: '#form-title',
```

```
                inputField: 'input',
                invalidField: 'input:has(+span.error)',
                requiredField: 'input',
                errorMsg: '.error',
            },
            focusOnInvalidField: true,
        });
    </script>
</body>
</html>
```

Figure 3.21 Form fixed using u1 manual selectors

3.4.7 Tabs Use Case

Tabs or tab-control is a component responsible for changing the view of a specific section when a tab is clicked. Unless your tab-control doesn't refresh the page, you should use the tabs fixer. If your tabs are refreshing the page you can use the link fixer for the tabs and make sure they have a proper description.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.22 and 3.23.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Tabs Example</title>
</head>

<body>
    <div class="tabs-container" u1-tabs>
        <div class="tabs-list" u1-list>
            <div class="tab" onclick="selectTab(0)" u1-tab>Screen
Reader</div>
            <div class="tab" onclick="selectTab(1)" u1-
tab>Keyboard Navigation</div>
            <div class="tab" onclick="selectTab(2)" u1-tab>Colors
Profile</div>
        </div>
        <div class="tabs-panel" id="tabsPanel" u1-panel>
            <span id="content"></span>
```

```
        </div>
    </div>

    <script>
        const tabsContent = [
            "Content for Screen Reader tab",
            "Content for Keyboard Navigation tab",
            "Content for Colors Profile tab"
        ];

        function selectTab(index) {
            const contentElement =
document.getElementById('content');
            contentElement.textContent = tabsContent[index];
        }
    </script>
</body>
</html>
```

Figure 3.22 Tabs fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Tabs Example</title>
</head>

<body>
    <div class="tabs-container">
        <div class="tabs-list">
            <div class="tab" onclick="selectTab(0)">Screen
Reader</div>
            <div class="tab" onclick="selectTab(1)">Keyboard
Navigation</div>
            <div class="tab" onclick="selectTab(2)">Colors
Profile</div>
        </div>
        <div class="tabs-panel" id="tabsPanel">
            <span id="content"></span>
        </div>
    </div>
```

```
    <script>
        const tabsContent = [
            "Content for Screen Reader tab",
            "Content for Keyboard Navigation tab",
            "Content for Colors Profile tab"
        ];

        function selectTab(index) {
            const contentElement =
document.getElementById('content');
            contentElement.textContent = tabsContent[index];
        }

        window.u1?.fix.tabs('.tabs-container', {
            selectors: {
                tab: '.tab',
                tabList: '.tabs-list',
                tabPanel: '.tabs-panel',
            },
        });
    </script>
</body>
</html>
```

Figure 3.23 Tabs fixed using u1 manual selectors

3.4.8 Menu Use Case

A menu is a widget that offers a list of choices to the user, such as a set of links or actions. There are two types of menus: Navigation menus and Application menus. Application menus include actions or functions inside an application and behave like native operating system menus, such as the menus that are commonly found at the top of many desktop applications, whereas Navigation menus are used to navigate a website and usually contain links.

IMPORTANT:

Need to take into consideration that if there are any links in the menu that are triggers of a submenu and can also navigate you to a different page, it is not accessible. Links of that kind will be overwritten or have duplicate focus which may confuse the user.

Need to make sure to add these links as part of the items in your submenu, otherwise the component is at risk of not being fully accessible.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.24 and 3.25.

```
<!-- plain text -->
<div id="menu">
    <a href="/about" class="menu-item">
        About
    </a>
    <a href="/careers" class="menu-item">
        Careers
    </a>
    <a href="/contact-us" class="menu-item">
        Contact Us
    </a>
</div>

<!-- using aria-label -->
<div id="menu">
    <a href="/about" class="menu-item" aria-label="about">
        <img src="aboutIcon" alt="" />
        About
    </a>
    <a href="/careers" class="menu-item" aria-label="careers">
        <img src="careerIcon" alt="" />
        Careers
    </a>
    <a href="/contact-us" class="menu-item" aria-label="contact
us">
        <img src="contactIcon" alt="" />
        Contact Us
    </a>
</div>
</html>
```

Figure 3.24 Menu fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <title>Menu Example</title>
</head>

<body>
    <ul id="menu">
        <li class="has-submenu" onmouseenter="setIsOpen(true)"
        onmouseleave="setIsOpen(false)">
            <a class="menu-item" href="/about">
                About
            </a>
            <ul class="submenu" style="display:none;">
                <li>
                    <a href="/our-team" class="menu-item">
                        Our Team
                    </a>
                </li>
                <li>
                    <a href="/our-product" class="menu-item">
                        Our Product
                    </a>
                </li>
            </ul>
        </li>
        <li>
            <a href="/careers" class="menu-item">
                Careers
            </a>
        </li>
        <li>
            <a href="/contact-us" class="menu-item">
                Contact Us
            </a>
        </li>
    </ul>

    <script>
        let isOpen = false;

        function setIsOpen(value) {
            isOpen = value;
            const submenu = document.getElementById('submenu');
            submenu.style.display = isOpen ? 'block' : 'none';
        }
```

```
        window.u1?.fix.menu('#menu', {
            selectors: {
                menu: '#menu',
                horizontalMenu: '#menu',
                submenus: '.submenu',
                items: 'a.menu-item:not(.has-submenu), li.has-
submenu',
                triggers: 'li.has-submenu',
                openByMouseover: 'li.has-submenu',
            },
            menuDescription: 'Site navigation menu',
        });
    </script>
</body>
</html>
```

Figure 3.25 Menu fixed using u1 manual selectors

3.4.9 Listbox Use Case

A listbox component contains a trigger element that opens up a set of options. The list is predefined and its amount of options is finite and permanent. Unless you use a native select element correctly, you will need a U1 intervention. You will also need to
195 make sure the listbox trigger and options contain proper descriptions.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.26 and 3.27.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Listbox Example</title>
</head>

<body>
    <div class="listbox-container" u1-listbox>
        <h2 u1-label>Amount of users</h2>
        <div class="listbox-trigger" onclick="setIsOpen(!isOpen)"
u1-trigger>
            Select Amount
        </div>
        <ul id="listbox-list" style="display: none;" u1-list>
```

```
            <li class="listbox-option" onclick="selectOption(0)"
u1-option>Option 1</li>
            <li class="listbox-option" onclick="selectOption(1)"
u1-option>Option 2</li>
            <li class="listbox-option" onclick="selectOption(2)"
u1-option>Option 3</li>
        </ul>
    </div>

    <script>
        var isOpen = false;

        function setIsOpen(value) {
            isOpen = value;
            var listboxList = document.getElementById('listbox-
list');
            listboxList.style.display = isOpen ? 'block' :
'none';
        }

        function selectOption(index) {
            console.log('Selected option:', index + 1);
        }
    </script>
</body>
</html>
```

Figure 3.26 Listbox fixed using u1 attribute

With manual selectors exist different options available for use. For this method it is required to provide CSS selectors in case this option is more convenient for current project or its using a UI library with no access to its HTML. There might be a requirement to add classes and id's to the components to create strong selectors for use. The first property is a selector of the element of the DOM, the second property is an object containing a selectors object and / or options as shown below. Please use relative selectors to the container element unless specified otherwise.

