



## Assignment of master's thesis

<b>Title:</b>	Web Components UI Library
<b>Student:</b>	Bc. Dominik Fryč
<b>Supervisor:</b>	Ing. Josef Pavlíček, Ph.D.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Web Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2023/2024

### Instructions

Front-end development is a complex process of creating a user interface between humans and applications. Developers often utilize component libraries to save time, not repeat themselves and achieve consistency. This diploma project aims to build a Web Components library based on a custom design system. Components should be standards-based, performant, accessible and extensible. The design language is expected to be customizable by modifying design tokens. The library should implement enough components to build a web admin interface.

Requirements for the thesis:

- Explore design systems and UI components
- Compare and analyze existing JavaScript UI libraries
- Analyze web components technology and frameworks
- Create a design system including tokens, components and templates
- Implement, document and test a component library based on the design system
- Deploy and distribute the library under an open-source license
- Demonstrate the actual usage of created components in an application
- Evaluate the result and discuss the possibilities for further development





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Master's thesis

## **Web Components UI Library**

*Bc. Dominik Fryč*

Department of Software Engineering  
Supervisor: Ing. Josef Pavlíček, Ph.D.

February 15, 2024



---

## **Acknowledgements**

I would like to express my gratitude to my thesis advisor Ing. Josef Pavlíček, Ph.D. for helpful advice during consultations. Also, I would like to thank my family and close ones for their support during the whole studies.



---

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on February 15, 2024

Czech Technical University in Prague

Faculty of Information Technology

© 2024 Dominik Fryč. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Fryč, Dominik. *Web Components UI Library*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.



---

# Abstract

In today's dynamic web development landscape, a well-designed and well-crafted UI library can significantly enhance web designers' and developers' efficiency and productivity. Web Components, a modern technology standard, offer a promising approach to creating reusable and encapsulated UI components that can seamlessly integrate across different web frameworks. This diploma thesis aims to create a component library, establish a custom design system, provide comprehensive documentation, and distribute the package to a public repository. It dives into the comprehensive process of designing and building a component library, containing exploration and analysis, design system creation, component library implementation, testing, distribution and usage. It creates a valuable resource for any developer interested in developing the Web Components UI library.

**Keywords** components, web components, custom elements, component library, UI components, design system, Lit, Vite, TypeScript, Storybook



---

# Abstrakt

V dnešní dynamické době vývoje webu může dobře navržená a dobře vytvořená knihovna uživatelského rozhraní výrazně zvýšit efektivitu a produktivitu webdesignérů a vývojářů. Web Components, moderní technologický standard, nabízí slibný přístup k vytváření opakovaně použitelných a zapouzdřených komponent uživatelského rozhraní, které se mohou bezproblémově integrovat napříč různými webovými frameworky. Tato diplomová práce má za cíl vytvořit knihovnu komponent, založit vlastní design systém, poskytnout komplexní dokumentaci a distribuovat balíček do veřejného repozitáře. Zabývá se celým procesem návrhu a stavby knihovny uživatelského rozhraní, která obsahuje průzkum a analýzu, vytváření design systému, implementaci knihovny komponent, testování, distribuci a její použití. Představuje cenný zdroj pro každého vývojáře, který má zájem o vývoj vlastní knihovny uživatelského rozhraní pomocí technologie Web Components.

**Klíčová slova** components, web components, custom elements, component library, UI components, design system, Lit, Vite, TypeScript, Storybook



---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Exploration and Analysis</b>	<b>3</b>
1.1 Design Systems . . . . .	4
1.2 Component Libraries . . . . .	6
1.3 Web Components Technology . . . . .	9
1.4 Web Components Frameworks . . . . .	12
<b>2 Design System</b>	<b>15</b>
2.1 Motivation . . . . .	16
2.2 Principles . . . . .	16
2.3 Realization . . . . .	18
2.4 Design Tokens . . . . .	19
2.5 Components . . . . .	28
<b>3 Implementation</b>	<b>35</b>
3.1 Project Structure . . . . .	36
3.2 Build Process . . . . .	37
3.3 Documentation . . . . .	40
3.4 Linting and Formatting . . . . .	42
3.5 Forms and Validation . . . . .	43
3.6 Accessibility . . . . .	44
<b>4 Testing</b>	<b>47</b>
4.1 Strategies and Types . . . . .	48
4.2 Used Tools . . . . .	50
4.3 Test Cases . . . . .	50

<b>5</b>	<b>Distribution and Usage</b>	<b>55</b>
5.1	Release . . . . .	56
5.2	Components Usage . . . . .	59
5.3	Customization . . . . .	63
	<b>Conclusion</b>	<b>69</b>
	<b>Bibliography</b>	<b>71</b>
<b>A</b>	<b>List of Abbreviations</b>	<b>79</b>
<b>B</b>	<b>Contents of Attachments</b>	<b>81</b>

---

# List of Figures

1.1	Stencil vs. Lit downloads in past 2 years on npm [31] . . . . .	13
2.1	Design tokens: Primitive color palette . . . . .	21
2.2	Design tokens: Color aliases . . . . .	22
2.3	Components: Alert . . . . .	29
2.4	Components: Button . . . . .	30
2.5	Components: Checkbox, Switch, and Radio . . . . .	31
2.6	Components: Choice Group . . . . .	32
2.7	Components: Input and Textarea . . . . .	33
2.8	Components: Label . . . . .	34
2.9	Components: Spinner . . . . .	34
3.1	Project file structure . . . . .	36
3.2	Component file structure . . . . .	37
5.1	Comparison between light and dark theme . . . . .	65
5.2	Custom theme example . . . . .	66





---

## List of Tables

2.1	Design tokens: Font family . . . . .	23
2.2	Design tokens: Font size . . . . .	24
2.3	Design tokens: Font weight . . . . .	24
2.4	Design tokens: Line height . . . . .	25
2.5	Design tokens: Border radius . . . . .	25
2.6	Design tokens: Box shadow . . . . .	26
2.7	Design tokens: Spacing . . . . .	26
2.8	Design tokens: Transitions . . . . .	27
2.9	Design tokens: Z-index . . . . .	27
2.10	Design tokens: Forms . . . . .	28



---

# List of Listings

1.1	Simple web component definition . . . . .	12
3.1	TSDoc annotation example for Custom Elements Manifest . . . . .	42
4.1	Test case: Checkbox is defined . . . . .	51
4.2	Test case: Spinner renders with default values . . . . .	52
4.3	Test case: Label renders with custom attributes correctly . . . . .	53
4.4	Test case: Input passes the accessibility audit . . . . .	53
4.5	Test case: Button fires <i>dfx-click</i> event when clicked . . . . .	54
5.1	Conventional Commits format . . . . .	56
5.2	Example of conventional commit . . . . .	57
5.3	Component usage in React . . . . .	61
5.4	Component usage in Vue . . . . .	62
5.5	Component usage in Angular . . . . .	63
5.6	Custom theme usage . . . . .	66
5.7	CSS Custom Properties example . . . . .	67
5.8	CSS Shadow Parts example . . . . .	67



---

# Introduction

Design systems and components play an essential role in modern software development. A design system is a collection of components, principles, and templates that create consistency and efficiency in the design and development process. It provides a unified visual language and a set of rules for building user interfaces. Components are self-contained, reusable elements that can be combined to create complex user interfaces. It simplifies workflow for designers and developers, improves collaboration, and delivers consistent, high-quality user experiences across different platforms.

Finding a reliable and efficient component library could be a struggle. Existing libraries often come with challenges that block the development process. One common problem is the need for more customization options. Many libraries offer limited pre-designed components, making it challenging to tailor them to specific project requirements. Additionally, compatibility issues arise when integrating these libraries with existing codebases, leading to time-consuming debugging and refactoring tasks. Another issue is the lack of documentation, which could cause the learning curve and troubleshooting process. These problems highlight the need for a powerful and flexible component library that addresses these pain points and supports developers in creating highly customizable user interfaces.

Usually, a suitable component library comes in handy when working on a new front-end project. First, I started using popular UI libraries to build a solid foundation for my work. However, I soon realized that none of them met my expectations. So, I started creating custom components that fit my needs for a particular project. For new projects, all those components were copied from previous ones, but when a component was changed in one project,

I had to distribute the changes to all other projects. It wasn't effective and sustainable at all. Moreover, I started working on a project using a different front-end framework, so my existing components weren't compatible. That's why I chose to create a custom component library.

This thesis aims to create a component library that will be compatible with all front-end frameworks and easy to use and customize. Components should be efficient, accessible by default, and standards-based, following W3C Web Components specifications [1]. The project will be accompanied by a design system providing design tokens, components, and templates demonstrating components composition. Modifying design tokens should customize the design language. Documentation should provide information about all components and their usage. The library will be distributed as a package at a public repository and published under an open-source license.

The thesis follows a logical structure, starting with analyzing and exploring web components technology. This initial section provides a complete understanding of the technologies and their relevance in modern software development. Following the analysis, the thesis dives into creating a design system, individual design tokens, and components. Implementation focuses on the practical aspects of building a JavaScript component library. This section covers the design and development process, including the tools and technologies used. The following section describes the testing strategies and test cases. Next, the thesis covers the distribution and usage of the component library. This section explores how the library can be easily installed and integrated into different front-end frameworks and how to customize it for various projects. Finally, the thesis concludes with a project evaluation that reflects on the challenges faced during the development process. It includes a summary of the achieved goals and future project improvements.

---

# Exploration and Analysis

This chapter explores and analyzes various aspects of front-end development, focusing on design systems, component libraries, and web components technology. Developing a design system that creates efficient and consistent UI development requires a robust foundation. I will describe and compare key technologies and tools that will serve as the building blocks for my project.

In the first section, I will discuss the purpose and benefits of design systems in modern web development. I explore how these systems establish a single source of truth for design principles, UI components, and development practices, leading to faster development, improved consistency, and enhanced maintainability.

The second section will introduce component libraries and a comparative analysis of development approaches with existing solutions. I describe the strengths and weaknesses of different component library options, including framework-specific and framework-agnostic libraries. This comparative analysis will inform my choice of the most suitable approach for my design system.

Next, I dive into the heart of my chosen approach: web components technology. This section demystifies the key concepts that power custom elements, including Shadow DOM, HTML templates, CSS custom properties, CSS parts, slots, and events. Understanding these mechanisms is crucial for building robust web components.

Finally, I write about the landscape of web component frameworks. I explore the advantages and disadvantages of libraries Lit and Stencil, considering their underlying implementation approach, ease of use, community support, and performance characteristics. This analysis will guide me in selecting the most suitable framework for my project's needs.

By the end of this chapter, I will provide a firm understanding of the technologies and tools that will form the foundation for my component library. This exploration will prepare me to dive into the project's design and implementation phases.

### 1.1 Design Systems

A design system is a set of components and principles that reflect a brand or product's visual language and user experience standards. It serves as a single source of truth for the whole team, providing a common language and clear guidelines for all team members. Design systems typically include design tokens, components, and templates, which work together. [2] There are few reasons why design systems matter [3]:

**Accessibility** – The Design system should make sure that it uses an accessible color palette with sufficient contrast ratios, large enough font sizes for all texts, and that all components are usable with a keyboard or a screen reader.

**Consistency** – One of the key advantages of design systems is that they promote consistency in user interfaces. By reusing components and following defined standards, a design system can provide a consistent user experience across different parts of an application or even different applications.

**Updates** – A design pattern or specific component can be updated at one place within the system, and the change will propagate to each product after a library update. This could also be a disadvantage since some changes could break existing behavior, and developers must fix it.

**Onboarding** – Well-documented design systems are a great source for onboarding new team members. Having one source of truth can make understanding the whole system faster and easier.

**Speed** – With a stable release of the design system, designers and developers can prototype new applications rapidly.

#### 1.1.1 Design Tokens

One of the key elements of design systems is the concept of design tokens. Design tokens abstract design properties such as color, typography, spacing,



motion, and elevation into reusable and platform-agnostic variables. By defining these properties at a higher level and using them in place of hard-coded values, design tokens maintain consistency across various platforms and design tools while allowing for easy customization and adaptation to different contexts. [4][5]

### 1.1.2 Patterns and Templates

Patterns and templates are integral parts of design systems that encapsulate commonly used design solutions and layouts. Patterns represent reusable solutions to recurring design problems, while templates provide predefined structures for specific page layouts or interface elements. By incorporating patterns and templates into design systems, teams can correctly and consistently implement design solutions, reducing user cognitive load and streamlining the design and development process. [6]

### 1.1.3 Components

Components are the fundamental units of any design system to create visual and functional consistency. They are reusable, self-contained pieces of user interface that serve a specific function. Components can range from simple elements like buttons and inputs to complex elements like data tables and modals. Components are supposed to be modular, so they can be combined and composed together to create more complex user interfaces. [7]

Each component in the library is a self-contained unit of code, typically consisting of HTML, CSS, and JavaScript. A design system should have elements that are reusable, customizable, and accessible. Reusability ensures that the components can be used in a variety of contexts. Customizability allows the components to be adapted to fit different requirements. Accessibility ensures that the components can be used by any user, including those with disabilities.

There are some examples of high-quality design systems:

- Nord by Nordhealth
- Orbit by Kiwi
- Polaris by Shopify
- Fluent 2 by Microsoft
- Primer by GitHub

## 1.2 Component Libraries

The component library is the implementation of the design system. It represents a coded, tested, and well-documented collection of reusable UI components that follow the principles and guidelines established within the design system and are ready to be used inside a project. Using component libraries, developers can reuse code across multiple projects, reducing the time required to create new applications. This is particularly beneficial in large teams where consistency in design and functionality is crucial. [8] [9]

### 1.2.1 Framework-specific vs. agnostic library

The most challenging part about creating a new component library is deciding which development approach should be used. There are multiple ways to implement a component library. It could be a framework-specific approach when the components only work with a particular JavaScript framework. On the other hand, it could be a framework-agnostic approach, so the library works within any JavaScript project. Let's compare the advantages and disadvantages of both methods.

#### **Framework-specific approach**

This option makes the most sense for teams that have selected a single framework for all their applications or as their preferred option for new apps. This could be due to various reasons, such as the framework's robustness, ease of use, or specific features that align with the company's needs.

Framework-specific component libraries offer several key advantages. Firstly, they provide seamless integration with the framework's ecosystem, including its tools, components, and libraries, simplifying the development process. Secondly, these libraries are designed to fully utilize the framework's features and optimizations. This leads to efficient rendering and minimal overhead, enhancing the performance of the applications built with them. Another significant advantage is the speed of the development process. These libraries' ready-made components save valuable development time and ensure consistency with the framework's conventions. Lastly, these libraries often come with extensive documentation and community support specific to the framework, making it easier for developers to learn and debug problems. [10]

They also come with certain limitations. One of the main downsides is the lack of portability. These libraries are not usable outside the chosen framework, which limits their reuse potential across different projects. This leads to another issue known as framework lock-in, where the project becomes tied to a specific framework. This could potentially hinder future technology shifts and limit flexibility. Additionally, there's a learning curve associated with these libraries. Users who are unfamiliar with the framework may face challenges when adopting and customizing components from the library. Lastly, these libraries often come with an opinionated structure. They may enforce specific design patterns, or ways of working that may not fit all needs, potentially limiting creativity and adaptability. [11]

Here are some examples of framework-specific component libraries:

- Chakra UI
- Material UI
- Mantine
- Angular Material
- Vuetify

### **Framework-agnostic approach**

This approach relies on web components APIs [12] supported by all modern browsers, ensuring their future-proof relevance. It involves creating components based on web standards, such as HTML, CSS, and JavaScript, without relying on any specific framework like React, Angular, or Vue. As a result, components can be used in any JavaScript application, regardless of its framework or library, or even without any framework.

