Degree Thesis, Åland University of Applied Sciences, Bachelor of Information Technology

PROOF OF CONCEPT THEMABLE COMPONENT LIBRARY

– with Storybook and Atomic Design

Felicia Åkerfelt, Anna Östman

2025:31

Date of approval: 04.06.2025
Academic Supervisor: Jonas Waller

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Felicia Åkerfelt, Anna Östman
Arbetets namn:	Koncepttest för tema-anpassningsbart komponentbibliotek – med Storybook och Atomic Design
Handledare:	Jonas Waller
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Detta arbete, beställt av Crosskey Banking Solutions, syftar till att utforska metodologin Atomic Design med fokus på återanvändbara UI-komponenter. Målet är att skapa ett proof-of-concept Angular-baserat komponentbibliotek med komponenter på atom- och molekylnivå, samt en Storybook instans som fungerar som centraliserad dokumentation. Komponentbibliotektet avses fungera som en gemensam resurs för olika kundlösningar.

Arbetet utgår ifrån två frågeställningar; *"Hur kan ett delat komponentbibliotek med fokus på återanvändbarhet implementeras för att tillgodose olika kunders anpassningsbehov?"* och *"Vilka fördelar innebär användningen av Atomic Design i denna typ av projekt?"* Frågorna har undersökts genom praktisk implementation och teoretisk analys.

Resultatet är ett tema-anpassningsbart komponentbibliotek med en valbar basstyling som kan anpassas genom CSS-variabler. Resultatet visar att det är fullt möjligt att skapa ett delat komponentbibliotek för att tillgodose olika kunders anpassningsbehov. En längre studie rekommenderas för att utvärdera de långsiktiga fördelarna med Atomic Design.

Nyckelord (sökord)
webbutveckling, Atomic Design, Storybook, komponentbibliotek, designsystem

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
2025:31	1458-1531	Engelska	74

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
18.5.2025	23.05.2025	04.06.2025

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Felicia Åkerfelt, Anna Östman
Title:	Proof of Concept Themable Component Library – with Storybook and Atomic Design
Academic Supervisor:	Jonas Waller
Commissioned by:	Crosskey Banking Solutions

Abstract

This work, commissioned by Crosskey Banking Solutions, aims to explore the Atomic Design methodology with a focus on reusable UI components. The goal is to create a proof-of-concept Angular-based component library featuring components at the atomic and molecular levels, along with a Storybook instance serving as centralized documentation. The component library is intended to function as a shared resource for various customer solutions.

The work is based on two research questions: *"How can a shared component library focused on reusability be implemented to accommodate the customization needs of different customers?"* and *"What are the benefits of using Atomic Design in this type of project?"* These questions have been explored through practical implementation and theoretical analysis.

The result is a themable component library with an optional base styling that can be customized using CSS variables. The findings demonstrate that it is entirely possible to create a shared component library to meet different customers' customization needs. A longer study is recommended to evaluate the long-term benefits of Atomic Design.

Keywords

web development, Atomic Design, Storybook, component libraries, design systems

Serial number:	ISSN:	Language:	Number of pages:
2025:31	1458-1531	English	74

Handed in:	Date of presentation:	Approved:
18.5.2025	23.05.2025	04.06.2025

TABLE OF CONTENTS

1. INTRODUCTION

Since the 1990s, websites and applications have grown significantly in complexity. What began as simple static pages has evolved into intricate, dynamic systems composed of interconnected components. We, the developers of today, can no longer assume the properties of the user device used to view our websites, like front-end developers before us once could. (Frost, 2016; Godbolt, 2016).

As a response to this growing complexity, design systems have become an essential part of web development. They offer a comprehensive set of guidelines and assets for creating consistent, scalable and maintainable user interfaces. A key element of these systems is the use of component libraries. Basic UI components often share the same core functionality, with their primary distinction being their styling. This makes them highly suitable for reuse across different products within an organization, with the benefit of ensuring consistency and reducing redundant development efforts.

This project has been issued by Crosskey Banking Solution, with the purpose of creating a proof-of-concept application to display and document a component library that can be shared across multiple applications. The team that initiated this project sought to explore the benefits of Atomic Design, particularly in terms of component reusability and readability within a multi-customer context, where a single project is designed to meet the requirements of multiple, distinct customer groups. The challenge in such a context is creating a system that is flexible enough to adapt to the needs of the different customers, while also maintaining a shared front-end component library to prevent duplicate development efforts.