```
<!DOCTYPE html>
<html lang="en">

<head>
```

```
    <title>Listbox Example</title>
</head>

<body>
    <div class="listbox-container">
        <h2 id="listbox-label">Amount of users</h2>
        <div class="listbox-trigger"
onclick="setIsOpen(!isOpen)">
            Select Amount
        </div>
        <ul id="listbox-list" style="display: none;">
            <li class="listbox-option"
onclick="selectOption(0)">Option 1</li>
            <li class="listbox-option"
onclick="selectOption(1)">Option 2</li>
            <li class="listbox-option"
onclick="selectOption(2)">Option 3</li>
        </ul>
    </div>

    <script>
        var isOpen = false;

        function setIsOpen(value) {
            isOpen = value;
            var listboxList = document.getElementById('listbox-
list');
            listboxList.style.display = isOpen ? 'block' :
'none';
        }

        function selectOption(index) {
            console.log('Selected option:', index + 1);
        }

        window.u1?.fix.listbox('.listbox-trigger', {
            selectors: {
                listbox: '#listbox-list',
                trigger: '.listbox-trigger',
                options: '.listbox-option',
                label: '#listbox-label',
            },
        });
    </script>
```

```
</body>
</html>
```

Figure 3.27 Menu fixed using u1 manual selectors

3.4.10 Combobox Use Case

A combobox component contains an input trigger with which you can filter and search through a set of options. The list can either be predefined or not and its amount of options can vary, like an autocomplete component for example. You will need to make sure the combobox trigger and options contain proper descriptions.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.28 and 3.29.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Combobox Example</title>
</head>

<body>
    <div class="combobox-container" u1-combobox>
        <label for="combobox-trigger" id="combobox-label" u1-
label>
            Find products
        </label>
        <div u1-combobox-input>
            <input type="text" id="combobox-trigger"
placeholder="Type here..." u1-textbox />
        </div>
        <ul id="combobox-list" style="display: none;" u1-listbox>
            <li onclick="handleSuggestionClick('Suggestion 1')"
u1-option>Suggestion 1</li>
            <li onclick="handleSuggestionClick('Suggestion 2')"
u1-option>Suggestion 2</li>
            <li onclick="handleSuggestionClick('Suggestion 3')"
u1-option>Suggestion 3</li>
        </ul>
    </div>

    <script>
```

```
        var inputValue = '';
        var showSuggestions = false;
        var filteredSuggestions = ['Suggestion 1', 'Suggestion
2', 'Suggestion 3'];

        function handleInputChange(value) {
            inputValue = value;
            showSuggestions = true;
            renderSuggestions();
        }

        function handleSuggestionClick(suggestion) {
            showSuggestions = false;
            renderSuggestions();
        }

        function renderSuggestions() {
            var listboxList = document.getElementById('combobox-
list');
            listboxList.style.display = showSuggestions ? 'block'
: 'none';
        }
    </script>
</body>
</html>
```

Figure 3.28 Combobox fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Combobox Example</title>
</head>

<body>
    <div class="combobox-container">
        <label for="combobox-trigger" id="combobox-label">
            Find products
        </label>
        <div>
            <input type="text" id="combobox-trigger"
placeholder="Type here..." />
        </div>
        <ul id="combobox-list" style="display: none;">
```

```
            <li onclick="handleSuggestionClick('Suggestion
1')">Suggestion 1</li>
            <li onclick="handleSuggestionClick('Suggestion
2')">Suggestion 2</li>
            <li onclick="handleSuggestionClick('Suggestion
3')">Suggestion 3</li>
        </ul>
    </div>

    <script>
        var inputValue = '';
        var showSuggestions = false;
        var filteredSuggestions = ['Suggestion 1', 'Suggestion
2', 'Suggestion 3'];

        function handleInputChange(value) {
            inputValue = value;
            showSuggestions = true;
            renderSuggestions();
        }

        function handleSuggestionClick(suggestion) {
            showSuggestions = false;
            renderSuggestions();
        }

        function renderSuggestions() {
            var listboxList = document.getElementById('combobox-
list');
            listboxList.style.display = showSuggestions ? 'block'
: 'none';
        }

        window.u1?.fix.combobox('.combobox-container', {
            selectors: {
                combobox: '.combobox-container',
                textbox: '#combobox-trigger',
                listbox: '#combobox-list',
                options: 'li',
                label: '#combobox-label',
            },
        });
    </script>
</body>
</html>
```

Figure 3.29 Combobox fixed using u1 manual selectors

3.4.11 Carousel Use Case

A Carousel is a slider of images that may contain links or buttons. Carousels must have previous and next buttons and optionally picker buttons to select a specific slide. All elements in the slider such as images / clickable elements must have proper descriptions.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.30 and 3.31.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Carousel Example</title>
</head>

<body>
    <div class="carousel-container" u1-carousel>
        <div onclick="prevSlide()" class="prev-btn" u1-prev-
btn>Prev</div>
        <div class="slide" u1-slide>
            <a id="slide-link" target="_blank" rel="noopener
noreferrer">
                <img id="slide-image" />
            </a>
        </div>
        <div onclick="nextSlide()" class="next-btn" u1-next-
btn>Next</div>
        <div class="picker-buttons" id="picker-buttons">
            <img src="selectedPickerIcon.png" alt="Slide 1"
onclick="changeSlide(0)" u1-picker />
            <img src="unselectedPickerIcon.png" alt="Slide 2"
onclick="changeSlide(1)" u1-picker />
        </div>
    </div>
    <script>
        var slides = [
            { src: "slide1.jpg", alt: "Description for slide 1",
link: "https://slide1.com/" },
            { src: "slide2.jpg", alt: "Description for slide 2",
link: "https://slide2.com/" },
        ];
```

```
        var currentSlide = 0;

        function prevSlide() {
            currentSlide = (currentSlide - 1 + slides.length) %
slides.length;
            updateSlide();
        }

        function nextSlide() {
            currentSlide = (currentSlide + 1) % slides.length;
            updateSlide();
        }

        function updateSlide() {
            var slideImage = document.getElementById('slide-
image');
            var slideLink = document.getElementById('slide-
link');
            slideImage.src = slides[currentSlide].src;
            slideImage.alt = slides[currentSlide].alt;
            slideLink.href = slides[currentSlide].link;
        }

        function changeSlide(index) {
            currentSlide = index;
            updateSlide();
        }
    </script>
</body>
</html>
```