Web components offer a unique set of advantages due to their encapsulation. This encapsulation ensures that their implementation is self-contained, preventing styles and scripts from leaking out or being influenced by the rest of the application. This leads to a high degree of flexibility, offering more control over design and behavior without the constraints of a specific framework. Furthermore, web components leverage native browser APIs, which ensures broad compatibility and consistent behavior across all modern browsers. [13]

While web components offer numerous advantages, they also present specific challenges. Firstly, the learning curve of developing and maintaining web components can be steep for some developers. This is because it requires

a deeper understanding of web standards and browser APIs, which can increase the complexity of the development process. Secondly, web components might require additional setup and configuration compared to framework-specific libraries. This could slow the initial development phase and increase the complexity of integrating these components into an application [14]. Lastly, the community support for web components might be smaller than popular frameworks. Fewer resources might be available for learning and troubleshooting, which could pose challenges for developers, especially those new to web components.

These examples represent the framework-agnostic approach:

- Shoelace
- FAST
- Ionic
- Clarity
- Baklava

The ideal choice depends on specific needs and constraints. In modern front-end development, there are many popular choices of frameworks, and new ones are released every year. Even though the most popular library, React, will not disappear from one day to another, the "best" framework could change, and every developer has a different opinion. Large companies often use various frameworks for different applications based on a specific team, their experience, and project needs. [15] [16]

In conclusion, one of the features of my design system is compatibility with any JavaScript framework or no framework at all, so for that reason, I tend to use framework-agnostic web components technology. I want to keep the component library future-proof so I can use it no matter what current popular technology stack I will be using. Thanks to the integration of web components in modern browsers, I have the assurance of future compatibility.

### 1.2.2 Single vs. multi-package library

Another aspect to consider during the component library development is how end-users will use it in their application. Should they consume the library as a single package or multiple component packages? Each approach has

advantages and disadvantages, and the choice often depends on the project's specific needs. [17]

Most libraries are distributed as a single-versioned package, meaning all components are bundled into a single package. This approach simplifies the process of versioning and dependency management, as there is only one version of the library to maintain. It makes the most sense for tightly coupled libraries, where components depend on each other. There could be a concern about application bundles containing unused components. The Open WC offers best practices for publishing web components, one of which recommends exporting the library as ESM. Doing so means the consumer application can remove unused code using a bundler during the build process [18]. One disadvantage could be that updating a single component requires creating a new version for the whole component library.

On the other hand, the multi-package approach, often implemented in a monorepo using tools like Lerna, involves creating a separate package for each component. This approach allows consumers only to install the components they need, leading to potentially smaller bundle sizes. It also allows for independent versioning of each component, which can be beneficial when different components evolve at different rates. However, this approach can complicate dependency management and increase the library's overhead, as each component needs to be versioned and published separately. [19]

Since most of my components are composed of other components, there is a strong dependency between them, so adding all components into a single package makes more sense. My component library's main consumer use case for development is to utilize most components rather than just one component. Also, as mentioned, versioning and publishing a single package is much easier than maintaining a monorepo with multiple component packages. [20]

## 1.3 Web Components Technology

Web Components are a groundbreaking technology in web development, allowing developers to package HTML, CSS, and JavaScript to make a custom element reusable across browsers and other web technologies. They are made of existing standards and Web APIs browsers have implemented over the years. They now have enough use and maturity to challenge the existing popular frameworks. Web Components contain several key specifications, including Custom Elements, Shadow DOM, and HTML Templates. [1] [12]

The concept of web components started in the early 2010s [21]. Web components initially held the promise of standardized reusable UI components. However, their adoption was initially limited by browser inconsistencies and a lack of robust tooling. The turning point arrived in 2018 with W3C standardization of Custom Elements v1, which became available in every ever-green browser: Chrome, Safari, and Firefox. Despite initial challenges such as browser support and ecosystem maturity, web components have gained significant momentum, with major browsers now offering full support [22].

### 1.3.1 Custom Elements

Custom Elements are a key feature of Web Components, enabling developers to define and use new types of DOM elements in a document. They extend the set of HTML elements available in the browser, allowing developers to create reusable components with encapsulated functionality. Custom Elements are implemented as classes that extend `HTMLElement` or the interface you want to customize. The specification provides a flexible approach to component-based development, allowing developers to define custom elements with their lifecycle callbacks, properties, attributes, methods, and events. [23]

### 1.3.2 Shadow DOM

The Shadow DOM (Document Object Model) encapsulates styles and functionality within custom elements. At its core, the Shadow DOM provides a scoped subtree for a custom element, isolating its internal structure, styling, and behavior from the rest of the document. This encapsulation ensures that the styles defined within the Shadow DOM are scoped to the component, preventing unintended conflicts. As a result, developers can create custom elements with consistent styling and behavior, even in the presence of complex CSS frameworks or third-party libraries. [24]

Shadow DOM, while powerful, does have some challenges associated with it. One of the main issues is that Shadow DOM can cause layout shifting during the page load or show a flash of unstyled content (FOUC) before the Shadow Root's stylesheets are fully loaded [25]. Declarative Shadow DOM is a web platform feature that addresses some of these limitations [26]. It introduces a new way of defining a shadow tree directly in HTML, which is particularly useful for server-side rendering of Web Components and in contexts where JavaScript is disabled. A Declarative Shadow Root can be created by a `<template>` element using a `shadowrootmode` attribute. When

an HTML parser detects it, it sets the content as the shadow root of its parent element. This feature removes the limitation of expressing Shadow Roots in the server-generated HTML, making Shadow DOM more accessible for server-side rendering and static HTML generation.

Customizing the Shadow DOM from outside can be achieved through two primary methods: CSS Custom Properties and CSS Parts. CSS Custom Properties are reusable values that can be used throughout the stylesheet. It enables developers to define custom properties within the Shadow DOM that can be dynamically changed from outside the component. Outside of CSS variables, only color and font properties are inherited inside the Shadow DOM. CSS Parts offer a mechanism for exposing specific elements within the Shadow DOM to the parent document, allowing the styling of these elements from outside. Developers can specify which elements should be exposed and customizable by defining CSS Parts within the Shadow DOM. This provides a precise and controlled way to customize the component's appearance without breaking encapsulation. [27]

### 1.3.3 HTML Templates

Templates are a fundamental part of web components. They provide a declarative way to define the structure that can be used to populate the Shadow DOM of a web component. An HTML template is determined using the `<template>` element. The content inside this element is not rendered in the DOM, but it can be referenced and used in JavaScript to append it to the Shadow DOM. [28]

Moreover, HTML templates also support `<slot>` elements, which are placeholders that can be filled with any markup projected inside the template. This slot-based content distribution mechanism adds flexibility to the templates, making them more reusable and customizable to adapt to different use cases and contexts.

Let's create a simple web component to demonstrate the technologies mentioned above. First, I made a class `HelloWorld` which extends `HTMLElement`. Inside the constructor, I added an HTML template with one default slot. Then, I attached Shadow DOM to the class in open mode and set my template as content. This web component can then be used in HTML as `<hello-world>John</hello-world>` element, which renders "Hello John!".

Listing 1.1: Simple web component definition

```
class HelloWorld extends HTMLElement {
  constructor() {
    super();
    const template = document.createElement('template');
    template.innerHTML = `Hello <slot>world</slot>!`;
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.append(template.content.cloneNode(true));
  }
}

customElements.define('hello-world', HelloWorld);
```

## 1.4 Web Components Frameworks

Since web components are built on web standards and technologies, they could be written in plain JavaScript. The most significant advantage is that this approach does not rely on external dependencies, so no framework knowledge is required. Plus, the developer has complete control over the component's behavior. On the other hand, there is a massive overhead of unnecessary code. Writing custom elements in vanilla JavaScript could become more lengthy. [29]

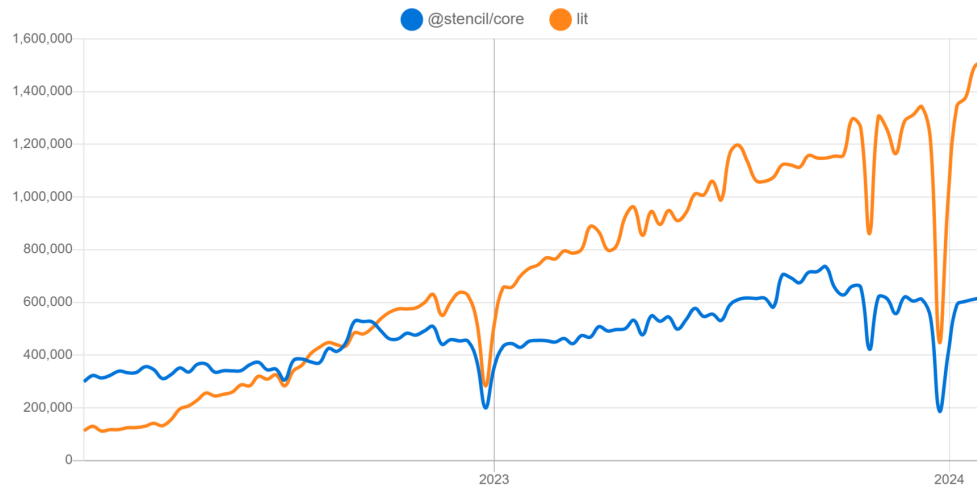
There are about 60 other ways to write web components besides plain JavaScript [30]. One uses a front-end framework like React, Angular, or Vue to generate web components. React users can use a simple wrapper, Angular uses Angular Elements API, and Vue has a `defineCustomElement` method. While this method seems useful, web component generation could be bloated by the framework and its dependencies, which could strongly affect rendering performance. Although these frameworks can generate web components, they are not built for it in the first place.

A better option is using a library designed to create custom elements. The most significant advantage of this approach is the library's abstraction, tooling, and conventions. A few years ago, the most popular choice was Stencil. It is a compiler that generates web components in plain JavaScript. This tool is built by Ionic team, which develops software to support multi-platform development. According to Figure 1.1, Lit is the most used option for creating web components nowadays. It is maintained by Google, and it became



a successor of the deprecated framework Polymer. I won't mention other libraries because they are too small compared to Lit and Stencil, so there are insufficient resources and small or nonexistent communities.

Figure 1.1: Stencil vs. Lit downloads in past 2 years on npm [31]



### 1.4.1 Stencil

Stencil is a modern web component compiler offering much out of the box. Component code is defined in classes, encapsulating HTML structure, CSS styles, and TypeScript logic within a single unit. Template syntax is using JSX. It is converted into lightweight standards-compliant JavaScript code that can run in any modern web browser. Stencil provides robust testing tools, a comprehensive build process, and integration with popular front-end frameworks, making it easy for developers to get started.

Stencil uses a compiler-based approach to generate optimized code and provide good performance thanks to its efficient rendering engine utilizing a virtual DOM. While Stencil aims to simplify the creation of web components, there may be knowledge required for developers unfamiliar with its architecture and tooling. The virtual DOM implementation might introduce a small performance overhead compared to vanilla JavaScript solutions. Stencil has a smaller community and ecosystem than other front-end frameworks and libraries, which could impact the availability of third-party plugins, extensions, and support resources. [32]

### 1.4.2 Lit

Lit is a tiny library for building web components. Thanks to its small size, the library loads fast and minimizes the bundle size of the package. It builds on top of web components standards, so the template looks similar to plain JavaScript. It offers features like TypeScript support and hot module reloading. The project setup can be adjusted since it doesn't include any build or testing tools, but the ecosystem is broad, and Lit is easy to integrate with all popular libraries.

It doesn't use a virtual DOM. Instead, it directly manipulates the real DOM for efficient updates. It uses template literals with HTML-like syntax for defining component templates, offering a familiar and declarative approach with efficient rendering capabilities. Directives are JavaScript decorators that enable developers to define reactive behavior, event handlers, and data bindings within a component. By combining templates and directives, developers can create web components with minimal overhead.

Its easy-to-understand syntax, performant DOM updates, and focus on maintainability make it a robust library for creating web components. An active community provides a rich ecosystem of resources, tutorials, and third-party libraries. One of the drawbacks is the lack of routing or state management, which could be found in other libraries. There may be a learning curve for developers unfamiliar with its template and directive syntax. [33]

### 1.4.3 Conclusion

Lit has grown in popularity for the last two years while Stencil has stagnated. Most of the component libraries of big companies like Spectrum from Adobe, Material Web from Google, and Lion from ING use Lit, while I haven't found almost any open-source projects that use Stencil. I read an article from Cory LaViska, the creator of the popular web component library called Shoelace, who explains why they moved from Stencil to Lit [34]. Some reasons include a lack of documentation, difficulty debugging, and a closed-source bug-reporting system. It is quite a complex tool, and using it could feel like a black box. The benefits of Lit he mentioned include more control over the build process, the use of standard web technologies, and a more open development community. I will use Lit to implement my component library for this and other mentioned reasons.

---

# Design System

This chapter dives into creating and implementing a custom design system, exploring its motivation, principles, realization, design tokens, and components. It serves as a guide for understanding the system's structure, functionality, and benefits, which using the design system can bring.

The motivation behind creating this design system originated from the inability to find an existing system that would fit the needs of particular projects. The desire for a more tailored solution led to the development of a unique design system that could adapt to specific requirements and preferences.

The following section introduces principles that guide this design system, including adaptability, modularity, efficiency, customizability, usability, transparency, and accessibility. These principles form the system's foundation, ensuring it is flexible, efficient, user-friendly, and accessible to all users.

The design system was realized using Figma, a versatile design tool known for its collaborative features and robust functionality. The pros and cons of Figma and its role in implementing the design system are discussed in detail. The structure of the design system file is also described.

This section focuses on design tokens and explains three token types, naming conventions, and my decisions during creation. These tokens cover categories such as color, typography, border radius, box shadow, spacing, transitions, z-index, and forms.

Finally, the section about components discusses the various elements implemented in the design system. These include Label, Spinner, Button, Alert, Input, Textarea, Checkbox, Radio, and Choice Groups. Each component is designed with the system's principles in mind, ensuring consistency and usability across the design.

### 2.1 Motivation

Third-party design systems and component libraries are already implemented, documented, and tested. Some of the good ones are even open-sourced. At first, it seems like a good start for a project. But I couldn't find the one that would exactly meet my expectations. There are a few aspects that stopped me from using those. No matter how good the component library was, those solutions were designed for a particular product's or company's needs. It resembles someone else's brand. They follow their design principles and guidelines to accomplish their goals, which usually aren't the same as my goals.

The second issue is that all components have a specific default styling and behavior. To ensure the best developer experience, I want to use components out of the box, but if I don't sympathize with their default state, I can't use them without any changes. Most offer some level of customization, but there is usually a case that is not covered. I used to end up in situations when I used 80 percent of components as they were, but those last 20 percent were heavily customized or hacked for my particular use case. There is a point when the initial benefits of using a third-party library are outweighed by the time spent on modifying and rewriting the library.

Even though my component library will probably suffer from these issues too, I'll try to minimize their impact. The goal is to make it as aesthetic-agnostic as possible and provide a wide variety of options for customization. Eventually, this design system is created for my needs and wants, so it fits my projects perfectly in the first place. However, some developers or organizations could share the same key values of design principles, so I decided to open it to the public.

### 2.2 Principles

In the first chapter of this thesis, I mentioned the advantages of design systems and why they matter. Those are general observations, and they apply to most projects. However, different design systems have different goals and priorities. I would like to dedicate this section to explaining my design principles. Each of those seven categories is crucial for creating this design system, and together, they create the foundation that helped me make important decisions during the design and development.

**Adaptability** – Components should be built on modern web standards, making them future-proof and long-lasting. They should be able to integrate with any web technology. The design system should follow the new trends and technologies to support innovations while maintaining a standards-based approach.