Figure 3.30 Carousel fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Carousel Example</title>
</head>

<body>
    <div class="carousel-container">
        <div onclick="prevSlide()" class="prev-btn">Prev</div>
        <div class="slide">
            <a id="slide-link" target="_blank" rel="noopener
```

```
noreferrer">
                <img id="slide-image" />
            </a>
        </div>
        <div onclick="nextSlide()" class="next-btn">Next</div>
        <div class="picker-buttons" id="picker-buttons">
            <img src="selectedPickerIcon.png" alt="Slide 1"
onclick="changeSlide(0)" />
            <img src="unselectedPickerIcon.png" alt="Slide 2"
onclick="changeSlide(1)" />
        </div>
    </div>

    <script>
        var slides = [
            { src: "slide1.jpg", alt: "Description for slide 1",
link: "https://slide1.com/" },
            { src: "slide2.jpg", alt: "Description for slide 2",
link: "https://slide2.com/" },
        ];
        var currentSlide = 0;

        function prevSlide() {
            currentSlide = (currentSlide - 1 + slides.length) %
slides.length;
            updateSlide();
        }

        function nextSlide() {
            currentSlide = (currentSlide + 1) % slides.length;
            updateSlide();
        }

        function updateSlide() {
            var slideImage = document.getElementById('slide-
image');
            var slideLink = document.getElementById('slide-
link');
            slideImage.src = slides[currentSlide].src;
            slideImage.alt = slides[currentSlide].alt;
            slideLink.href = slides[currentSlide].link;
        }

        function changeSlide(index) {
            currentSlide = index;
            updateSlide();
        }
```

```
        window.u1?.fix.carousel('.slide', {
            selectors: {
                carouselContainer: '.carousel-container',
                slide: '.slide',
                prevButton: '.prev-btn',
                nextButton: '.next-btn',
                slidePickerButtons: '.picker-buttons>img',
            },
        });
    </script>
</body>
</html>
```

Figure 3.31 Carousel fixed using u1 manual selectors

3.4.12 Dialog Use Case

A dialog component is a modal that visually appears at the front of the page content and usually provides important information or confirms a user action on the page. A dialog will prevent the content beneath it from being clickable until it is closed.

IMPORTANT!

In order for a dialog to be fully accessible:

1. It must have a close button.

2. The close button must be the first clickable element in the dialog, visually and DOM wise.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.32 and 3.33.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Dialog Example</title>
</head>

<body>
    <div class="dialog-container">
```

```
        <h1>Dialog Example</h1>
        <button onclick="openDialog()" class="dialog-trigger" u1-
dialog-trigger
        u1-a11y-ref="dialog-example">Open Dialog</button>
        <div id="dialog" class="dialog" u1-dialog u1-a11y-
ref="dialog-example">
            <button onclick="closeDialog()" class="close-btn" u1-
close-btn>X</button>
            <div class="dialog-header">
                <h2 u1-heading>Dialog Title</h2>
            </div>
            <div class="dialog-content" u1-content>
                <p>This is the content of the dialog.</p>
            </div>
        </div>
    </div>

    <script>
        let isOpen = false;

        function openDialog() {
            document.getElementById('dialog').style.display =
'block';
            isOpen = true;
        }

        function closeDialog() {
            document.getElementById('dialog').style.display =
'none';
            isOpen = false;
        }
    </script>

</body>
</html>
```

Figure 3.32 Dialog fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Dialog Example</title>
</head>
```

```
<body>
    <div class="dialog-container">
        <h1>Dialog Example</h1>
        <button onclick="openDialog()" class="dialog-
trigger">Open Dialog</button>
        <div id="dialog" class="dialog">
            <button onclick="closeDialog()" class="close-
btn">X</button>
            <div class="dialog-header">
                <h2>Dialog Title</h2>
            </div>
            <div class="dialog-content">
                <p>This is the content of the dialog.</p>
            </div>
        </div>
    </div>

    <script>
        let isOpen = false;

        function openDialog() {
            document.getElementById('dialog').style.display =
'block';
            isOpen = true;
        }

        function closeDialog() {
            document.getElementById('dialog').style.display =
'none';
            isOpen = false;
        }
        window.u1?.fix.dialog('.dialog-trigger', {
            selectors: {
                dialog: '.dialog',
                trigger: '.dialog-trigger',
                closeBtn: '.close-btn',
                heading: '.dialog-header',
                textContent: '.dialog-content',
            },
        });
    </script>
</body>
</html>
```

Figure 3.33 Dialog fixed using u1 manual selectors

3.4.13 Datepicker Use Case

A datepicker component allows a user to select a date using a calendar. Usually, the calendar will contain a grid of days of the current month and the user will be able to switch the months and years to find the desired day. Any datepicker that is not a native input of type datepicker, will require our attention.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.34 and 3.35.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Datepicker Example</title>
</head>

<body>

    <div u1-datepicker>
        <h2>Select a Date:</h2>
        <div class="datepicker-trigger" onclick="toggleOpen()"
aria-label="select date"
        u1-trigger></div>
        <div id="datepickerContainer" style="display: none;"
class="datepicker-container"
        u1-modal>
            <div id="calendarWrapper">
            </div>
        </div>
        <span id="selectedDateSpan" style="display:
none;">Selected date:
            <span id="selectedDate">
            </span>
        </span>
    </div>

    <script>
        let open = false;
        let selectedDate = null;

        function toggleOpen() {
            open = !open;
            const datepickerContainer =
```

```
document.getElementById("datepickerContainer");
            const selectedDateSpan =
document.getElementById("selectedDateSpan");

            if (open) {
                datepickerContainer.style.display = "block";
                renderCalendar();
            } else {
                datepickerContainer.style.display = "none";
            }

            if (selectedDateSpan) {
                selectedDateSpan.style.display = selectedDate ?
"inline" : "none";
                document.getElementById("selectedDate").innerText
=
                    selectedDate ?
selectedDate.toLocaleDateString() : "";
            }
        }

        function renderCalendar() {
            const today = new Date();
            const currentYear = today.getFullYear();
            const currentMonth = today.getMonth() + 1;
            const firstDayOfMonth = new Date(currentYear,
currentMonth - 1, 1).getDay();
            const daysInMonth = new Date(currentYear,
currentMonth, 0).getDate();
            const days = Array.from({ length: daysInMonth }, (_,
i) => i + 1);

            const calendarWrapper =
document.getElementById("calendarWrapper");
            if (calendarWrapper) {
                calendarWrapper.innerHTML = `
                <h2>
                    <span class="month" u1-month-label>
                        ${today.toLocaleDateString('en-US', {
month: 'long' })}
                    </span>
                    <span class="year" u1-year-
label>${currentYear}</span>
                </h2>
                <table>
```

```
                    <thead>
                        <tr>
                            <th>Sun</th>
                            <th>Mon</th>
                            <th>Tue</th>
                            <th>Wed</th>
                            <th>Thu</th>
                            <th>Fri</th>
                            <th>Sat</th>
                        </tr>
                    </thead>
                    <tbody class="datepicker-grid" u1-days-table>
                        <tr>
                            ${Array(firstDayOfMonth).fill('<td
class="disabled" u1-disabled-day></td>')
                        .join('')}
                            ${days.map((day, index) => `
                                <td u1-day class="day"
onclick="handleDateChange(${currentYear},
                                 ${currentMonth - 1}, ${day},
${index}, this)">${day}</td>`).join('')}
                        </tr>
                    </tbody>
                </table>
            `;
            }
        }

        function handleDateChange(year, month, day, index,
element) {
            const selectedDay = new Date(year, month, day);
            selectedDate = selectedDay;
            const selectedDateSpan =
document.getElementById("selectedDateSpan");
            const selectedDateElement =
document.getElementById("selectedDate");

            const allDayCells =
document.querySelectorAll('.day');
            allDayCells.forEach(cell =>
cell.classList.remove('selected'));

            element.classList.add('selected');
            element.classList.setAttribute('u1-selected-day',
'true');
```

```
            if (selectedDateSpan) {
                selectedDateSpan.style.display = "inline";
                selectedDateElement.innerText =
selectedDate.toLocaleDateString();
            }
        }
    </script>

</body>

</html>
```

Figure 3.34 Datepicker fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Datepicker Example</title>
</head>

<body>

    <div>
        <h2>Select a Date:</h2>
        <div class="datepicker-trigger" onclick="toggleOpen()"
aria-label="select date">
            
        </div>
        <div id="datepickerContainer" style="display: none;"
class="datepicker-container">
            <div id="calendarWrapper">
            </div>
        </div>
        <span id="selectedDateSpan" style="display:
none;">Selected date:
        <span id="selectedDate"></span></span>
    </div>

    <script>
        let open = false;
        let selectedDate = null;
```

```
        function toggleOpen() {
            open = !open;
            const datepickerContainer =
document.getElementById("datepickerContainer");
            const selectedDateSpan =
document.getElementById("selectedDateSpan");

            if (open) {
                datepickerContainer.style.display = "block";
                renderCalendar();
            } else {
                datepickerContainer.style.display = "none";
            }

            if (selectedDateSpan) {
                selectedDateSpan.style.display = selectedDate ?
"inline" : "none";
                document.getElementById("selectedDate").innerText
= selectedDate ?
                selectedDate.toLocaleDateString() : "";
            }
        }

        function renderCalendar() {
            const today = new Date();
            const currentYear = today.getFullYear();
            const currentMonth = today.getMonth() + 1;
            const firstDayOfMonth = new Date(currentYear,
currentMonth - 1, 1).getDay();
            const daysInMonth = new Date(currentYear,
currentMonth, 0).getDate();
            const days = Array.from({ length: daysInMonth }, (_,
i) => i + 1);

            const calendarWrapper =
document.getElementById("calendarWrapper");
            if (calendarWrapper) {
                calendarWrapper.innerHTML = `
                <h2>
                    <span
class="month">${today.toLocaleDateString('en-US', { month: 'long'
})}
                    </span>
                    <span class="year">${currentYear}</span>
                </h2>
                <table>
                    <thead>
```

```
                        <tr>
                            <th>Sun</th>
                            <th>Mon</th>
                            <th>Tue</th>
                            <th>Wed</th>
                            <th>Thu</th>
                            <th>Fri</th>
                            <th>Sat</th>
                        </tr>
                    </thead>
                    <tbody class="datepicker-grid">
                        <tr>
                            ${Array(firstDayOfMonth).fill('<td
class="disabled"></td>').join('')}
                            ${days.map((day, index) => `
                                <td class="day"
onclick="handleDateChange(${currentYear},
                                ${currentMonth - 1}, ${day},
${index}, this)">${day}</td>
                            `).join('')}
                        </tr>
                    </tbody>
                </table>
            `;
            }
        }

        function handleDateChange(year, month, day, index,
element) {
            const selectedDay = new Date(year, month, day);
            selectedDate = selectedDay;
            const selectedDateSpan =
document.getElementById("selectedDateSpan");
            const selectedDateElement =
document.getElementById("selectedDate");

            const allDayCells =
document.querySelectorAll('.day');
            allDayCells.forEach(cell =>
cell.classList.remove('selected'));

            element.classList.add('selected');

            if (selectedDateSpan) {
                selectedDateSpan.style.display = "inline";
                selectedDateElement.innerText =
```

```
selectedDate.toLocaleDateString();
            }
        }

        window.u1?.fix.datepicker('.datepicker-trigger', {
            selectors: {
                container: '.datepicker-container',
                trigger: '.datepicker-trigger',
                year: {
                    label: '.year',
                },
                month: {
                    label: '.month',
                },
                days: {
                    table: 'tbody.datepicker-grid',
                    day: 'td.day',
                    selected: '.selected',
                    disabled: '.disabled',
                },
            },
        });
    </script>
</body>
</html>
```

Figure 3.35 Datepicker fixed using u1 manual selectors

3.4.14 Table Use Case

A table component is a static component. It's not interactive and doesn't contain any events, it is used strictly for reading purposes. If you're not using native table elements, or they are not implemented correctly and you're reluctant to make any changes in your table - you will need the U1 intervention. The table must consist of rows, so if you're not using native table elements, please make sure your table cells are divided into actual row containers and not just appear as rows by using CSS.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.36 and 3.37.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <title>Table Example</title>
</head>

<body>
    <div class="table" id="tableContainer" u1-table>
        <div class="header row" u1-row>
            <div class="cell header-cell" u1-col-
header>Name</div>
            <div class="cell header-cell" u1-col-header>Document
Type</div>
            <div class="cell header-cell" u1-col-
header>Description</div>
        </div>
    </div>

    <script>
        const docs = [
            { name: "Document Name 1", type: "Contract",
            desc: "This is the description for document 1." },
            { name: "Document Name 2", type: "",
            desc: "This is the description for document 2." }
        ];

        function populateTable() {
            const tableContainer =
document.getElementById("tableContainer");

            if (tableContainer) {
                docs.forEach((doc, index) => {
                    const row = document.createElement("div");
                    row.className = "row";
                    row.setAttribute('u1-row', '');

                    const nameCell =
document.createElement("div");
                    nameCell.className = "cell";
                    nameCell.setAttribute('u1-cell', '');
                    nameCell.innerText = doc.name;
                    row.appendChild(nameCell);

                    const typeCell =
document.createElement("div");
                    typeCell.className = "cell";
```

```
                    typeCell.setAttribute('u1-cell', '');
                    typeCell.innerText = doc.type;
                    row.appendChild(typeCell);

                    const descCell =
document.createElement("div");
                    descCell.className = "cell";
                    descCell.setAttribute('u1-cell', '');
                    descCell.innerText = doc.desc;
                    row.appendChild(descCell);

                    tableContainer.appendChild(row);
                });
            }
        }

        window.onload = function () {
            populateTable();
        };
    </script>
</body>
</html>
```