**Modularity** – The design system should be modular and scalable. Components should function as independent units so they can be composed to create more complex elements. Each component should be extensible to add custom behavior.

**Efficiency** – All components should be built as minimal and lightweight. They should use the web platform to the maximum extent possible to ensure the best performance. The library should ideally have no dependencies. Most of the tasks and processes should be automatized.

**Customizability** – Components should come with a minimum of styles so they can be extensively customized to match the brand of the project or organization. Modifying design tokens should adjust the design language to maintain consistency and quality of UI elements.

**Usability** – The library usage should be easy and enjoyable to offer a great developer experience. This could be achieved by setting good default aesthetics and predictable behavior to use components out of the box. Documentation should be understandable and intuitive without deep understanding and support maintainability.

**Transparency** – Sharing knowledge and understanding should be a high priority. The design system should be built transparently to increase visibility and openness. The project should be published under an open-source license to encourage contributions and collaboration.

**Accessibility** – The library should ensure the user interface is highly accessible to everyone. Users with special needs should have the same experience as others. The design system should be user-centered and inclusive. Components should be internationalized to remove all language barriers.

### 2.3 Realization

I chose Figma as my primary tool to create the design system. It is the most popular design tool for creating interface design. The main features are vector editing, real-time collaboration, and prototyping. Figma has a large plugin ecosystem, so it is possible to extend its functionality. It offers a dev mode, which provides tools for translating design into code. Figma is an excellent option for creating design systems. Designers can create reusable components, styles, and variables that can be used throughout the project. You can also build a library to distribute the design system to other projects. It helps set guidelines and ensure consistency in design. [35]

The most significant advantage is its cross-platform compatibility. Since it is a cloud-based tool, you can use it on any system using the web browser. Figma has an automatic version control, so it is possible to track and revert changes or switch to different branches. The UI of the application is user-friendly and easy to understand. On the other hand, the free plan is not unlimited, and paid plans can be expensive. It requires internet access, so there is no offline functionality. While the web-based solution is excellent, there could be performance issues with big projects on slower devices. Are there any alternative tools? Not really. Sketch is only available on a Mac, so I can't use it, and the main Figma competitor, Adobe XD, just got discontinued [36].

My Figma project follows a logical structure. The first page is a cover image for the design system, including the logo. The next page is all about design tokens. Each section contains a category for styles like color, typography, border radius, and box-shadow. There are also icon definitions, which are used across the project. After that, there is a page with all the designed components. A component section contains its name, description, and variants with different states. Last but not least is the template page. It shows the usage and composition of multiple components together to create pages.

These are the projects that inspired me during the design process:

- Polaris Components by Shopify
- Vitamin - Web UI kit by Decathlon
- UI Kit by Strapi
- Material 3 Design Kit by Google
- Design System by Vaadin

## 2.4 Design Tokens

Design tokens are design decisions that create the visual style of the design system. They use semantic names to store visual design attributes instead of hard-coded values. It establishes a clear relationship between style choices and their correct usage. Using the same design tokens creates consistency between design and implementation, even when the color value assigned to a token changes. Style updates will propagate through an entire product or the whole organization.

Design language customization or custom theme creation could be achieved using design tokens. Users can set their preferences in the operating system or web browser, and the design system should reflect them. For example, a larger font size or a higher contrast ratio could help people with low vision. My design system uses `rem` units, which are relative to the default font size in the browser.

Tokens are platform-agnostic, but they are typically used in platform-specific code. In my component library, they are represented as CSS custom properties. Based on my research [37], I decided to use three different types of design tokens:

**Global tokens** – Global tokens are context-agnostic and hold raw values, like hex colors, sizes in pixels, or font weights. They define the design language of the design system, like color, typography, spacing, shadow, or shape. These tokens do not change based on the context.

**Alias tokens** – There are also alias tokens that point to global tokens. They add semantic meaning to the static values easily recognizable by their names. The advantage is their usage doesn't require knowing the token's exact value. They change their values by context, enabling customization and creation of custom themes.

**Component tokens** – Component tokens are alias tokens but represent design attributes of a particular element. Each component token maps to a global token with a concrete value. All component tokens must have a default value because components can be used without including global design tokens.

The design system needs a good naming strategy for design tokens. I created this pattern: **Prefix-Category-Concept-Variant-State**. This is an example of a design token CSS variable: `--dfx-form-border-error-active`.

First, I realized multiple CSS libraries could be used simultaneously (even though this is not ideal, it could happen when integrating a library into an existing project). Therefore, I added a *Prefix* `dfx` to my design token names to avoid overriding existing variables. The second part of the name is a *Category*. It represents a design language category or a specific component. For example, it could be a color, font, size or button, input, or label. The category is followed by *Concept*. The concept is a detailed specification of the target variable. It has a *Variant*, which could be a type or value. Finally, there is an optional *State*, which is self-explanatory.

### 2.4.1 Color

Color is a vital part of any design system. It helps communicate the brand, set a hierarchy, and create a sense of unity. To create a theme, you need a color palette. I could not find a good color palette that would play nicely with my theme and could be easily customized for alternative schemes. The usual problem was that the color contrast of one shade was not equal or similar to other colors. This caused the aesthetics to look bad and complicated the creation of custom themes since there could be contrast issues in accessibility. For that reason, I decided to create a tailored color palette.

To ensure I have the same or similar contrast between different colors, I used OKLCH. It is a perceptually uniform color space designed to imitate human perception of color. It provides a more consistent and intuitive representation of color compared to other color spaces like RGB or HSL. But it is not a perfect solution. There could still be some inconsistencies, especially at extreme chroma levels. It uses these parameters to define a color:

- L represents perceived lightness (0% to 100%).
- C represents chroma or saturation (0 to theoretically unbounded, but practically around 0.5).
- H represents the hue angle (0° to 360°).
- A is optional and represents alpha or opacity (0% to 100%). [38]

The color palette of my design system includes four colors, each having eleven shades. Every color palette needs a neutral color. It consists of white, black, and gray shades in between. These colors are used for text, backgrounds, layouts, and all other places across the design system. Neutral colors



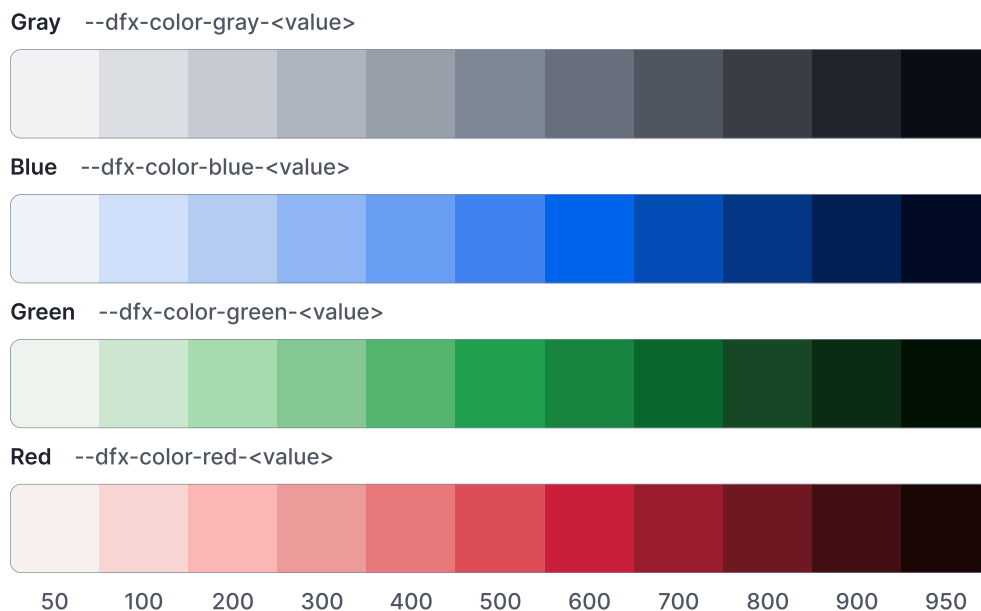
form a visual hierarchy of different surfaces. It helps the user's eye navigate to the essential areas on the screen.

The primary color is critical for representing brand identity. I chose a vibrant shade of blue because it attracts the human eye and aligns with the rest of the palette. It is used for primary actions or to mark a selected item. Semantic colors are used to provide feedback, status, or urgency. They should not be used for decoration. They are based on real-world associations: green for positive feedback and red for danger.

## Primitives

The design palette is built on a foundation of color primitives. These are the core colors that are used throughout the system. Primitives should only be used for theme definition referenced by aliases and not directly in components because of their context-agnostic approach. Figure 2.1 shows the color palette with primitive color tokens.

Figure 2.1: Design tokens: Primitive color palette



### Aliases

Aliases are used to provide semantic meaning to colors. They are used to communicate the intent of the color rather than the specific color itself. Aliases should be used in components instead of primitives. When creating a custom theme, aliases should be modified by changing their reference primitive. Each of these aliases has four variants. The **dark** and **light** variants represent the primary color values for different elements based on context. On the other hand, the **dark-variant** and **light-variant** are alternative shades that provide additional flexibility and contrast within each theme.

Figure 2.2: Design tokens: Color aliases



The tokens *Background*, *Popover*, and *Border* are used for defining the visual aspects of surfaces and layouts. The *Text* token is explicitly used for typography, ensuring that the text elements are legible and aesthetically aligned with the overall design. The *Contrast* token is used on surfaces where the contrast between the foreground and background would be low, typically light text on a dark background in a light theme and dark text on a light background in a dark theme. This helps in maintaining readability and accessibility across different themes. Lastly, the tokens *Neutral*, *Primary*, *Success*, and *Danger* carry semantic meanings, representing different states or actions in the interface. For instance, *Primary* could be used for main actions, *Success* for positive outcomes, and *Danger* for alerts or errors. Alias color tokens are shown in Figure 2.2.

## 2.4.2 Typography

Typography creates visual consistency across different platforms and products. It involves carefully selecting typefaces, font sizes, font weights, and line heights that align with the brand's identity. A well-defined typographic hierarchy in a design system can assist users in navigating through the content, making it more accessible and easier to understand.

### Font family

The selected primary typeface for this design system is Inter, a versatile and modern sans-serif typeface designed specifically for digital screens. Its wide range of weights and styles make it adaptable to various design contexts, contributing to a cohesive and harmonious visual language. However, the design system is flexible and allows using a different font when necessary by adjusting the design token value. Additionally, my component library supports the use of the system font stack, which represents *San Francisco* on Apple devices, *Segoe UI* on Windows, *Roboto* on Android, and *Arial* on other platforms. It can help ensure optimal performance and native aesthetics across different operating systems and devices.

Table 2.1: Design tokens: Font family

Variable	Value
<code>--dfx-font-family</code>	Inter, -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Arial, sans-serif

### Font size

Font sizes promote harmonious typographic hierarchy by standardizing these sizes into design tokens to enhance readability. This system uses eight variants, represented by intuitive T-shirt sizes, to cover all use cases. These sizes range from 10px to 32px, providing a comprehensive scale that fits various typographic needs. This range ensures that every textual element can be displayed with optimal legibility and aesthetic appeal.

The simplicity of T-shirt sizes is used in some of the design tokens. I use them for sizing, making it comprehensible for technical and non-technical individuals to understand the distinctions while maintaining consistency. I always utilize the abbreviated form (**s**, **m**, **l**), simplifying references for developers. The default size is **m**, serving as the standard or base value.

Table 2.2: Design tokens: Font size

Variable	Value
<code>--dfx-font-size-xs</code>	0.625rem (10px)
<code>--dfx-font-size-s</code>	0.75rem (12px)
<code>--dfx-font-size-m</code>	0.875rem (14px)
<code>--dfx-font-size-l</code>	1rem (16px)
<code>--dfx-font-size-xl</code>	1.25rem (20px)
<code>--dfx-font-size-2xl</code>	1.5rem (24px)
<code>--dfx-font-size-3xl</code>	1.75rem (28px)
<code>--dfx-font-size-4xl</code>	2rem (32px)

### Font weight

Font weight design tokens manage the visual weight of typography. Four different weights are used, ranging from 400 to 700. These weights align with the standard font-weight naming convention, which includes **regular** for 400, **medium** for 500, **semibold** for 600, and **bold** for 700.

Table 2.3: Design tokens: Font weight

Variable	Value
<code>--dfx-font-weight-regular</code>	400
<code>--dfx-font-weight-medium</code>	500
<code>--dfx-font-weight-semibold</code>	600
<code>--dfx-font-weight-bold</code>	700

## Line height

Line height design tokens improve readability and the overall aesthetic of the text. For simplicity, I use three line height variants, each represented by a T-shirt size. The **s** variant is used for headlines, providing a compact presentation. The **m** variant is used for regular paragraphs, offering a balanced line spacing. Lastly, the **l** variant is used for introduction paragraphs, providing a more open and inviting layout that draws the reader in.

Table 2.4: Design tokens: Line height

Variable	Value
<code>--dfx-line-height-s</code>	1.25
<code>--dfx-line-height-m</code>	1.5
<code>--dfx-line-height-l</code>	1.75

### 2.4.3 Other styles

#### Border radius

Border radius design tokens define the radius of element corners. Tokens are represented by three T-shirt sizes: **s**, **m**, and **l**, each corresponding to a specific corner radius, from subtle curvatures to more pronounced ones. Additionally, a unique token for a pill or circle shape provides a fully rounded border, ideal for elements like buttons or tags.

Table 2.5: Design tokens: Border radius

Variable	Value
<code>--dfx-border-radius-s</code>	0.25rem (4px)
<code>--dfx-border-radius-m</code>	0.375rem (6px)
<code>--dfx-border-radius-l</code>	0.5rem (8px)
<code>--dfx-border-radius-full</code>	62.438rem (999px)

#### Box shadow

Box shadow represents the shadow effects of elements to create depth and visual interest. Three T-shirt sizes represent design tokens, and each size corresponds to a specific shadow effect, with **s** providing a subtle shadow, **m** offering a moderate shadow, and **l** creating a more pronounced shadow.

Table 2.6: Design tokens: Box shadow

Variable	Value
<code>--dfx-box-shadow-s</code>	0 0.125rem 0.5rem 0 rgba(0, 0, 0, 0.1)
<code>--dfx-box-shadow-m</code>	0 0.25rem 1rem 0 rgba(0, 0, 0, 0.1)
<code>--dfx-box-shadow-l</code>	0 0.5rem 2rem 0 rgba(0, 0, 0, 0.1)

## Spacing

Spacing ensures consistent space between elements across the interface. These tokens can be used for various purposes, including margins, paddings, gaps, density, or other sizes. I defined 13 different sizes, ranging from 1px to 64px, to cover all use cases in components. The sizes are represented by intuitive T-shirt sizes, making it easy for designers to select the appropriate spacing while maintaining consistency across the design.

Table 2.7: Design tokens: Spacing

Variable	Value
<code>--dfx-size-5xs</code>	0.0625rem (1px)
<code>--dfx-size-4xs</code>	0.125rem (2px)
<code>--dfx-size-3xs</code>	0.25rem (4px)
<code>--dfx-size-2xs</code>	0.5rem (8px)
<code>--dfx-size-xs</code>	0.75rem (12px)
<code>--dfx-size-s</code>	0.875rem (14px)
<code>--dfx-size-m</code>	1rem (16px)
<code>--dfx-size-l</code>	1.25rem (20px)
<code>--dfx-size-xl</code>	1.5rem (24px)
<code>--dfx-size-2xl</code>	2rem (32px)
<code>--dfx-size-3xl</code>	2.5rem (40px)
<code>--dfx-size-4xl</code>	3rem (48px)
<code>--dfx-size-5xl</code>	4rem (64px)

## Transitions

Transition design tokens consistently approach animations, motion, and interactivity. This system uses three types of transitions, each categorized by speed: fast, medium, and slow. These speed categories allow designers to create responsive, deliberate, and gradual animations. Each of these tokens uses an `ease-in-out` easing function. It means the animation starts slowly,

speeds up in the middle, and then slows down towards the end, which provides a smooth and natural-feeling transition.

Table 2.8: Design tokens: Transitions

Variable	Value
<code>--dfx-transition-fast</code>	0.1s ease-in-out
<code>--dfx-transition-medium</code>	0.2s ease-in-out
<code>--dfx-transition-slow</code>	0.4s ease-in-out

## Z-index

Z-index elevation design tokens are used for layering elements on the Z-axis. This system uses five z-index types, each assigned a value depending on their context. Overlays start with the first and lowest value, creating a level of depth that allows them to cover the underlying content. Dialogs appear above most other elements, including overlays. They could include popovers, which should be displayed on top. Notifications are visible over other elements to draw user attention immediately. Lastly, tooltips have the highest value, ensuring they show on top of everything else. This approach to z-index elevation ensures a logical stacking order that enhances usability and visual hierarchy without manual z-index manipulation.

Table 2.9: Design tokens: Z-index

Variable	Value
<code>--dfx-z-index-overlay</code>	100
<code>--dfx-z-index-dialog</code>	200
<code>--dfx-z-index-popover</code>	300
<code>--dfx-z-index-notification</code>	400
<code>--dfx-z-index-tooltip</code>	500

## Forms

Form design tokens provide a consistent approach to styling form controls. The opacity of disabled form control communicates its inactive state to users. The outline offset of focused form control adjusts the space between the outline and the border of control. Then, there are tokens for setting the box shadow of form controls. The first is for the default state when there is no user

interaction. Active state highlights the current field and guides the user's attention. Lastly, the error state provides a clear visual cue to users that their input has an issue.

Table 2.10: Design tokens: Forms

Variable	Value
<code>--dfx-form-disabled-opacity</code>	0.5
<code>--dfx-form-outline-offset</code>	0.2rem
<code>--dfx-form-border</code>	inset 0 0 0 0.05rem var(--dfx-color-border-dark)
<code>--dfx-form-border-active</code>	inset 0 0 0 0.1rem var(--dfx-color-contrast-background)
<code>--dfx-form-border-error</code>	inset 0 0 0 0.05rem var(--dfx-color-danger-dark)
<code>--dfx-form-border-error-active</code>	inset 0 0 0 0.1rem var(--dfx-color-danger-dark)

## 2.5 Components

Components are the building blocks that construct the user interface. This section dives into the details and characteristics of components within this design system. There are two types of components: atomic and composed. Atomic components are the most basic UI elements that cannot be broken down further. Composed components, on the other hand, are complex UI elements containing multiple atomic components.

All components have a semantic meaning and can be used independently. Together, they create a composable UI, allowing for a wide range of UI designs with a consistent look and feel. A key feature of components is their adaptability. Each element can have variants, themes, or states, allowing for various visual and functional adaptations to different contexts and user interactions. The following section will discuss the anatomy, properties, layout, and behavior of each component.

In the design system, a total of 30 components have been precisely designed, each serving a unique purpose. Ten components have been fully implemented in code and are ready to be used in real-world applications. A notable achievement is the implementation of core form components, which together create a powerful and flexible form system. More components will be implemented in future development.



### 2.5.1 Alert

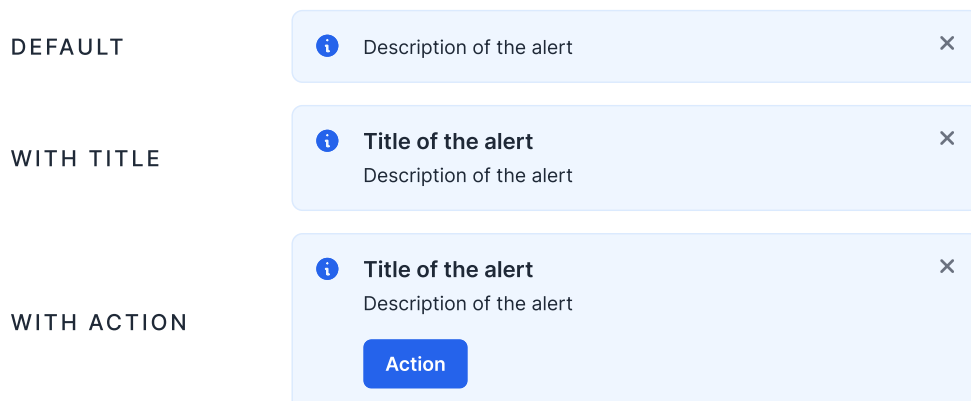
The Alert component includes three essential parts: an icon, content, and a close button. The icon visually represents the type of alert, the content provides the alert message, and the close button allows users to dismiss the alert. This component depends on the Button component, which is used as a close button. The layout of the Alert component is responsive and horizontally stacked. The inner content, however, is vertically stacked. Several attributes characterize the Alert component:

- `closable` – Determines whether the alert can be dismissed.
- `hidden` – Controls the visibility of component.
- `theme` – Sets theme as primary, neutral, success, or danger.
- `hide-icon` – Controls the visibility of the icon.

The component also includes a default slot for content projection, allowing custom content to be inserted into the alert. Additionally, if desired, there is an `icon` slot for inserting a custom icon.

If the alert is closable, a close button is rendered. The close button is keyboard-focusable, so users can navigate to it using the keyboard or narrator. When clicked, this button triggers an event that closes the alert. The content of the alert could also contain a button that triggers a custom action, providing additional interactivity.

Figure 2.3: Components: Alert























## 2.5.2 Button

The Button component consists of horizontally stacked content, which could be a text, an icon, or a combination of both. The only dependency is the Spinner component, which is used when the button is in a loading state. There are three slots: default for the main content in the middle, **start** for content at the beginning, and **end** for content at the end. The button component has these properties:

- **variant** – Sets variant as filled, tonal, outlined, or text.
- **theme** – Sets theme as primary, neutral, success, or danger.
- **label** – Is used for accessibility when the content is only an icon.
- **disabled** – Controls state which turns off interaction.
- **loading** – Controls state that turns off interaction and shows a spinner.
- **loading-text** – Provides the displayed text during the loading state.

The button fires an event when clicked, providing interactive feedback to the user. The button has multiple states: default (with no interaction), hover (when the mouse hovers over it), active (when clicked or pressed), loading (when the action is being processed), and disabled (when it is not possible to click the component). If the button is not in a loading or disabled state, it must be keyboard-focusable.

Figure 2.4: Components: Button

	FILLED	TONAL	OUTLINED	TEXT
DEFAULT				
HOVER				
ACTIVE				
LOADING				
DISABLED				

### 2.5.3 Checkbox, Switch, and Radio

Checkbox, Switch, and Radio have very similar anatomy. The component comprises three main parts: the text content, the input control, and a helper or error label. The text content provides the label for the checkbox through the default slot, the input control is the actual checkbox/switch/radio that users interact with, and the helper or error label provides additional information or feedback to the user. The only dependency is the Label component. It uses a horizontal layout for the text content and input control, but the helper or error label is positioned under the main text content. These properties characterize the component:

- `checked` – Determines whether the control is checked or unchecked.
- `disabled` – Controls whether the control is interactive or not.
- `helper-text` – Provides additional information.
- `required` (only for Checkbox and Switch) – Sets control as required.

Checkbox and Switch have the same behavior. When the input control is clicked, it fires a change event and toggles the state between checked and unchecked, providing immediate visual feedback to the user. With Radio, the click does not toggle the checked value. It stays checked until another radio control is selected. When the component is disabled, the control cannot be clicked or focused by the keyboard, ensuring that the user cannot interact with it. When the control is marked as required, it shows an error when validated.

Figure 2.5: Components: Checkbox, Switch, and Radio

	CHECKBOX	SWITCH	RADIO
UNCHECKED	<input type="checkbox"/> Label Additional label	<input type="checkbox"/> Label Additional label	<input type="radio"/> Label Additional label
CHECKED	<input checked="" type="checkbox"/> Label Additional label	<input checked="" type="checkbox"/> Label Additional label	<input checked="" type="radio"/> Label Additional label
DISABLED	<input checked="" type="checkbox"/> Label Additional label	<input checked="" type="checkbox"/> Label Additional label	<input checked="" type="radio"/> Label Additional label

### 2.5.4 Choice Group

Choice Group is created by two main parts: the label and a set of checkbox/switch/radio controls. The label describes the group, and controls are provided through the default slot. The only dependency is the Checkbox, Switch, or Radio component. The component uses a vertical layout, arranging the controls in a column. It has the following properties:

- `label` – Provides a descriptive label for the group.
- `disabled` – Indicates whether the controls in the group are disabled.
- `helper-text` – Provides additional information.
- `required` – Indicates whether at least one control must be selected.

When the control is marked as disabled, none of the inner controls can be selected. Checkbox and Switch controls can be navigated using the Tab key while navigating through Radio controls using arrow keys. This is the standard behavior of the web platform.

Figure 2.6: Components: Choice Group

	CHECKBOX	SWITCH	RADIO
	Group label	Group label	Group label
DEFAULT	<input checked="" type="checkbox"/> Label	<input checked="" type="checkbox"/> Label	<input checked="" type="radio"/> Label
	<input type="checkbox"/> Label	<input type="checkbox"/> Label	<input type="radio"/> Label
	<input type="checkbox"/> Label	<input type="checkbox"/> Label	<input type="radio"/> Label

### 2.5.5 Input and Textarea

The Input and Textarea components share common parts: a label describes the control, the control is the actual input field, and the helper or error text provides additional information or feedback. The Input component can also have inner buttons or a loading indicator. Both components depend on the Label component, while the Input component also depends on the Button and Spinner components. Both components use a vertical layout, but the Input control uses horizontal stacking of inner elements, which could be inserted through `start` and `end` slots. The properties of the components include:

- `label` – Provides a descriptive label for the control.
- `disabled` – Indicates whether the control is interactive or not.
- `read-only` – Indicates whether the control is writable or read-only.
- `placeholder` – Provides a placeholder, which disappears on focus.
- `helper-text` – Provides additional information.
- `hide-label` – Controls the visibility of the label.
- `type` (only for Input) – Sets input type like text, password or number.

When the control is disabled, it does not allow any input and cannot be focused by the keyboard. Writing single characters fires an input event while changing the value, and losing focus of the field fires a change event. Both components have validation rules like `required`, `minlength`, or `maxlength`, which show the error label when the field is invalid.

Figure 2.7: Components: Input and Textarea

	INPUT	TEXTAREA
DEFAULT	<p>Label</p> <input type="text" value="Text"/>	<p>Label</p> <div style="border: 1px solid #ccc; padding: 5px; min-height: 40px;">Text</div>
ACTIVE	<p>Label</p> <input style="border: 2px solid #000;" type="text" value="Text"/>	<p>Label</p> <div style="border: 2px solid #000; padding: 5px; min-height: 40px;">Text</div>
ERROR	<p>Label</p> <div style="border: 2px solid #f00; padding: 5px; min-height: 20px;">Text</div> <p style="color: #f00; font-size: small;">Error label</p>	<p>Label</p> <div style="border: 2px solid #f00; padding: 5px; min-height: 40px;">Text</div> <p style="color: #f00; font-size: small;">Error label</p>
DISABLED	<p>Label</p> <input style="background-color: #eee; border: 1px solid #ccc;" type="text" value="Text"/>	<p>Label</p> <div style="background-color: #eee; border: 1px solid #ccc; padding: 5px; min-height: 40px;">Text</div>

### 2.5.6 Label

The Label component has no dependencies, making it a simple atomic component. The central part of the component is the text, which is inserted via the default slot. The Label component has a single attribute `type`, which determines the role of the label. The value `control` provides a descriptive label for the control, `error` provides an error message, and `helper` includes additional information for the form control.







Figure 2.8: Components: Label

	CONTROL	HELPER	ERROR
DEFAULT	Label	Helper label	Error label

### 2.5.7 Spinner

The Spinner component is an atomic component, meaning it has no dependencies. The central part of the component is an SVG image with an animation that symbolizes that a process is ongoing. It has two attributes: `size` can be set to small, medium, or large, allowing the spinner to adapt to different contexts and layouts, and `label` sets the text for the spinner, providing additional information to the user about the ongoing process.

Figure 2.9: Components: Spinner

	SMALL	MEDIUM	LARGE
DEFAULT			
WITH LABEL	 Loading...	 Loading...	 Loading...

---

# Implementation

This chapter dives into the intricate details of the project's implementation. It begins with the Project Structure, which provides a comprehensive overview of the project's architecture, including the purpose and relation of various folders and files. It also describes individual files which are used together to create a component.

The subsequent section, Build Process, clarifies the differences between building an application and a library. It highlights the benefits of using TypeScript and the importance of providing correct typings for consumers. This section also outlines the workflow employed to build the library package.

The Documentation section expresses the three primary objectives of the documentation. It introduces the tools for generating component documentation, namely Storybook and Custom Elements Manifest. It describes the structure and content of the documentation, including the specifics of each component documentation page.

The Linting and Formatting distinguishes between these two procedures and discusses the tools used, ESLint and Prettier. It emphasizes their integral role in the development process.

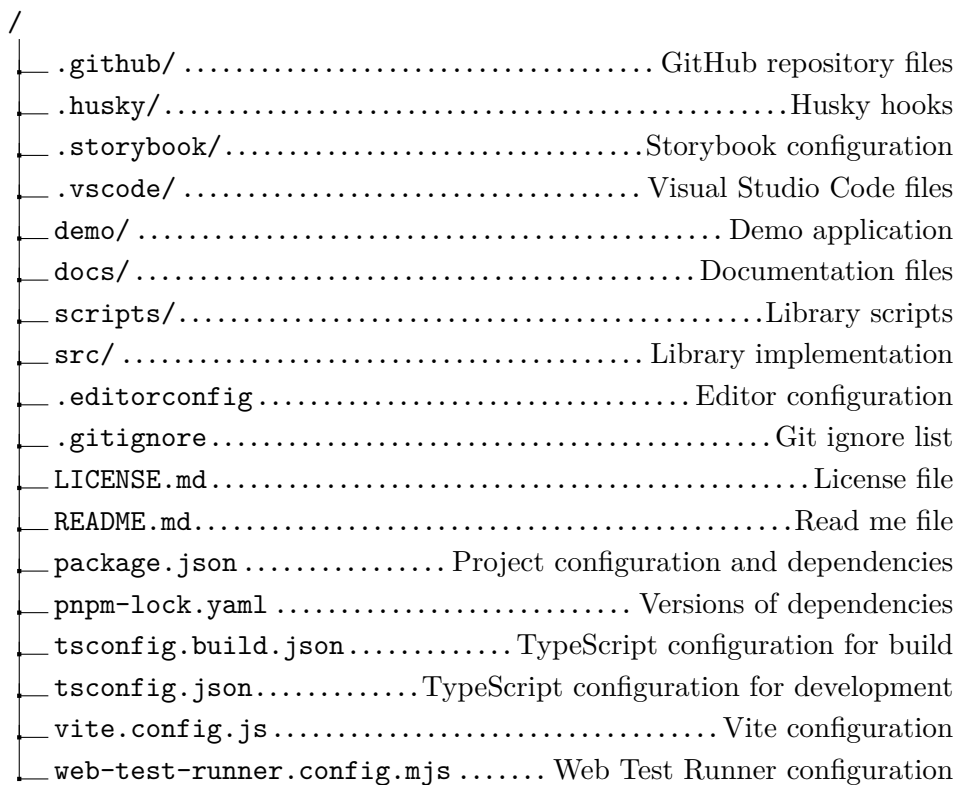
The Forms and Validation section explains how constraint validation is implemented with custom elements. It details the technologies and APIs that associate components with forms, apply validation rules, and describe the custom form validators.