Figure 3.36 Table fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Table Example</title>
</head>

<body>
    <div class="table" id="tableContainer">
        <div class="header row">
            <div class="cell header-cell">Name</div>
            <div class="cell header-cell">Document Type</div>
            <div class="cell header-cell">Description</div>
        </div>
    </div>

    <script>
        const docs = [
            { name: "Document Name 1", type: "Contract",
```

```
            desc: "This is the description for document 1." },
            { name: "Document Name 2", type: "",
            desc: "This is the description for document 2." }
        ];

        function populateTable() {
            const tableContainer =
document.getElementById("tableContainer");

            if (tableContainer) {
                docs.forEach((doc, index) => {
                    const row = document.createElement("div");
                    row.className = "row";

                    const nameCell =
document.createElement("div");
                    nameCell.className = "cell";
                    nameCell.innerText = doc.name;
                    row.appendChild(nameCell);

                    const typeCell =
document.createElement("div");
                    typeCell.className = "cell";
                    typeCell.innerText = doc.type;
                    row.appendChild(typeCell);

                    const descCell =
document.createElement("div");
                    descCell.className = "cell";
                    descCell.innerText = doc.desc;
                    row.appendChild(descCell);

                    tableContainer.appendChild(row);
                });
            }
        }

        window.onload = function () {
            populateTable();
        };

        window.u1?.fix.table('.table', {
            selectors: {
                grid: '.table',
                row: '.row',
```

```
                cell: '.cell:not(.header-cell)',
                columnheader: '.header-cell',
            },
        });
    </script>
</body>
</html>
```

Figure 3.37 Table fixed using u1 manual selectors

3.4.15 Grid Use Case

A grid component is an interactive table as opposed to a regular table. If it has any clickable elements such as links, buttons, inputs or a filtering mechanism, it will be a grid table. The grid table must consist of rows, so if you're not using native table elements, please make sure your table cells are divided into actual row containers and not just appear as rows by using CSS.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.38 and 3.39.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Grid Table Example</title>
</head>

<body>
    <div class="grid-table" id="gridTable" u1-grid>
        <div class="grid-header grid-row" u1-row>
            <div class="grid-cell header-cell" u1-col-
header>Name</div>
            <div class="grid-cell header-cell" u1-col-
header>Document Type</div>
            <div class="grid-cell header-cell" u1-col-
header>Print</div>
        </div>
    </div>

    <script>
```

```
        const docs = [
            { id: 1, name: "Document 1", type: "Contract" },
            { id: 2, name: "Document 2", type: "Invoice" },
            { id: 3, name: "Document 3", type: "Receipt" }
        ];

        function populateGridTable() {
            const gridTable =
document.getElementById("gridTable");

            if (gridTable) {
                docs.forEach((doc, index) => {
                    const row = document.createElement("div");
                    row.className = "grid-row";

                    const nameCell =
document.createElement("div");
                    nameCell.className = "grid-cell";
                    nameCell.setAttribute('u1-cell', '');
                    nameCell.innerText = doc.name;
                    row.appendChild(nameCell);

                    const typeCell =
document.createElement("div");
                    typeCell.className = "grid-cell";
                    typeCell.setAttribute('u1-cell', '');
                    typeCell.innerText = doc.type;
                    row.appendChild(typeCell);

                    const printCell =
document.createElement("div");
                    printCell.className = "grid-cell";
                    printCell.setAttribute('u1-cell', '');
                    printCell.innerHTML = `<div
onclick="printDocument(${doc.id})">Print</div>`;
                    row.appendChild(printCell);

                    gridTable.appendChild(row);
                });
            }
        }

        function printDocument(documentId) {
            // print logic
        }
```

```
        window.onload = function () {
            populateGridTable();
        };
    </script>
</body>
</html>
```

Figure 3.38 Grid fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Grid Table Example</title>
</head>

<body>
    <div class="grid-table" id="gridTable">
        <div class="grid-header grid-row">
            <div class="grid-cell header-cell">Name</div>
            <div class="grid-cell header-cell">Document
Type</div>
            <div class="grid-cell header-cell">Print</div>
        </div>
    </div>

    <script>
        const docs = [
            { id: 1, name: "Document 1", type: "Contract" },
            { id: 2, name: "Document 2", type: "Invoice" },
            { id: 3, name: "Document 3", type: "Receipt" }
        ];

        function populateGridTable() {
            const gridTable =
document.getElementById("gridTable");

            if (gridTable) {
                docs.forEach((doc, index) => {
                    const row = document.createElement("div");
                    row.className = "grid-row";
```

```
                    const nameCell =
document.createElement("div");
                    nameCell.className = "grid-cell";
                    nameCell.innerText = doc.name;
                    row.appendChild(nameCell);

                    const typeCell =
document.createElement("div");
                    typeCell.className = "grid-cell";
                    typeCell.innerText = doc.type;
                    row.appendChild(typeCell);

                    const printCell =
document.createElement("div");
                    printCell.className = "grid-cell";
                    printCell.innerHTML = `<div
onclick="printDocument(${doc.id})">Print</div>`;
                    row.appendChild(printCell);

                    gridTable.appendChild(row);
                });
            }
        }
        function printDocument(documentId) {
            // print logic
        }
        window.onload = function () {
            populateGridTable();
        };
        window.u1?.fix.grid('.grid-table', {
            selectors: {
                grid: '.grid-table',
                row: '.grid-row',
                cell: '.grid-cell:not(.header-cell)',
                columnheader: '.header-cell',
            },
        });
    </script>
</body>
</html>
```

Figure 3.39 Grid fixed using u1 manual selectors

3.4.16 Pagination Use Case

A pagination component is used to divide content into pages usually when there's too much content for one page. It consists of a set of numbered buttons and often previous and next buttons which provide different results when clicked.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.40 and 3.41.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Pagination Example</title>
</head>

<body>

    <div class="pagination-container" u1-pagination>
        <div id="results">
            <div class="result" u1-result>
                <h2>Website 1</h2>
                <a href="http://example.com">Website 1</a>
                <p>Description of Website 1.</p>
            </div>
            <div class="result" u1-result>
                <h2>Website 2</h2>
                <a href="http://example.com">Website 2</a>
                <p>Description of Website 2.</p>
            </div>
        </div>

        <div id="pagination">
            <div class="prev" id="prevButton" u1-prev>
                Prev
            </div>
            <div class="pagination-button" id="page1Button"
u1-page>1</div>
            <div class="pagination-button" id="page2Button"
u1-page>2</div>
            <div class="next" id="nextButton" u1-next>
                Next
            </div>
        </div>
```

```
    </div>

    <script>
        const results = [
            { name: "Website 1", href:
"http://example.com",
            description: "Description of Website 1." },
            { name: "Website 2", href:
"http://example.com",
            description: "Description of Website 2." }
        ];

        const pages = [
            { number: 1 },
            { number: 2 }
        ];

        function changePage(pageNumber) {
            console.log("Changing to page:", pageNumber);
        }

        document.getElementById("prevButton").addEventListe
ner("click", function () {
            changePage(-1);
        });
        document.getElementById("nextButton").addEventListe
ner("click", function () {
            changePage(1);
        });
        document.getElementById("page1Button").addEventList
ener("click", function () {
            changePage(1);
        });
        document.getElementById("page2Button").addEventList
ener("click", function () {
            changePage(2);
        });
    </script>
</body>
</html>
```