The final section focuses on accessibility in web components. It discusses semantic markup, keyboard handling, and color contrast across the library. The overarching goal is to enhance accessibility for all users, demonstrating a commitment to inclusivity in web development.

### 3.1 Project Structure

Figure 3.1 shows the project structure, which includes several directories, each with a specific purpose. The GitHub directory contains GitHub-specific files like issue and pull request templates, Actions workflows, and community files, including code of conduct, contributing guide, or security policy. The Husky folder contains `pre-commit` and `commit-msg` git hooks. The Storybook directory includes `main`, `manager`, and `preview` configurations with an assets folder. The VS Code folder has project recommendations for extensions and settings. Inside the demo folder, there is a simple application demonstrating component usage. The documentation folder consists of markdown files, stories, and assets. The scripts folder contains a single script for generating new components. The source folder includes an entry point for all components, individual component implementations, themes (default and dark), utilities (events and validators), and assets (SVG icons).

Figure 3.1: Project file structure





In addition to these directories, there are several important files. Coding styles for different IDEs are defined in `.editorconfig`, and `.gitignore` lists files for Git to ignore. The project’s license and overview are in `LICENSE.md` and `README.md`, respectively. `package.json` contains project metadata and dependencies, and `pnpm-lock.yaml` specifies exact dependency versions. TypeScript configurations are in `tsconfig.build.json` and `tsconfig.json`, including one for the library build process and one for development. In `vite.config.js`, there are Vite configurations for library build. Web Test Runner configuration is located in `web-test-runner.config.mjs` file.

The file structure for each component in this project, illustrated by Figure 3.2, is designed to support a modular and maintainable codebase. It consists of five essential files. The word `component` is replaced by the actual component tag name. `components.css` contains the CSS styles that define the look and feel of the component. File `component.stories.ts` is used for Storybook documentation, providing a visual testing playground for the component and containing stories for different component states. `component.test.ts` includes the Web Test Runner tests to ensure the component’s functionality. The actual implementation of the Lit component is found in `component.ts`. Lastly, `index.ts` serves as the component’s entry point for imports. It also includes custom element definitions and TypeScript typings.

Figure 3.2: Component file structure



## 3.2 Build Process

First, let’s have a look at what involves building a JavaScript application. It usually requires using a bundler, which is a crucial tool for optimizing the application for production use. A bundler’s primary job is to compile JavaScript or TypeScript files from a format that is easy to work with during development to an efficient format for browsers to interpret and run. This

could involve transforming modern JavaScript code into a version that can be understood by a wide range of browsers (a process known as transpilation) and combining multiple JavaScript files into one or more bundles (a process known as bundling). [39]

Webpack, Rollup, and Vite are popular choices for this task. They take in various resources like JavaScript, CSS, and HTML files, and transform them into an optimal format for the end-user. This includes minification, which removes unnecessary characters (like spaces and comments) from the source code to reduce its size and load time. In addition, bundlers also perform tree shaking [40]. It is a process of eliminating unused code from the final bundle. This results in a smaller, more efficient bundle that loads faster and consumes less memory.

Another important feature of these tools is code splitting [41]. This is splitting the application's code into multiple chunks that can be loaded on demand. For instance, code for a rarely visited page can be split into a separate chunk from the main bundle, and only loaded when the user navigates to that page. This can improve the initial loading speed of the application, as the browser only needs to download the minimal amount of code necessary to display the initial view to the user.

#### 3.2.1 Building a Library

Building a JavaScript library in the modern web ecosystem involves a different approach than building an application. The focus is creating reusable code easily integrated into other projects. Over the past years, browsers have improved significantly. All modern browsers now support standard ES modules, which include import and export statements. This is an excellent fit for web components, as it allows developers to leverage technology already supported by the browser, reducing the need for additional tools or libraries. [42]

Open WC, an organization that provides resources for web component development, suggests in their publishing guidelines to publish only the latest browser-compatible JavaScript (standard ECMAScript) [18]. This ensures the library can be used in many modern environments without requiring transpilation. One of the key concepts in modern JavaScript library development is the idea of going "buildless". This means the files do not need a build process and can work out of the box in the browser, simplifying the development process. [43]

It is important to note that tasks such as bundling, minifying, and optimizing the code should be left to the developers consuming the library. These are considered application-level concerns, and by leaving these tasks to the consumer, the library can remain lightweight and flexible, allowing developers to optimize their applications according to their specific needs. [18]

### 3.2.2 Using TypeScript

In recent years, TypeScript has significantly grown in popularity among developers. TypeScript shares the syntax of JavaScript and adds typing rules on top of it. It provides a runtime with compile-time type checking. This means that TypeScript checks the types of variables, function parameters, and return values during the compilation process rather than at runtime. TypeScript runtime also preserves the behavior of the JavaScript runtime, meaning JavaScript code will run in TypeScript in the same way. Once TypeScript has checked the types' validity, it removes the types and compiles the code into plain JavaScript. This enables developers to produce more robust and maintainable code by detecting errors earlier in development. [44]

When developing a JavaScript component library, TypeScript can provide several benefits. Providing TypeScript typings offers solid guarantees for component consumption, helping prevent bugs and improve code understanding. It can offer type information for properties, methods, and events, which can be beneficial whether the consumer application uses TypeScript or not. This type information can also enable code auto-completion in modern IDEs such as VS Code, making the development process more efficient. [45]

### 3.2.3 Build Workflow

My component library's build process is a well-thought-out procedure that leverages modern tools and practices for optimal results. Initially, I wanted to use buildless approach since I aim for the latest browsers only, but the build workflow for my library is a bit more complicated, and using bundler makes it much easier and straightforward. I use Vite as a bundler for the development server and build for several reasons:

**TypeScript transpilation** – While this could be done by *tsc* (TypeScript's compiler), Vite uses *esbuild* under the hood, which is significantly faster.

**Assets import** – CSS and SVG files are imported directly into TypeScript as text, and then methods `unsafeCSS` and `unsafeSVG` from Lit transform

them into objects. As there is currently no native way to do this in the browser, Vite is used for this purpose. It allows me to have CSS styles and SVG assets in separate files.

**Copying static styles** – My library includes default and dark themes, which include design tokens. They are represented by static CSS styles, which must be copied separately into the build folder. I might use SCSS in the future, which requires a bundler anyway.

After Vite builds the library, Custom Elements Manifest and TypeScript typings are added. The directory with build artifacts then gets packaged up and published to a software registry. It is an excellent example of a modern, efficient, and effective build process for a JavaScript component library.

## 3.3 Documentation

Creating documentation for a component library is essential to the development process. Good documentation guides users and developers, providing clear information about using the library's components and what to expect from them. It also helps maintain and expand the library in the future [46]. I set 3 main goals for my documentation:

1. The primary goal is to describe the components and their states. This includes clearly explaining what each component does, its properties, methods, and events, and the different states it can be in. This information helps users understand how to use the components and what to expect from them.
2. Another important goal is to include examples and use cases. Practical examples showing how to use the components in real-world scenarios can benefit users. These examples cover common use cases and demonstrate handling different component states. Including code snippets and screenshots can make these examples even more helpful.
3. Finally, it is essential to keep the documentation up to date with the Figma design. The corresponding Figma file is the source of truth for the design, and the documentation must reflect any changes made there. This ensures consistency between the design and the actual components, making it easier for users to understand how the components should look and behave.

### 3.3.1 Storybook and Custom Elements Manifest

Storybook is a popular tool for documenting and developing UI components. It provides an isolated development environment to build and test components, making developing and maintaining complex user interfaces easier. One of the key features is stories, which represent visual test cases that showcase various states and variations of a component. Storybook isolates components for independent development and testing and provides auto-generating documentation from stories. It also has a rich ecosystem of plugins, allowing for easy extension and customization of its functionality. [47]

The Custom Elements Manifest is a file format describing a project's custom elements [48]. It provides rich information to tooling and IDEs about the custom elements inside a project. It generates a `custom-elements.json` file containing metadata about the custom elements in a project: their properties, methods, attributes, events, slots, CSS shadow parts, and CSS custom properties. Storybook can integrate Custom Elements Manifest to auto-generate API references of components, which could be used to control elements in the live preview. It helps in keeping the documentation of the project up-to-date.

### 3.3.2 Content Structure

The documentation primarily serves as a code reference for developers using the library. It provides detailed information about the code structure and usage of the components in the library. The documentation includes the following pages:

**Getting started** – This section starts with a brief introduction about the library, its purpose, benefits, and what it offers. It also includes instructions on installing and using the library in a project.

**Design tokens** – This section describes the design tokens used in the library. Every token is represented by its variable name, value, description, and example, if available.

**Customization** – This section provides information on how to customize the components in the library. This includes details on how to use default and dark themes, create a custom theme, and modify the appearance of components via CSS custom properties and CSS shadow parts.

**Contributing** – This component library is an open-source project, so it is vital to include guidelines for contributing. This contains ways to contribute, contributing workflow, release flow, commit rules, coding conventions, and how to get started locally with the library development.

**Component pages** – Every component in the library has a page that includes a description of the component, API reference, and examples.

Each component is documented with TSDoc, a popular tool for generating API documentation comments in TypeScript. This includes all component attributes, methods, events, slots, CSS properties, and CSS parts. Every property is represented by name, description, type, and default value, if available. The documentation page of the component includes a story (variant) for each attribute, demonstrating how it works and how it should be used. Every story is accompanied by an interactive live preview and its source code.

Listing 3.1: TSDoc annotation example for Custom Elements Manifest

```
/**
 * Alert informs the user about important changes or results
 * of an operation.
 *
 * @element dfx-alert
 * @slot icon - Slot for icon
 * @attr {boolean} closable - Renders close button
 * @fires {Event} dfx-close - Fires when the alert is closed
 * @cssprop [--dfx-alert-padding=var(--dfx-size-m)] - Padding
 * @csspart close-button - Close button element
 */
export class Alert extends LitElement {}
```

## 3.4 Linting and Formatting

Code linting is a static analysis technique to identify problematic code patterns that do not adhere to specific coding guidelines. One of the most popular linting tools in JavaScript is ESLint. It is used to find and fix problems in code, making the code more consistent and avoiding bugs. ESLint does this by analyzing code without executing it. It is highly configurable by using custom rules or extending other coding guides. My project extends ESLint, Web

Components, Lit, Accessibility, Storybook, TypeScript, and Prettier configurations to align with all the tools and technologies I use. Most code editors have plugins for ESLint, which can highlight errors and warnings while writing the code. [49]

On the other hand, code formatting enhances readability and maintainability by ensuring that code follows a consistent style. Prettier is one of the most popular JavaScript code formatters. It removes all original styling and enforces that all outputted code conforms to a consistent code style. While Prettier is opinionated, it is possible to customize its behavior. I set it to avoid function parentheses, set print width to 100 characters, and use single quotes instead of double quotes. Most code editors support Prettier integration to run code formatting whenever a file is saved. [50]

I decided to enforce the project's linting and formatting rules by automating these procedures. A lint-staged utility runs ESLint and Prettier as a pre-commit Husky hook against staged git files and will not let the user commit unless the linter passes. It ensures that all committed code sticks to my defined coding standards. I also run linting and format checking as part of my continuous integration pipeline. This helps catch issues early and keeps the codebase clean.

## 3.5 Forms and Validation

HTML5 forms are the native way in the browser to collect user input. These forms are designed to provide improved data entry controls and validation mechanisms. The `<form>` element contains input elements, such as a text field, checkbox, radio, or submit button. The built-in form validation is achieved by using specific attributes on form elements. For instance, the `type` attribute of input control, such as `email` or `number`, will let the browser check user input and provide feedback if the input is incorrect. There are also validation-related attributes like `required`, `minlength`, `maxlength`, or `pattern`. The form can only be submitted once all the form controls are valid. [51]

This client-side browser validation is using Constraint Validation API [52]. Submitting a form triggers a validation of all form-associated elements. The `checkValidity()` method can check if an input field contains valid data according to its validation rules. If the field is valid, the method returns `true`; otherwise, it returns `false` and fires an `invalid` event. In contrast, the `reportValidity()` method additionally reports any constraint failures

to the user. Setting the `novalidate` attribute on the form element prevents interactive validation of the constraints (validation bubble). Calling the `submit()` method on the form element does not trigger constraint validation, but `requestSubmit()` does. The `validity` property returns a `ValidityState` object, which contains several properties related to the validity of an input element. These properties can be used to check for specific types of validation errors. The Constraint Validation API also includes the `setCustomValidity()` method, which can be used to set a custom validation message.

In my implementation, I utilize the `@open-wc/form-control` package, which provides a `FormControlMixin` base class for my Lit elements [53]. This integration allows my custom elements to be form-associated. It leverages the features of the `ElementInternals` API [54], which is now widely supported across all modern browsers. This package provides several methods. For example, the `setValue()` method sets the form value of the element, the `resetFormControl()` method cleans up the form control's values by setting the initial state, and the `validationTarget` property sets validation on an element inside the shadow DOM.

My library creates two custom validators:

1. `innerInputValidators` utilizes built-in validators for input elements to set the validity based on validity-related attributes like `required`, `minlength`, `maxlength`, and `pattern`.
2. `groupRequiredValidator` is used for choice groups (checkbox or radio), which use the `required` attribute to ensure the user selects at least one of the options

Both of these custom validators provide validation messages that are automatically localized by browsers. My form elements display an error message when they are invalid, and their value has changed, or the form has been submitted. This approach ensures robust client-side validation and a user-friendly form experience.

## 3.6 Accessibility

In HTML, elements can be defined using roles. The beauty of using native elements is that there is no need to define a role, as these elements inherently receive all relevant aria mappings [55]. However, when it comes to custom elements, developers must manually add all relevant accessibility attributes. This



is where my library comes into play. It eliminates this manual labor by automatically applying all relevant accessibility attributes for a specific component in the shadow DOM when my elements are used. I follow the recommended ARIA patterns for creating components [56]. Furthermore, the components in my library provide multiple accessibility-related attributes, which can be utilized to increase accessibility within the context of each application. I focused on the following areas:

**Semantic markup** – The `tabindex` attribute is used with values of either 0 or 1, ensuring a logical and efficient keyboard navigation order. The system refrains from using the `autofocus` and `title` attributes to prevent unexpected focus changes and redundancy in screen reader output. The `<a>` element is strictly used for links, while the `<button>` element is used for buttons, maintaining the integrity of native HTML semantics. Decorative SVG icons are marked with `aria-hidden="true"` to ensure assistive technologies ignore them. All input elements are associated with a label element, enhancing the clarity of the form controls. Additionally, the library provides helper text for further description.

**Keyboard handling** – All controls are designed to be accessible via mouse, keyboard, and touch inputs. All interactive elements have a visible focus style to enhance user interaction. The keyboard focus order matches the visual layout, ensuring a seamless navigation experience, and is designed to follow a “z” pattern, moving from left to right and top to bottom, mimicking the natural reading flow. The system lacks any invisible focusable elements, preventing any unexpected focus shifts. Buttons are made accessible using the space or enter keys, while radio buttons utilize arrow keys for navigation, sticking to standard accessibility practices.

**Color contrast** – The minimum contrast ratio between the standard text and background must be 4.5:1. For larger text, the contrast ratio should be at least 3:1. This rule also applies to interactive and non-textual components such as icons, which must preserve the same contrast ratio of at least 3:1. These measures ensure that the contrast is perceivable by individuals with low visibility or color blindness. It is also important to note that color is not the exclusive way of communication. It is supplemented by text, icons, and other indicators to provide information in many ways, enhancing overall accessibility.