Figure 3.40 Pagination fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Pagination Example</title>
</head>

<body>

    <div class="pagination-container">
        <div id="results">
            <div class="result">
                <h2>Website 1</h2>
                <a href="http://example.com">Website 1</a>
                <p>Description of Website 1.</p>
            </div>
            <div class="result">
                <h2>Website 2</h2>
                <a href="http://example.com">Website 2</a>
                <p>Description of Website 2.</p>
            </div>
        </div>

        <div id="pagination">
            <div class="prev" id="prevButton">
                Prev
            </div>
            <div class="pagination-button" id="page1Button">1</div>
            <div class="pagination-button" id="page2Button">2</div>
            <div class="next" id="nextButton">
                Next
            </div>
        </div>
    </div>

    <script>
        const results = [
            { name: "Website 1", href: "http://example.com",
            description: "Description of Website 1." },
            { name: "Website 2", href: "http://example.com",
            description: "Description of Website 2." }
        ];

        const pages = [
            { number: 1 },
            { number: 2 }
        ];
```

```
        function changePage(pageNumber) {
            console.log("Changing to page:", pageNumber);
        }

        document.getElementById("prevButton").addEventListener("click
", function () {
            changePage(-1);
        });
        document.getElementById("nextButton").addEventListener("click
", function () {
            changePage(1);
        });
        document.getElementById("page1Button").addEventListener("clic
k", function () {
            changePage(1);
        });
        document.getElementById("page2Button").addEventListener("clic
k", function () {
            changePage(2);
        });

        window.u1?.fix.pagination('.pagination-button', {
            selectors: {
                container: '.pagination-container',
                pageButtons: '.pagination-button',
                prevBtn: '.prev',
                nextBtn: '.next',
                results: '.result',
            },
        });
    </script>
</body>
</html>
```

Figure 3.41 Pagination fixed using u1 manual selectors

3.4.17 Loading Use Case

A loading component is used to indicate the loading state of a page or the current section the user is at. Displaying just a UI component is not enough for screen reader users, it should also announce a loading state. For this purpose you will need to use
265 the U1 loading fixer.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.42 and 3.43.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Loading Example</title>
</head>

<body>
    <div id="searchResults">
        <h1>Search results</h1>
        <div id="loadingMessage" u1-loading>Loading...</div>
        <div id="loadedContent"></div>
    </div>

    <script>
        const loading = true;
        const results = "Search results will be displayed here.";

        function displayContent() {
            const loadingMessage =
document.getElementById("loadingMessage");
            const loadedContent =
document.getElementById("loadedContent");

            if (loading) {
                loadingMessage.style.display = "block";
                loadedContent.style.display = "none";
            } else {
                loadingMessage.style.display = "none";
                loadedContent.style.display = "block";
                loadedContent.textContent = results;
            }
        }

        window.onload = function () {
            displayContent();
        };
    </script>
</body>
</html>
```

Figure 3.42 Loading fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Loading Example</title>
</head>

<body>

    <div id="searchResults">
        <h1>Search results</h1>
        <div id="loadingMessage" class="loading">Loading...</div>
        <div id="loadedContent"></div>
    </div>

    <script>
        const loading = true;
        const results = "Search results will be displayed here.";

        function displayContent() {
            const loadingMessage =
document.getElementById("loadingMessage");
            const loadedContent =
document.getElementById("loadedContent");

            if (loading) {
                loadingMessage.style.display = "block";
                loadedContent.style.display = "none";
            } else {
                loadingMessage.style.display = "none";
                loadedContent.style.display = "block";
                loadedContent.textContent = results;
            }
        }

        window.onload = function () {
            displayContent();
        };

        window.u1?.fix.loading('.loading', {
            selectors: {
                loadingBar: '.loading',
            },
        });
    </script>
```

```
</body>
</html>
```

Figure 3.43 Loading fixed using u1 manual selectors

3.4.18 Tooltip Use Case

A tooltip component is primarily used to provide further information about a topic or a section on the page. Usually, it's triggered by hovering over an info button but could also be triggered by a click or focus event.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.44 and 3.45.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Tooltip Example</title>
</head>

<body>

    <div class="tooltip-container" id="tooltipContainer">
        <button class="tooltip-trigger" id="tooltipTrigger" u1-
trigger u1-a11y-ref="tooltip1">
          More Info
        </button>
        <div class="tooltip" id="tooltipContent" u1-tooltip u1-
a11y-ref="tooltip1"></div>
    </div>

    <script>
        function handleMouseEnter() {
            document.getElementById("tooltipContent").style.display
= "block";
        }

        function handleMouseLeave() {
            document.getElementById("tooltipContent").style.display
= "none";
        }
```

```
        function updateTooltip(showTooltip, text) {
            const tooltipContent =
document.getElementById("tooltipContent");
            if (showTooltip) {
                tooltipContent.textContent = text;
                tooltipContent.style.display = "block";
            } else {
                tooltipContent.textContent = "";
                tooltipContent.style.display = "none";
            }
        }

        document.getElementById("tooltipTrigger")
         .addEventListener("mouseenter", handleMouseEnter);
        document.getElementById("tooltipTrigger")
         .addEventListener("mouseleave", handleMouseLeave);

        const showTooltip = true;
        const text = "Tooltip content goes here.";

        updateTooltip(showTooltip, text);
    </script>

</body>

</html>
```

Figure 3.44 Tooltip fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Tooltip Example</title>
</head>

<body>

    <div class="tooltip-container" id="tooltipContainer">
        <button class="tooltip-trigger" id="tooltipTrigger">More
Info</button>
        <div class="tooltip" id="tooltipContent"></div>
    </div>
```