### 3. IMPLEMENTATION

---

My component library is dedicated to delivering accessible components to everyone, regardless of their abilities or the technology they utilize. I am engaged in an ongoing effort to enhance accessibility, ensuring equal access for every user. My goal is to promote accessibility and meet the WCAG 2.2 level AA guidelines, and I persistently strive towards achieving this. My components are designed with the prerequisites for screen reader support, leveraging the support of HTML, ARIA, and WCAG standards. This aids individuals who use screen reader software, further emphasizing my dedication to accessibility. [57][58]

---

# Testing

In this chapter, we dive into the world of testing, a critical aspect of software development that ensures the code's functionality, reliability, and performance. This chapter is divided into the following sections: Strategies and Types, Web Test Runner, and Playwright and Test Cases.

The first section demystifies the difference between manual and automated testing, providing insights into their respective advantages and use cases. It further explores various types of testing, including unit, integration, and end-to-end testing, each with its unique role in the software development lifecycle. The section concludes with an explanation of the component testing strategy, a focused approach to testing web components.

The next section introduces the testing tool used in the project, specifically Web Test Runner. It provides a detailed overview of its features and benefits, including the integration of Playwright headless browsers, which allows for automated testing in a browser environment without a user interface.

The final section outlines the practical application of the testing strategies and tools discussed earlier. It describes the comprehensive testing approach adopted, which includes manual testing, the creation of a testing suite containing 56 total tests, and the generation of code coverage metrics. The tests are run concurrently in three evergreen browsers, ensuring broad compatibility. The section also delves into specific test cases for testing various aspects of web components, including their definition, default values, custom attributes, shadow DOM accessibility, and interactive events.

By the end of this chapter, we will have a solid understanding of the importance of testing, the different strategies and types, the tools that can facilitate the process, and how to create and execute practical test cases.

## 4.1 Strategies and Types

Testing plays a vital role in the process of software development. It involves the evaluation of software applications to detect differences between existing and required conditions (bugs) and to evaluate the features of the software applications. The main goal of testing is to guarantee that all functionalities of a software application are working as expected. [59]

**Manual Testing** – Actual people conduct manual testing. It involves executing test cases without the use of any automation tools. Testers simulate the end-user behavior and validate whether the system behaves as expected. This method is helpful for exploratory, usability, and ad-hoc testing. However, it can be time-consuming, prone to human error, and difficult to manage when dealing with large codebases or complex features.

**Automated Testing** – On the other hand, automated testing is conducted by computer. It involves using software tools to run predefined test cases. The main advantage of automated testing is its speed and consistency. Tests can be run quickly and repeatedly at any time, reducing the time and effort required. It is beneficial for regression testing, load testing, and repetitive tasks. However, writing automated tests requires technical skills and can be costly up-front.

While manual testing allows for human intuition and adaptability, it can be slow and inconsistent due to human error. On the other hand, automated testing is fast and reliable but may need to catch up on unanticipated errors or usability issues that a human tester could catch. Therefore, a balanced approach that utilizes manual and automated testing per the project's requirements and context often delivers the best results. Let me divide test automation into different types for a better understanding:

**Unit Testing** – Unit testing represents the most detailed level of testing, where each code segment is tested individually. Developers should practice unit testing, essentially testing every function they write. The concept is straightforward - for every piece of code we create, we should verify its functionality and existence. Each line of code should be observed to ensure it behaves as expected. As developers sharpen their unit testing skills over time, they better understand the code they write.

**Integration Testing** – Integration testing involves examining the interaction between multiple components. This broader concept builds on unit testing as we continue to test our code and integrate those tests with other components and systems. An example of an integration test could be testing a feature composed of several components and ensuring these components interact harmoniously. Testing this integration is crucial to ensure our application functions as intended.

**End-to-end Testing** – Finally, end-to-end testing involves testing an entire application or feature as an end user would. While this approach may seem impractical and not necessarily essential, it has its merits. It resembles manual testing but with the added benefit of automation. Being able to write robust end-to-end tests implies that the application is straightforward and logical. However, for some applications, the benefits of end-to-end tests may not justify the effort required.

Each testing method has strengths and weaknesses, often combined to achieve a comprehensive testing strategy. Unit testing is excellent for catching minor, isolated issues early, while integration testing helps to catch issues that only arise when different components interact. End-to-end testing, while more time-consuming and complex, helps ensure that the system works together to fulfill its requirements. [60]

Component testing is a specific form of testing that fits into the broader categories of unit, integration, and end-to-end testing. A component could be considered as a "unit" in unit testing. All component properties, attributes, or events are tested separately during component testing. It could also be considered integration testing since most components are composed of other components, so it tests their combined behavior and interaction between them. End-to-end testing has a strong connection with component testing because components are tested in a browser environment, the same environment as real users will use them in production applications. [61]

As mentioned, component testing should ideally be conducted in a browser environment. It provides the most accurate environment for testing the behavior and interaction of components. It is also worth noting that using DOM shimming calls is not recommended for web component testing [62]. While tools like JSDOM can simulate a browser environment, they might not ideally mimic the actual browser behavior, leading to inconsistencies and potential issues that might not be caught during testing. Instead, utilizing a headless browser engine for component testing is possible and often more effective.

This means tests can be run in a browser environment without visibly opening a browser, significantly speeding up the testing process. It could be beneficial for automation, especially for continuous integration pipelines. [63]

### 4.2 Used Tools

Web Test Runner is a modern, lightweight test runner specifically engineered for testing web components and JavaScript projects in a real browser environment. This tool is developed by the Open WC team, which offers recommendations and defaults to aid developers in building web components. It supports headless browsers from Puppeteer, Playwright, Selenium, and WebDriverIO, allowing running tests without a visible UI. One of its notable features is its support for ES modules, which allows components to be tested without the need for prior compilation. Utilizing the latest browser features, it executes tests concurrently, so it can significantly reduce test durations. Web Test Runner is powered by `esbuild` and `rollup` plugins, offering fast build times and a wide range of compatibility with various JavaScript features. [64]

Web Test Runner integrates seamlessly with Playwright, a powerful browser automation library. Playwright is designed to enable reliable and efficient end-to-end testing for modern web applications. One of the standout features of Playwright is its ability to test across multiple browsers. With its integration in Web Test Runner, it can test web components in headless browsers such as Chromium, Firefox, and WebKit. This ensures components function correctly across different browser environments, enhancing the overall compatibility and robustness of the web application. Additionally, it comes with built-in debugging support, making it easy to pause tests and inspect components using the browser's developer tools. [65]

### 4.3 Test Cases

In my approach to manual component testing, I employ a variety of techniques to ensure comprehensive coverage. This includes interacting with components using different inputs such as a mouse, keyboard, and touch. To test accessibility, I utilize a magnification tool to zoom up to 400 % and adjust the browser font size to 200 %. I also make use of assistive technologies, specifically the Narrator in Windows and TalkBack in Android, to ensure the components are accessible to all users. Additionally, I conduct manual visual regression testing to identify any visual deviations from the intended design.

It is crucial to cover all states and scenarios when creating automated component test cases. I created a comprehensive suite of 56 different tests to evaluate each component in the library. These tests are designed to cover a wide range of scenarios and edge cases, ensuring that each component behaves as expected under different conditions. The test setup also computes code coverage, a metric that provides insight into the degree to which the program's source code has been tested. Currently, the code coverage stands at 89.74 %, indicating that the tests have covered a significant portion of the code. However, there is always room for improvement, and efforts will be made to increase this percentage in the future.

One of the standout features of my testing setup is the ability to run all tests concurrently in evergreen browsers: Chromium, Firefox, and WebKit. This cross-browser testing ensures that my components work correctly across different browser environments, enhancing the web application's overall compatibility and user experience.

Test cases of my implementation inspired by [66] cover the following component characteristics: custom element definition, default values, custom attributes, shadow DOM accessibility, and interactive events.

### 4.3.1 Definition

It is crucial to verify whether a custom component is defined correctly. This involves creating an instance of the component and checking if it is an instance of the expected class. For example, consider a Listing 4.1. The test scenario would involve creating a custom element `dfx-checkbox` and asserting that this element is an instance of the `Checkbox` class. This test ensures that the `Checkbox` component is correctly defined and can be instantiated as expected.

Listing 4.1: Test case: Checkbox is defined

```
describe('Checkbox', () => {
  it('is defined', () => {
    const element = document.createElement('dfx-checkbox');

    expect(element).toBeInstanceOf(Checkbox);
  });
});
```

### 4.3.2 Defaults

This testing scenario verifies that a component renders with its default values. This involves creating an instance of the component and checking if its properties match the expected default values. For instance, consider a test where a `dfx-spinner` component is created. The test would then assert that this element's `size` property equals `medium`, which is its default value, and that the `label` property is undefined. This test ensures that the Spinner component is correctly initialized with its default values when rendered. Verifying that a property matches its default value might seem redundant, but it establishes a fundamental baseline before proceeding with further tests.

Listing 4.2: Test case: Spinner renders with default values

```
describe('Spinner', () => {
  it('renders with default values', async () => {
    const element = await fixture<Spinner>(html`
      <dfx-spinner></dfx-spinner>
    `);

    expect(element.size).to.equal('medium');
    expect(element.label).to.be.undefined;
  });
});
```

### 4.3.3 Customization

Testing scenarios often include verifying that a component renders correctly with custom attributes. This involves creating an instance of the component with the custom attributes and checking if its properties match the expected values. For instance, consider a test where a `dfx-label` component is created with a custom attribute `type` set to `error`. The test would then assert that this element's `type` property equals `error`, which is the custom value provided. This test ensures the Label component correctly initializes and renders with the provided custom attributes.



Listing 4.3: Test case: Label renders with custom attributes correctly

```
describe('Label', () => {
  it('renders with custom attributes correctly', async () => {
    const type = 'error';

    const element = await fixture<Label>(html`
      <dfx-label type=${type}></dfx-label>
    `);

    expect(element.type).to.equal(type);
  });
});
```

#### 4.3.4 Accessibility

The final scenario verifies the accessibility of a component's shadow DOM. For instance, consider a test where a `dfx-input` component with a label is created. The test would then perform an accessibility audit on the component's shadow DOM. The expectation is that the shadow DOM of the component passes this audit. The test will fail if I remove the `label` attribute from the component. This kind of test ensures that the `Label` component is accessible.

Listing 4.4: Test case: Input passes the accessibility audit

```
describe('Input', () => {
  it('passes the accessibility audit', async () => {
    const element = await fixture<Input>(html`
      <dfx-input label="Label"></dfx-input>
    `);

    await expect(element).shadowDom.to.be.accessible();
  });
});
```

### 4.3.5 Interactivity

The following scenario verifies that a component fires the correct events when interacted with. For instance, consider a test where a `dfx-button` component is created and a click event is simulated. The test would then listen for the custom `dfx-click` event that should be fired when the button is clicked. The expectation is that this event does indeed occur and that the type of the event detail equals `click`. This test ensures that the Button component correctly emits custom events in response to user interactions.

Listing 4.5: Test case: Button fires *dfx-click* event when clicked

```
describe('Button', () => {
  it('fires dfx-click event when clicked', async () => {
    const element = await fixture<Button>(html`
      <dfx-button></dfx-button>
    `);
    const button = element.shadowRoot?.querySelector('button');

    setTimeout(() => button?.click());
    const event = await oneEvent(element, 'dfx-click', false);

    expect(event).to.exist;
    expect(event.detail.type).to.equal('click');
  });
});
```

---

## Distribution and Usage

This chapter dives into the distribution and usage of the web components UI library developed as part of this diploma thesis. It summarizes the distribution process, components usage, and customization possibilities of the web components UI library. It equips readers with the necessary knowledge to effectively utilize and adapt the library to their requirements.

The Release section outlines the process of preparing the library for distribution. It discusses the conventional commit rules and the tools to enforce them, ensuring a consistent and meaningful commit history. It also explains the principles of semantic versioning and the individual steps of the GitHub Actions release workflow. These steps include installing dependencies and browsers, checking types and formatting, linting, building, testing, and finally, releasing the package to the npm repository. The section also touches upon the concept of release branches and documentation deployment.

The Components Usage section provides a comprehensive guide on installing, importing, and using the library's components. It introduces a demo application designed to showcase all the components in action. Furthermore, it elaborates on the usage and configurations of web components in different web frameworks, including React, Vue, and Angular, providing a versatile understanding of the library's applicability.

The final section, Customization, focuses on the adaptability of the library. It explains how to use the default and dark themes included in the component library and describes the ability to create a custom theme. It also details how to customize individual components using CSS custom properties and CSS shadow parts, allowing users to tailor the components to their specific needs.

## 5.1 Release

A release workflow is a crucial part of the software development process that automates the steps involved in releasing a new version of your software. It often includes building the software, running tests, and deploying it to a production environment. Conventional commits and semantic versioning are vital aspects of a robust release workflow. Together, they can streamline the release workflow and make it easier to automate.

### 5.1.1 Conventional Commits

Conventional Commits is a specification that adds meaning to commit messages [67]. This convention allows for readable messages that are easy to follow when looking through the project history, and it also enables the automatic generation of changelogs and release notes. A conventional commit message follows this format:

Listing 5.1: Conventional Commits format

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

**Type** – This represents the nature of the changes and has to be one of the following: **feat** (feature), **fix** (bug fix), **docs** (documentation), **style** (formatting), **refactor** (code refactoring), **test** (testing), **build** (build process), **ci** (continuous integration), **perf** (performance), **chore** (maintain), and **revert** (commit reverts).

**Scope** – This is optional and should be used when the change affects only a specific component or a part of the project.

**Subject** – The brief description of the changes, written in the imperative, present tense: "change", not "changed", nor "changes", does not capitalize the first letter, and does not have a dot (.) at the end.

**Body and Footer** – These are optional and used for more detailed explanatory text, including indicating breaking changes or specific issue fixes.

Listing 5.2: Example of conventional commit

```
feat(button): add loading state
```

My project uses specific tools to enforce commit rules [68]. Commitlint is a tool that checks if the commit messages meet the rules of Conventional Commits. By setting up Husky with a `pre-commit` hook, it can automatically lint commit messages before they are committed. The commit will be rejected if the message does not follow the Conventional Commits format.

On the other hand, Commitizen is a command-line tool that helps to create commit messages that adhere to the Conventional Commits rules. When you commit using Commitizen, it prompts you to fill out any required commit fields. This ensures that every commit meets the standards set by Conventional Commits.

Adhering to the Conventional Commits format makes versioning more efficient and consistent, making it easier for developers and contributors to follow changes in the project. It also helps to automate the release flow and generate changelog notes.

### 5.1.2 Semantic Versioning

Semantic Versioning, abbreviated as SemVer, is a versioning scheme that aims to provide information about the underlying changes in a release. A semantic version number is composed of three parts `MAJOR.MINOR.PATCH`:

**MAJOR** version increment indicates that there are incompatible changes in the API, and developers may need to modify the code of the application to accommodate these changes.

**MINOR** version increment indicates that new features have been added in a backward-compatible manner. Users can use the new features without altering existing code.

**PATCH** version increment indicates that backward-compatible bug fixes have been introduced. These fixes do not add or change new features but rather fix issues with existing features.

An optional pre-release or build metadata identifiers can be appended to the version. For example, 1.0.0-beta.1 or 1.0.0-next.10. Semantic versioning provides a clear contract to the users of your software, allowing users to comprehend the potential consequences of upgrading to a new version. It is widely adopted in the open-source community and is used by many package managers, including npm. [69]

### 5.1.3 Release Workflow

My component library uses GitHub Actions. It is a powerful tool for implementing a continuous integration (CI) pipeline directly within a GitHub project. It automates software workflows, including building, testing, and deploying applications. Multiple events, such as push or pull requests, can trigger workflows. CI workflows are defined in code using YAML syntax, so they are version-controlled and reviewed as part of pull requests. The GitHub Actions release workflow for my component library consists of the following steps:

1. **Install dependencies** – This step installs the project dependencies. It ensures that the exact versions of dependencies specified in the lockfile are installed.
2. **Install Playwright browsers** – This step installs the browsers needed for the Playwright tool, a Node.js library used for browser automation.
3. **Check types** – This step checks the types in the project, ensuring there are no type errors. This is typically used in TypeScript projects.
4. **Check formatting** – This step checks the formatting of the code, ensuring it adheres to the specified style guide.
5. **Lint** – This step lints the code, checking for syntax errors and enforcing code style rules.
6. **Build** – This step builds the component library, including generating TypeScript types and Custom Elements Manifest.
7. **Test** – This step runs the project’s test suite concurrently in headless browsers, ensuring all tests pass.
8. **Release** – This step runs semantic-release to automate the versioning and package publishing process.

The project uses semantic-release to automate the release flow. It follows semantic versioning rules. When a commit is marked as **BREAKING CHANGE** in the commit message, that push creates a major release in the current channel. Backward incompatible changes should have breaking change commits. When some commits have a **feat** prefix in the message, that push creates a minor release. Finally, when some commits have a **fix** prefix in the message, and there are not any **feat** commits, that push creates a patch release. Every commit triggers a release flow based on the branch:

- **main**: Every push makes a stable release in **latest** npm channel.
- **next**: Every push makes a preview release in **next** npm channel.

With every release, the new version is published to the npm repository. The semantic-release also automatically creates a GitHub Release with auto-generated release notes, including all commits since the last release. I do not directly push changes to the **main** and **next** branches. The **main** branch will only be updated by a PR from the **next** branch that contains features/fixes that we are ready to release as a stable release. All changes for the current version will be submitted in PRs targeting the **next** branch. PRs are not making any releases.

As a last step of my release workflow, Storybook documentation is built and deployed as a static site to Netlify. Custom Elements Manifest is generated before the Storybook is built to provide metadata for all components. Once the build process is complete, the workflow deploys the static Storybook site to Netlify, a popular platform for hosting static websites. This automated process ensures that your Storybook documentation is always up-to-date, providing a valuable resource for developers.

## 5.2 Components Usage

Using components from my library in a JavaScript or TypeScript project involves the following steps:

1. **Installation** – Ensure the latest Node.js is installed on the machine. Use a package manager (such as npm, yarn, or pnpm) to install the component library from npm repository:

```
npm install diffix
```

- 2. Import** – Import the target component you want to use in the project from `diffix/components/component-name`. Import automatically registers the custom element (causes a side effect).

```
import 'diffix/components/button';
```

Or import all components at once:

```
import 'diffix';
```

- 3. Usage** – My components are designed to be used as Web Components. This means you import the target component in the script and then use it in the HTML, JavaScript, or JSX like a standard HTML element:

```
<dfx-button>Button</dfx-button>
```

There is a simple application using Vite in `demo` folder inside the library implementation. It showcases the usage of individual components and implements features like a theme toggle, saving the preferred theme outside the current session, or auto-detecting the user system preference.

### 5.2.1 Usage in Frameworks

Web Components enjoy robust support across all major browsers, enhancing their appeal for modern web development [22]. Their design allows for easy integration into contemporary front-end frameworks, providing developers with a versatile tool for creating reusable, encapsulated components. However, it is essential to note that the level of integration ease can vary, as only some frameworks facilitate a seamless integration process for Web Components. [70]

#### React

In React, you can use Web Components just like any regular HTML element. However, due to the difference in event handling between React and native DOM, you need to use React's `ref` API to attach event listeners to Web Components. Some components provide React wrappers for Web Components so you can use props and handle events just like any other React component. I do not provide these wrappers at the moment. A planned future release of React will improve support for using Web Components, which will make the wrapper components unnecessary. [71]

In an example shown in Listing 5.3, a `dfx-button` component is rendered, and a `ref` is attached to it. Inside the `useEffect` hook, an event listener



Listing 5.3: Component usage in React

```
import React, { useEffect, useRef } from 'react';
import 'diffix/components/button';

function MyComponent() {
  const ref = useRef();

  useEffect(() => {
    const element = ref.current;
    const handleClick = (event) => {
      console.log('dfx-click event fired!', event.detail);
    };

    element.addEventListener('dfx-click', handleClick);

    return () => {
      element.removeEventListener('dfx-click', handleClick);
    };
  }, []);

  return <dfx-button ref={ref}>Button</dfx-button>;
}
```

is added to the button for the `dfx-click` event. When the `dfx-click` event is fired, the `handleClick` function will be called, and the event's details will be logged to the console. The event listener is cleaned up when the component is unmounted to prevent memory leaks. This is a common pattern for using Web Components with custom events in React.

## Vue

Vue provides excellent support for the consumption of custom elements. This is beneficial whether integrating custom elements into an existing Vue application or starting from scratch. The process of using custom elements within a Vue application is largely similar to using native HTML elements, with a few considerations. By default, Vue tries to resolve a non-native HTML tag as a registered Vue component before it resorts to rendering it as a custom element. To ensure Vue treats certain elements as custom elements and

bypasses component resolution, the `compilerOptions.isCustomElement` option should be specified. If Vue is being used with a build setup, this option should be passed via build configurations as it's a compile-time option. [72]

In Listing 5.4, the `dfx-button` component is used in the Vue template. The `@dfx-click` directive is used to listen for the `dfx-click` event. When the `dfx-click` event is fired, the `handleClick` method will be called, and the event's details will be logged into the console.

Listing 5.4: Component usage in Vue

```
<template>
  <dfx-button @dfx-click="handleClick">Button</dfx-button>
</template>

<script>
import 'diffix/components/button';

export default {
  methods: {
    handleClick(event) {
      console.log('dfx-click event fired!', event.detail);
    },
  },
};
</script>
```

## Angular

Angular provides excellent support for Web Components, and its architecture allows seamless integration. This can be achieved using Angular's schema `CUSTOM_ELEMENTS_SCHEMA` to recognize and enable custom elements in Angular templates. Furthermore, Angular's change detection mechanism works well with properties and events of Web Components, allowing for efficient data binding and event handling. [73]

In Listing 5.5, the `dfx-button` component is used in the Angular template. The `(dfx-click)` syntax is used to listen for the `dfx-click` event. When the `dfx-click` event is fired, the `handleClick` method will be called, and the event's details will be logged into the console.

Listing 5.5: Component usage in Angular

```
import { Component } from '@angular/core';
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import 'diffix/components/button';

@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <dfx-button (dfx-click)="handleClick($event)">
      Button
    </dfx-button>
  `,
  schemas: [CUSTOM_ELEMENTS_SCHEMA]
})
export class AppComponent {
  handleClick(event: CustomEvent) {
    console.log('dfx-click event fired!', event.detail);
  }
}
```

## 5.3 Customization

Design System can be customized at a high level through design tokens. This gives the developer control over theme colors and general styling. It is possible to utilize component variables to target individual components for more advanced customizations.

One of the core principles of Web Components is the encapsulation. The HTML and styles are encapsulated in the Shadow DOM. This avoids conflicts with any CSS styles used in the rest of the application but also makes customization harder. However, styling Web Components is easily achievable thanks to CSS Custom Properties, which flow down inside the Shadow DOM and CSS Shadow Parts.

### 5.3.1 Default Theme

Design tokens are accessed through CSS custom properties defined in the `:root` block of theme styles. Because design tokens are global, they're always

prefixed with `--dfx` to avoid collisions with other libraries. To use the default theme, import the CSS file:

- Using an application bundler (Webpack, Rollup, Vite, etc.):

```
import 'diffix/themes/default.css';
```

- Or using a `<link>` tag:

```
<link href="/node_modules/diffix/dist/themes/default.css"
      rel="stylesheet" />
```

### 5.3.2 Dark Theme

The dark theme is a color scheme that uses light color for text, icons, and graphical user interface elements on a dark background. It is a popular feature in many operating systems, applications, and websites. The benefits of using a dark theme include reduced eye strain, especially in low-light conditions, potential energy savings on OLED displays, and it can be easier on the eyes for some users. Additionally, it can provide a visually appealing alternative to traditional light themes, offering a different aesthetic experience. [74]

To use the dark theme, import the CSS file of the default theme (for general design tokens) and the dark theme (changing semantic colors):

- Using an application bundler (Webpack, Rollup, Vite, etc.):

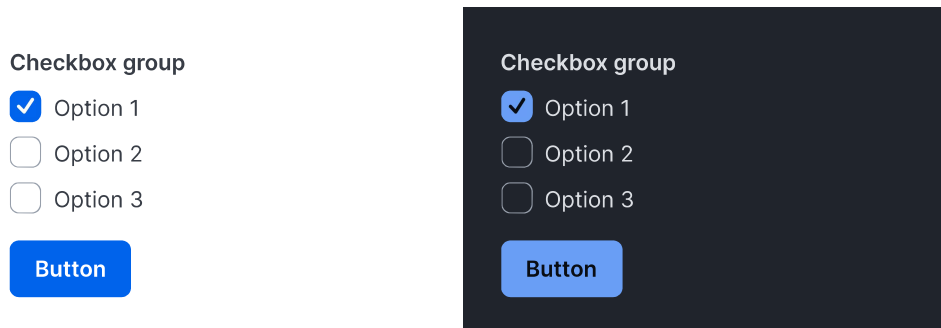
```
import 'diffix/themes/default.css';
import 'diffix/themes/dark.css';
```

- Or using a `<link>` tag:

```
<link href="/node_modules/diffix/dist/themes/default.css"
      rel="stylesheet" />
<link href="/node_modules/diffix/dist/themes/dark.css"
      rel="stylesheet" />
```

The dark theme is not applied automatically. Use the `data-theme` attribute on the `<html>` element to switch between the light and dark themes. Let's have a look at the light and dark mode comparison at Figure 5.1. It is possible to use an attribute on different element than `<html>`, so that the theme is only applied to the inner content.

Figure 5.1: Comparison between light and dark theme



### Custom Theme

The theme is a set of CSS Custom Properties. It is possible to customize and use them in the application with pure CSS, no preprocessor required. There are two similar ways to create a custom theme:

1. Create a custom theme by extending the default theme. Import the CSS file of the default theme and then your custom styles. You can extend the default theme by defining the same CSS custom properties with different values inside your styles.
2. Create a custom theme from scratch. Copy the content of the default theme and change the values of the variables. Then, import the CSS file of the custom theme instead of the default theme.

My design system uses Inter as the default typeface. It is imported into the default theme. However, you don't have to use Inter. You can use any font you prefer by setting the `--dfx-font-family` CSS custom property inside your theme styles. It is recommended to use a font optimized for the web. There are many options:

- Use a widely available font in browsers, such as system fonts (e.g., Arial) or the system font stack. It uses Segoe UI on Windows, San Francisco on macOS and iOS, and Roboto on Android and Chrome OS. If you don't import the default theme, components will fall back to the system font stack. Or you can set it manually in your theme:

```
--dfx-font-family: -apple-system, BlinkMacSystemFont,
'Segoe UI', Roboto, Arial, sans-serif;
```

- You can use the `@font-face` rule to import your locally hosted font. That's the way the default theme imports Inter.
- Use external font services such as Google Fonts. The most effective way is to import the font using the `<link>` tag in the `<head>` of your HTML.

The default theme is automatically applied because it is declared in the `:root` block. If you want to use more themes in your application, you can import them and switch between them using the `data-theme` attribute.

Figure 5.2: Custom theme example

Checkbox group

Option 1

Option 2

Option 3

Button

Listing 5.6: Custom theme usage

```
<article data-theme="custom">
  <dfx-checkbox-group label="Checkbox group" value="1">
    <dfx-checkbox value="1">Option 1</dfx-checkbox>
    <dfx-checkbox value="2">Option 2</dfx-checkbox>
    <dfx-checkbox value="3">Option 3</dfx-checkbox>
  </dfx-checkbox-group>
  <dfx-button variant="filled">Button</dfx-button>
</article>

<style>
  [data-theme='custom'] {
    --dfx-font-family: -apple-system, BlinkMacSystemFont,
      'Segoe UI', Roboto, Arial, sans-serif;
    --dfx-color-neutral-dark: #fbbf24;
    --dfx-color-neutral-dark-variant: #f59e0b;
    --dfx-color-contrast-text: #09090b;
  }
</style>
```

### 5.3.3 Customizing Components

Web Components offer a powerful way to create reusable, encapsulated components. Customizing the styling of these components can be achieved through two primary methods: CSS Custom Properties and CSS Shadow Parts.

#### CSS Custom Properties

CSS Custom Properties allow defining custom properties that can be reused throughout CSS. This is particularly useful in Web Components, as it allows component styles to be easily customized. My components provide variables that exist at the component level. They start with the prefix `--dfx-component-name`. For example, `dfx-button` component exposes a custom property `--dfx-button-border-radius` for customization of a component's border-radius. This allows the border radius to be easily changed without having to rewrite the component's CSS.

Listing 5.7: CSS Custom Properties example

```
dfx-button.rounded {  
  --dfx-button-border-radius: var(--dfx-border-radius-full);  
}
```

#### CSS Shadow Parts

CSS Shadow Parts provide a way to style specific parts of a component's Shadow DOM from outside the component. This is done by exposing parts of the component's internal structure, which can be targeted with CSS shadow part selector `::part()`. For example, a `dfx-button` component exposes a button element as a shadow part, allowing it to be styled directly.

Listing 5.8: CSS Shadow Parts example

```
dfx-button.underline::part(button) {  
  text-decoration: underline;  
}
```





---

# Conclusion

The diploma thesis aimed to create a component library, establish a custom design system, provide comprehensive documentation, and distribute the package to a public repository. These goals were accomplished by creating a performant, customizable, open-source UI library. It provides a set of reusable components that enable the building of web applications with a consistent user experience. It integrates seamlessly with any JavaScript framework, such as React, Vue, or Angular. The project follows the W3C Web Component standards, ensuring a highly performant and accessible web experience. It also allows customizing the design language for individual projects by modifying design tokens.

The thesis started with a comprehensive exploration and analysis of design systems, component libraries, web components technology, and web components frameworks. The motivation, principles, realization, design tokens, and components that formed the foundation of a robust design system were discussed. The implementation phase outlined the project structure, build process, documentation, linting, formatting, forms, and accessibility considerations. Various testing strategies, tools, and specific test cases were explored, emphasizing the importance of testing. Finally, the distribution and usage of the library were discussed, including the release process, how to use the components, and customize them.

Despite the lack of resources and the rapid obsolescence of articles older than about three years, I found a solid solution. I chose the Lit framework for implementation, and the library was written in TypeScript, with Vite used for the build process. Tests run in Web Test Runner utilizing Playwright headless browsers. Each component was documented in the Storybook, linted with

ESLint, and formatted by Prettier. Commits are checked with Commitlint and created with Commitizen. The component library uses semantic-release utility for automatic release workflow and deployment.

The future development and improvements of the component library are promising. The first step is implementing the rest of the designed components, further expanding the library's capabilities. The documentation will receive enhancements, including a section for migration and a roadmap of planned features, providing users with a clear path to the library's evolution. The testing process will also be upgraded, with testing of shadow DOM snapshots and visual regression testing. I consider using a CSS preprocessor, which aims to streamline working with styles, coupled with the establishment of CSS linting rules to maintain code quality. Form controls are set to gain the ability to set custom invalid text. Lastly, I want to experiment with the server-side rendering capabilities of the components.