```
    <script>
        function handleMouseEnter() {
            document.getElementById("tooltipContent").style.display
= "block";
        }

        function handleMouseLeave() {
            document.getElementById("tooltipContent").style.display
= "none";
        }

        function updateTooltip(showTooltip, text) {
            const tooltipContent =
document.getElementById("tooltipContent");
            if (showTooltip) {
                tooltipContent.textContent = text;
                tooltipContent.style.display = "block";
            } else {
                tooltipContent.textContent = "";
                tooltipContent.style.display = "none";
            }
        }

        document.getElementById("tooltipTrigger")
         .addEventListener("mouseenter", handleMouseEnter);
        document.getElementById("tooltipTrigger")
        .addEventListener("mouseleave", handleMouseLeave);

        const showTooltip = true;
        const text = "Tooltip content goes here.";

        updateTooltip(showTooltip, text);

        window.u1?.fix.tooltip('.tooltip-trigger', {
            selectors: {
                trigger: '.tooltip-trigger',
                tooltip: '.tooltip',
            },
        });
    </script>

</body>

</html>
```

Figure 3.45 Tooltip fixed using u1 manual selectors

3.4.19 Landmarks Use Case

Landmarks define regions of a page, allowing screen reader users to quickly jump to different sections (e.g., navigation, main content, footer). This provides a structural understanding of the page and enables efficient navigation. Without proper landmarking, users might have to tab through numerous elements to reach the desired content.

This can be fixed via u1 attributes or manual selectors, presented in the following figure 3.46 and 3.47.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Vanilla JS Playground</title>
    <script type="module" src="./script.ts"></script>
</head>
<body class="font-sans antialiased h-dvh">
    <div class="shadow-lg sticky py-3" u1-landmark="navigation">
        <div class="container">
            <a class="hover:bg-neutral-100 rounded p-2 font-
semibold"
                href="/component/button/index.html">Button</a>
            <a class="hover:bg-neutral-100 rounded p-2 font-
semibold"
                href="/component/landmark.index.html">Landmark</a>
        </div>
    </div>
    <div class="container py-4 min-h-screen" u1-landmark="main">
        <h1 class="text-2xl font-bold mb-4" u1-landmark-label-
for="main">Landmarks</h1>
        <div>Main Content</div>
        <div id="complementary-1">Complementary 1</div>
        <div id="complementary-2">
            <h2 id="complementary-2-label" class="text-lg font-
bold">Complementary Label</h2>
            Complementary 2
        </div>
    </div>
    <div class="fixed bottom-0 p-4 border-t-2 w-screen" u1-
```

```
landmark="contentinfo">
        <div class="container">
            (c)<a rel="noopener noreferrer" target="_blank"
href="https://user1st.com">User1st</a>
        </div>
    </div>
</body>
</html>
```

Figure 3.46 Landmark fixed using u1 attribute

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Vanilla JS Playground</title>
    <script type="module" src="./script.ts"></script>
</head>
<body class="font-sans antialiased h-dvh">
    <header id="banner" class="shadow-lg sticky py-3">
        <div class="container">
            <h1>Site Header</h1>
            <a class="hover:bg-neutral-100 rounded p-2 font-
semibold"
                href="/component/button/index.html">Button</a>
            <a class="hover:bg-neutral-100 rounded p-2 font-
semibold"
                href="/component/landmark.index.html">Landmark</a>
        </div>
    </header>

    <nav id="navigation" class="container py-2">
        <ul>
            <li><a href="#section1">Section 1</a></li>
            <li><a href="#section2">Section 2</a></li>
        </ul>
    </nav>

    <section id="search" class="container py-2">
        <h2>Search Area</h2>
        <form role="search">
            <label for="search-input">Search:</label>
            <input type="search" id="search-input" name="q">
```

```
            <button type="submit">Go</button>
        </form>
    </section>

    <main id="main" class="container py-4 min-h-screen">
        <h1 class="text-2xl font-bold mb-4">Main Content Area</h1>
        <div>Main Content</div>
        <div id="complementary-1">
            <h3>Complementary 1 Content</h3>
            <p>This is a complementary section.</p>
        </div>
        <div id="complementary-2">
            <h2 id="complementary-2-label" class="text-lg font-
bold">Complementary 2 Label</h2>
            <p>Another complementary section.</p>
        </div>
    </main>

    <section id="form-section" class="container py-4">
        <form id="form-1">
            <h2>Contact Us Form</h2>
            <label for="name">Name:</label>
            <input type="text" id="name">
            <button type="submit">Submit</button>
        </form>

        <form id="form-2">
            <h2 id="form-2-label">Newsletter Signup</h2>
            <label for="email">Email:</label>
            <input type="email" id="email">
            <button type="submit">Sign Up</button>
        </form>
    </section>

    <footer id="contentinfo" class="fixed bottom-0 p-4 border-t-2
w-screen">
        <div class="container">
            (c)<a rel="noopener noreferrer" target="_blank"
href="https://user1st.com">User1st</a>
        </div>
    </footer>

    <script>
        function navigate(url) {
            window.location.href = url;
        }
```

```
window.u1?.fix.landmarks({
    complementary: [
        {
            selectors: {
                landmark: '#complementary-1',
            },
            label: 'Complementary',
        },
        {
            selectors: {
                landmark: '#complementary-2',
                labelFor: '#complementary-2-label',
            },
        },
    ],
    banner: {
        selectors: {
            landmark: '#banner',
        },
        label: 'Banner',
    },
    navigation: [
        {
            selectors: {
                landmark: '#navigation',
            },
            label: 'Navigation',
        },
    ],
    main: {
        selectors: {
            landmark: '#main',
        },
        label: 'Main',
    },
    contentinfo: {
        selectors: {
            landmark: '#contentinfo',
        },
        label: 'Content Info',
    },
    search: [
        {
            selectors: {
                landmark: '#search',
            },
```

```
                    label: 'Search',
                },
            ],
            form: [
                {
                    selectors: {
                        landmark: '#form-1',
                    },
                    label: 'Form 1',
                },
                {
                    selectors: {
                        landmark: '#form-2',
                        labelFor: '#form-2-label',
                    },
                },
            ]
        });
    </script>
</body>
</html>
```

Figure 3.47 Landmark fixed using u1 manual selectors

4 Environment Setup

4.1 Installation

4.1.1 Package Manager

The u1 Toolkit can be installed through a package manager like NPM from User1st's private NPM registry.

4.1.2 CDN

The u1 Toolkit can also be embedded onto a web page via a script tag using a Content Delivery Network (CDN) hosted by User1st.

4.1.3 Direct Download

The u1 Toolkit can be distributed through an internal NPM registry or CDN, or by directly embedding it within the web application's codebase.

4.2 Configuration

4.2.1 u1 Toolkit

To gain access to both SDKs, you must register an account with a valid subscription on the u1 dashboard.

4.2.2 u1 Toolbar

Import the u1 Toolbar package or add it as a script, then integrate it into your web application using the `<u1-app>` tag. Ensure this is done only in the development environment to prevent bundling u1 Toolbar with the production application bundle.

4.2.3 u1 A11y

Import the u1 A11y package or add it as a script. Additionally, ensure the u1.css file is included in your web application, either in the main entry point file or the primary CSS file, or by adding `u1.css` as a link.

There are also additional configurations that can be set using the setu1Configuration function.

5 Technology Stack

All SDKs in the u1 Toolkit are implemented in TypeScript [9].

5.1 u1 Toolbar

The u1 Toolbar is implemented as a React-based [10] micro-front [11] web component leveraging the shadow DOM [12] for style isolation.

5.2 u1 A11y

u1 A11y follows the Object-Oriented Programming paradigm and patterns such as singleton and dependency injection. It also utilizes the Sizzle CSS selector engine [13] to enable jQuery selectors.

6 Discussions

6.1 Performance Metrics

6.1.1 u1 A11y

6.1.1.1 Bundle Size

The u1 A11y library has a bundle size of 258.5 KB (73.1 KB compressed). This should incur minimal cost to the bundle size of the web application.

6.1.1.2 Network Load Time

6.1.1.2.1 Package & Direct Download

The additional load time incurred by adding the u1 A11y library depends on the user's network speed, geographic location, bundle splitting strategy & number of functions imported from the library. Developers can optimize load times by employing efficient bundling techniques, tree shaking and using CDNs.

6.1.1.2.2 CDN

When using a CDN, the load time also depends on the user's network speed and geographic location. CDNs typically offer faster load times due to their distributed nature and proximity to end-users.

6.1.1.2 Initialization

The maximum initialization latency of u1 A11y is influenced by the user's network speed and geographic location due to an API call required to verify the domain is whitelisted. Additionally, there is a 1-second delay from u1 A11y's page polling architecture.

6.1.2 u1 Toolbar

6.1.2.1 Bundle Size

As a development tool, the u1 Toolbar does not add to the production bundle size, ensuring that end-users do not experience any performance degradation due to its inclusion.

6.1.2.2 Network Load Time

The network load time for the u1 Toolbar is also not a concern for end-users, as it is used solely in the development environment. However, during development, load times may vary based on the developer's network speed and geographic location [14].

6.1.2.3 Initialization

The initialization of the u1 Toolbar is dependent on the number of elements on the web page. Pages with a larger number of elements may experience longer initialization times. Despite this, the u1 Toolbar is optimized to provide timely feedback to developers, aiding in efficient accessibility debugging and remediation.

6.2 Limitations

6.2.1 u1 Toolbar

The u1 Toolbar can only perform dynamic analysis of the DOM, not the source code of the web application. In other words, it assesses the results of code execution rather than the code itself.

6.2.2 u1 A11y

u1 A11y currently only supports the following components:

- Accordion;
- Canvas;
- Carousel;

- Button;
- Checkbox;
- Link;
- Radio;
- Datepicker;
- Dialog;
- Form;
- Loading;
- Menu;
- Pagination;
- Select;
- Table;
- Table Grid;
- Tab;
- Tooltip;

Additionally, although rare, u1 A11y cannot always override behaviors implemented in a component. Therefore, making the component accessible may sometimes require the developer to modify the component's source code [15].

7. Future Plans

The u1 Toolkit is under continuous development and adjustment to meet new requirements, regulations, framework updates, and dependency updates. Our current future plans for the u1 Toolkit include:

- AI Integration: We aim to integrate artificial intelligence to enhance the detection and remediation of accessibility issues, providing more intelligent and automated solutions [16].

- WCAG Updates: We are committed to staying aligned with the latest Web Content Accessibility Guidelines (WCAG) updates, ensuring our tools provide up-to-date compliance and best practices [17].
- New SDK Versions: We plan to release new versions of the SDK with enhanced features, improved performance, and broader component support, continuously evolving to meet the needs of developers and accessibility standards.

Revisions

Date	Author	Changelog
09.08.2024	Xiaoyu (Aki) Gao	• Initial Document Version
12.08.2024	Vasile Nastas	• Added Revision Sections
16.08.2024	Xiaoyu (Aki) Gao	• Added Xiaoyu (Aki) Gao's initial document version revision
11.01.2025	Vasile Nastas	• Adjusted and formated document

References

1. Best CMS platforms, CloudWays 2024 [Online]. Available: https://www.cloudways.com/blog/best-cms-platforms/.
2. WordPress, 2024 [Online]. Available: https://wordpress.com/.
3. Shopify, 2024 [Online]. Available: https://www.shopify.com/.
4. Magento, 2024 [Online]. Available: https://magento-opensource.com/.
5. Umbraco, 2024 [Online]. Available: https://umbraco.com/.
6. Joomla, 2024 [Online]. Available: https://www.joomla.org/.
7. W3C, "Web Content Accessibility Guidelines (WCAG) 2.1," w3.org. [Online]. Available: https://www.w3.org/WAI/WCAG21/quickref/.
8. TypeScript, 2024 [Online]. Available: https://www.typescriptlang.org/.

9. Meta, “React - A JavaScript library for building user interfaces,” react.dev. [Online]. Available: https://react.dev/.
10. Mozilla, “Using Shadow DOM,” mozilla.org. [Online]. Available: https://developer.mozilla.org/en-220US/docs/Web/Web_Components/Using_shadow_DOM.
11. L. Tilkov and S. Lange, “Micro Frontends: The Microservice Approach to Front-End Development,” ThoughtWorks. [Online]. Available: https://www.thoughtworks.com/en-us/radar/techniques/micro-frontends.
12. Mozilla, “Using Shadow DOM,” mozilla.org. [Online]. Available: https://developer.mozilla.org/enUS/docs/Web/Web_Components/Using_shadow_DOM.
13. jQuery Foundation, “Sizzle Selector Engine Documentation,” sizzlejs.com.[Online]. Available: https://sizzlejs.com/.
14. Mozilla, “Web Performance Optimization,” mozilla.org. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/.
15. H. L. Antonelli, L. Sensiate, W. M. Watanabe, and R. 235 P. de Mattos Fortes, “Challenges of Automatically Evaluating Rich Internet Applications Accessibility,” dl.acm.org. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3328020.3353950.
16. S. Abou-Zahra, J. Brewer, and M. Cooper, “Artificial Intelligence (AI) for Web Accessibility: Is Conformance Evaluation a Way Forward?” Web4All 2018, 23-25 Apr., 2018, Lyon, France, ACM, 2018. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/116479/AI.pdf?isAllowed=y&sequence=1.
17. W3C, “The Evolution of WCAG: Preparing for WCAG 3.0,” w3.org. [Online]. Available: https://www.w3.org/WAI/standards-guidelines/wcag/wcag3-intro/.