In conclusion, this diploma thesis has successfully delivered a modern, customizable, and user-friendly web components UI library. The library is well-documented and readily available in a public npm repository, ready to be used out-of-the-box in existing front-end projects. This accomplishment fulfills the goals set out at the beginning of the thesis and contributes a valuable tool to the web development community.

---

# Bibliography

1. Web Components. *W3C Wiki* [online]. 2014 [visited on 2024-01-06]. Available from: <https://www.w3.org/wiki/WebComponents/>.
2. SISK, Kolby. *Design Systems Demystified* [online]. [visited on 2024-01-06]. Available from: <https://www.designsystem.tools/>.
3. *Why design systems matter* [online]. [visited on 2024-01-06]. Available from: <https://fem-design-systems.netlify.app/why-design-systems-matter>.
4. Design Tokens Format Module. *Design Tokens Community Group* [online]. 2023 [visited on 2024-01-06]. Available from: <https://tr.designtokens.org/format/>.
5. *Design tokens – Spectrum* [online]. [visited on 2024-01-06]. Available from: <https://spectrum.adobe.com/page/design-tokens/>.
6. CURTIS, Nathan. UX Patterns UI Components. *Medium* [online]. 2017 [visited on 2024-01-06]. Available from: <https://medium.com/eightshapes-llc/patterns-components-2ce778cbe4e8>.
7. CAPOZZI, Mae. The 5 layers of a design system [online]. 2020 [visited on 2024-01-06]. Available from: <https://maecapozzi.com/blog/layers-of-abstraction-in-design-systems>.
8. LANGVAD, Rasmus. What Is a Component Library [online]. 2020 [visited on 2024-01-08]. Available from: <https://langvad.dev/blog/what-is-a-component-library/>.
9. DYSZKIEWICZ, Joanna. What is a UI component library and when to use it? *Pagepro* [online]. 2022 [visited on 2024-01-08]. Available from: <https://pagepro.co/blog/what-is-a-ui-component-library/>.

10. VANTOLL, TJ. The Ultimate Guide to Building a UI Component Library—Part 1: Creating a Plan. *Telerik* [online]. 2021 [visited on 2024-01-08]. Available from: <https://www.telerik.com/blogs/ultimate-guide-to-building-ui-component-library-part-1-plan>.
11. KHOSRAVI, Kasra. Web components vs. React. *LogRocket* [online]. 2023 [visited on 2024-01-08]. Available from: <https://blog.logrocket.com/web-components-vs-react/>.
12. Web Components. *MDN Web Docs* [online]. 2023 [visited on 2024-01-08]. Available from: [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_components](https://developer.mozilla.org/en-US/docs/Web/API/Web_components).
13. GORE, Anthony. What makes a great component library. *Retool* [online]. 2022 [visited on 2024-01-08]. Available from: <https://retool.com/blog/what-makes-a-great-component-library>.
14. Guide to Building Design Systems With Web Components. *Ionic* [online]. 2022 [visited on 2024-01-08]. Available from: <https://ionic.io/resources/articles/building-design-systems-with-web-components>.
15. FROST, Brad. Let's talk about web components [online]. 2022 [visited on 2024-01-08]. Available from: <https://bradfrost.com/blog/post/lets-talk-about-web-components/>.
16. PORTS, Kevin. 5 Reasons Web Components Are Perfect for Design Systems. *Ionic* [online]. 2019 [visited on 2024-01-08]. Available from: <https://ionic.io/blog/5-reasons-web-components-are-perfect-for-design-systems>.
17. KAROULLA, Andrico. Scaffolding the repo for your UI library [online]. 2021 [visited on 2024-01-18]. Available from: <https://blog.andri.co/001-scaffolding-the-repo-for-your-ui-library/>.
18. *Developing Components: Publishing* [online]. [visited on 2024-01-18]. Available from: <https://fluent2.microsoft.design/design-tokens>.
19. FROST, Brad. Design system versioning: single library or individual components? [online]. 2022 [visited on 2024-01-18]. Available from: <https://bradfrost.com/blog/post/design-system-versioning-single-library-or-individual-components/>.
20. CAPOZZI, Mae. Should you version components separately or as a unified system? [online]. 2020 [visited on 2024-01-18]. Available from: <https://maecapozzi.com/blog/version-bundling>.

21. CRUTCHLEY, Corbin. Web Components 101: History. *Unicorn Utterances* [online]. 2021 [visited on 2024-01-08]. Available from: <https://unicorn-utterances.com/posts/web-components-101-history>.
22. *Can I use web components?* [online]. [visited on 2024-01-08]. Available from: <https://caniuse.com/?search=web%20components>.
23. WHATWG. Custom elements. *HTML Spec* [online]. 2024 [visited on 2024-01-11]. Available from: <https://html.spec.whatwg.org/multipage/custom-elements.html>.
24. WHATWG. Interface ShadowRoot. *DOM Spec* [online]. 2024 [visited on 2024-01-26]. Available from: <https://dom.spec.whatwg.org/#interface-shadowroot>.
25. LAVISKA, Cory. Flash of Undefined Custom Elements (FOUCE). *A Beautiful Site* [online]. 2021 [visited on 2024-01-11]. Available from: <https://www.abeautifulsite.net/posts/flash-of-undefined-custom-elements/>.
26. MILLER, Jason; FREED, Mason. Declarative Shadow DOM. *Chrome for Developers* [online]. 2023 [visited on 2024-01-11]. Available from: <https://developer.chrome.com/docs/css-ui/declarative-shadow-dom>.
27. Using Web Components: Styling Components Using Shadow DOM. *Web Components Community Group at W3C* [online]. 2022 [visited on 2024-01-11]. Available from: <https://web-components-cg.netlify.app/articles/using/styling-shadow/>.
28. WHATWG. The template element. *HTML Spec* [online]. 2024 [visited on 2024-01-11]. Available from: <https://html.spec.whatwg.org/multipage/scripting.html#the-template-element>.
29. CRUTCHLEY, Corbin. Web Components 101: Framework Comparison. *Unicorn Utterances* [online]. 2021 [visited on 2024-01-12]. Available from: <https://unicorn-utterances.com/posts/web-components-101-framework-comparison>.
30. All the Ways to Make a Web Component. *WebComponents.dev* [online]. 2022 [visited on 2024-01-12]. Available from: <https://webcomponents.dev/blog/all-the-ways-to-make-a-web-component/>.
31. *@stencil/core vs lit - npm trends* [online]. [visited on 2024-01-12]. Available from: <https://npmtrends.com/@stencil/core-vs-lit>.

32. Stencil: A Web Components Compiler. *Stencil* [online]. 2024 [visited on 2024-01-12]. Available from: <https://stenciljs.com/docs/introduction>.
33. What is Lit? *Lit* [online]. 2023 [visited on 2024-01-12]. Available from: <https://lit.dev/docs/>.
34. LAVISKA, Cory. Moving from Stencil to LitElement. *A Beautiful Site* [online]. 2021 [visited on 2024-01-12]. Available from: <https://www.abeautifulsite.net/posts/moving-from-stencil-to-lit-element/>.
35. *Design System Software* [online]. [visited on 2024-01-16]. Available from: <https://www.figma.com/design-systems/>.
36. *Adobe XD Learn & Support* [online]. [visited on 2024-01-16]. Available from: <https://helpx.adobe.com/support/xd.html>.
37. *Design tokens* [online]. [visited on 2024-01-16]. Available from: <https://fluent2.microsoft.design/design-tokens>.
38. SITNIK, Andrey; TURNER, Travis. OKLCH in CSS: why we moved from RGB and HSL. *Evil Martians* [online]. 2022 [visited on 2024-01-16]. Available from: <https://evilmartians.com/chronicles/oklch-in-css-why-quit-rgb-hsl>.
39. JavaScript Module Bundlers [online]. 2021 [visited on 2024-01-21]. Available from: <https://byby.dev/js-module-bundlers>.
40. WAGNER, Jeremy. Reduce JavaScript payloads with tree shaking [online]. 2018 [visited on 2024-01-21]. Available from: <https://web.dev/articles/reduce-javascript-payloads-with-tree-shaking>.
41. DJIRDEH, Houssein. Reduce JavaScript payloads with code splitting [online]. 2018 [visited on 2024-01-21]. Available from: <https://web.dev/articles/codelab-code-splitting>.
42. *Going Buildless: ES Modules* [online]. [visited on 2024-01-18]. Available from: <https://modern-web.dev/guides/going-buildless/es-modules/>.
43. KAROULLA, Andrico. How to build (and maybe bundle) your UI library's packages [online]. 2021 [visited on 2024-01-18]. Available from: <https://blog.andri.co/004-how-to-build-and-maybe-bundle-your-ui-library/>.

- 
44. *TypeScript: JavaScript With Syntax For Types* [online]. [visited on 2024-01-21]. Available from: <https://www.typescriptlang.org/>.
  45. ATTARDI, Joe. Understanding TypeScript's benefits and pitfalls [online]. 2022 [visited on 2024-01-21]. Available from: <https://blog.logrocket.com/understanding-typescripts-benefits-pitfalls/>.
  46. CURTIS, Nathan. Documenting Components [online]. 2018 [visited on 2024-01-22]. Available from: <https://medium.com/eightshapes-llc/documenting-components-9fe59b80c015>.
  47. *Why Storybook?* [online]. [visited on 2024-01-22]. Available from: <https://storybook.js.org/docs/get-started/why-storybook>.
  48. *Getting Started: Custom Elements Manifest* [online]. [visited on 2024-01-22]. Available from: <https://custom-elements-manifest.open-wc.org/analyzer/getting-started/>.
  49. *Core Concepts - ESLint* [online]. [visited on 2024-01-22]. Available from: <https://eslint.org/docs/latest/use/core-concepts>.
  50. *Why Prettier?* [online]. [visited on 2024-01-22]. Available from: <https://prettier.io/docs/en/why-prettier>.
  51. *Client-side form validation* [online]. [visited on 2024-01-22]. Available from: [https://developer.mozilla.org/en-US/docs/Learn/Forms/Form\\_validation](https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation).
  52. *Constraint validation* [online]. [visited on 2024-01-22]. Available from: [https://developer.mozilla.org/en-US/docs/Web/HTML/Constraint\\_validation](https://developer.mozilla.org/en-US/docs/Web/HTML/Constraint_validation).
  53. *@open-wc/form-control* [online]. [visited on 2024-01-23]. Available from: <https://github.com/open-wc/form-participation/tree/main/packages/form-control>.
  54. EVANS, Arthur. More capable form controls [online]. 2019 [visited on 2024-01-23]. Available from: <https://web.dev/articles/more-capable-form-controls>.
  55. *WAI-ARIA basics | MDN* [online]. [visited on 2024-01-23]. Available from: [https://developer.mozilla.org/en-US/docs/Learn/Accessibility/WAI-ARIA\\_basics](https://developer.mozilla.org/en-US/docs/Learn/Accessibility/WAI-ARIA_basics).
  56. *Patterns | APG | WAI | W3C* [online]. [visited on 2024-01-23]. Available from: <https://www.w3.org/WAI/ARIA/apg/patterns/>.

57. *Web Content Accessibility Guidelines (WCAG) 2.2* [online]. 2023-10-05. [visited on 2024-01-23]. Available from: <https://www.w3.org/TR/WCAG22/>.
58. *Accessible Rich Internet Applications (WAI-ARIA) 1.2* [online]. 2023-06-06. [visited on 2024-01-23]. Available from: <https://www.w3.org/TR/wai-aria-1.2/>.
59. SCHWERING, Ramona. Three common types of test automation [online]. 2023 [visited on 2024-01-25]. Available from: <https://web.dev/articles/ta-types>.
60. SOLOMON, Andrew. Introduction to Automation Testing [online]. 2023 [visited on 2024-01-25]. Available from: <https://medium.com/@andysolomon/introduction-to-automation-testing-ed50ac969bd2>.
61. KNIGHT, Andrew. Testing Storybook Components in Any Browser – Without Writing Any New Tests! [online]. 2022 [visited on 2024-01-25]. Available from: <https://applitools.com/blog/storybook-components-cross-browser-testing/>.
62. *Testing – Lit* [online]. [visited on 2024-01-25]. Available from: <https://lit.dev/docs/tools/testing/>.
63. KAROULLA, Andrico. Defining your UI library’s testing strategy [online]. 2021 [visited on 2024-01-25]. Available from: <https://blog.andri.co/005-defining-your-ui-librarys-testing-strategy/>.
64. *Web Test Runner* [online]. [visited on 2024-01-25]. Available from: <https://modern-web.dev/docs/test-runner/overview/>.
65. *Test Runner: Browsers* [online]. [visited on 2024-01-25]. Available from: <https://modern-web.dev/guides/test-runner/browsers/>.
66. JOHNSON, Westbrook. Testing Web Components with @web/test-runner [online]. 2023 [visited on 2024-01-25]. Available from: <https://dev.to/westbrook/testing-web-components-with-webtest-runner-51g6>.
67. *Conventional Commits* [online]. [visited on 2024-02-03]. Available from: <https://www.conventionalcommits.org/en/v1.0.0/>.
68. PENG, David. Add Commitlint, Commitizen, Standard Version, and Husky to SvelteKit Project [online]. 2022 [visited on 2024-02-03]. Available from: <https://dev.to/davipon/add-commitint-commitizen-standard-version-and-husky-to-sveltekit-project-14pc>.



69. *Semantic Versioning 2.0.0* [online]. [visited on 2024-01-29]. Available from: <https://semver.org/>.
70. *Custom Elements Everywhere* [online]. [visited on 2024-01-29]. Available from: <https://custom-elements-everywhere.com/>.
71. HELLBERG, Marcus. Exploring React's new web component support [online]. 2021 [visited on 2024-01-29]. Available from: <https://dev.to/marcushellberg/exploring-reacts-newly-added-web-component-support-19i7>.
72. *Vue and Web Components* [online]. [visited on 2024-01-29]. Available from: <https://vuejs.org/guide/extras/web-components.html>.
73. RYLAN, Cory. Using Web Components in Angular [online]. 2019 [visited on 2024-02-03]. Available from: <https://coryrylan.com/blog/using-web-components-in-angular>.
74. SCAGLIONE, Jenna. Why do people use dark mode? [online]. 2022 [visited on 2024-01-29]. Available from: <https://blog.superhuman.com/why-do-people-use-dark-mode/>.



## List of Abbreviations

**API** Application Programming Interface

**ARIA** Accessible Rich Internet Applications

**CI/CD** Continuous Integration and Continuous Deployment

**CLI** Command Line Interface

**CSS** Cascading Style Sheets

**DOM** Document Object Model

**ESM** ECMAScript Modules

**FOUC** Flash of unstyled content

**HSL** Hue Saturation Lightness

**HTML** Hypertext Markup Language

**JSX** JavaScript XML

**OLED** Organic Light-Emitting Diode

**OS** Operating System

**PR** Pull Request

**RGB** Red Green Blue

**SVG** Scalable Vector Graphics

**UI** User Interface

## A. LIST OF ABBREVIATIONS

---

**UX** User Experience

**WC** Web Components

**WCAG** Web Content Accessibility Guidelines

**W3C** World Wide Web Consortium

**XML** Extensible Markup Language

**YAML** Yet Another Markup Language

---

## Contents of Attachments

readme.txt.....	contents of attachments
implementation/	
_ documentation/.....	Storybook documentation
_ library/.....	library source codes
_ package/.....	library package
_ design_system.fig.....	design system in Figma
text/	
_ src/.....	L <sup>A</sup> T <sub>E</sub> X source codes
_ DP_Fryc_Dominik_2024.pdf.....	thesis text in PDF