The implementation of the project was centered around configuring a Storybook instance. Storybook is a tool for building, testing, and documenting UI components in isolation (Storybook, n.d.-f). Once Storybook is configured in a project, it provides an instance of a Storybook webpage that offers an interactive overview of UI components that the developer has configured as stories. Currently, the company maintains a separate Storybook instance for each customer front-end application.

Given the requirements outlined by our commissioner, “to explore the benefits of Atomic Design”, this project heavily relies on the methodology and workflow of Atomic Design principles. Originally conceptualized by Brad Frost in 2013, Atomic Design was later refined into an ebook published in 2016, which is publicly accessible online.

1.1 Purpose and Research Questions

The purpose of this project is to better understand if Atomic Design could be a valuable methodology to utilize in front-end development processes—both in a general context as well as in the specific work environment context that is Crosskey Banking Solution. This thesis explores the following research questions:

1. How can a shared component library focused on reusability be implemented to accommodate the customization needs of different customers?
2. What are the benefits of using Atomic Design in this type of project?

The research for this Bachelor’s thesis has included both theoretical exploration and a practical implementation. The requirements specification for the practical implementation has been developed by Crosskey and is further explained in a later chapter.

1.2 Case Company

Crosskey Banking Solution began operating in 2004 and was originally a part of Ålandsbanken. The separation of the IT system became what is today known as Crosskey Banking Solution. (Crosskey Banking Solutions, 2024).

Crosskey (Crosskey Banking Solutions, 2024) offers services that cater to all aspects of the financial industry; daily banking, capital markets, payment solutions, open finance and digital channels, lending and credit, and card solutions. Today, the company currently employs

around 350 people across its offices in Mariehamn, Turku, Helsinki and Stockholm (Ålandsbanken, 2024).

1.3 Requirement Specification

The requirements for this project were concisely defined and focused on delivering a proof of concept through a Storybook instance, showcasing multiple customers using a single shared UI component library. The key deliverables included:

- A single, deployed Storybook instance.
- A shared UI component library.
- Three mocked customer profiles with different styling.
- Components built at an atomic and molecular level.

The primary objective of the project was to implement a Storybook application that allows the user to switch between the different styling variations of the shared components, displaying a specific theme based on the selected customer. Additionally, an important aspect of the project was identifying potential limitations and challenges that could arise from using the tech stack in this manner.

We were also required to utilize Crosskey's preferred front-end framework, Angular. All other technologies and add-ons used in this project were optional; however, when additional tools were needed, we tried to prioritize using established technologies commonly used within the company whenever possible.

1.4 Limitations

This project is meant to be a proof of concept, meaning that the goal is to explore and demonstrate an idea rather than creating a complete product. As a result, not all components typically found in a component library have been implemented.

This project operated within certain constraints, including limited time, scope and resources, which influenced the extent to which Atomic Design principles could be implemented. A complete implementation of Frost's Atomic Design system would have required significant development efforts and long-term maintenance, which exceeded the scope of this project.

In light of these constraints and at the commissioner's request, we did not implement a full style guide or a complete Atomic Design system. The project focused solely on implementation of atoms and molecules, and several smaller principles described by Frost in his book were also omitted.

1.5 Key Terms and Concepts

This chapter defines key terms and concepts utilized throughout this work. It serves as a reference section, allowing readers to revisit and refine their understanding of essential topics while also simply providing an overall field context. The goal of the chapter is to eliminate the need to consult external sources. The terms are listed in alphabetical order.

Components, Component Library and Component-based Architecture

Components are reusable parts which serve a specific function (GeeksforGeeks, 2024d). A component library refers to a collection of components which holds all the predetermined and reusable UI elements (Fessenden, 2021). Some frameworks are built with component-based architecture in mind and often then refer to the reusable parts as components (GeeksforGeeks, 2024d).

Design System

Design Systems is a set of standards and building blocks that acts as a blueprint for digital products, providing teams a unified language and structure. They help the product keep the look and experiences consistent. Additionally, design systems can also help reduce the duplication of elements and patterns across large products. (Bergman, 2024). They often contain a component library, a pattern library and a style guide (Fessenden, 2021).

Design Token

Design Tokens form the foundation of a design system's visual style. They are small, reusable design decisions that define the system's appearance. A token can represent a color, font style, spacing and more. Each token is a paring of a name and a value in code, for example --color-primary: #007bff. (Atlassian, n.d.-b).

Graphical User Interface (GUI) & User Interface (UI)

Graphical User Interface is a system of interactive visual components within software. It allows users to interact with graphical elements using devices such as computers, laptops, tables and smartphones. (GeeksforGeeks, 2024c). The GUI can be described as a subtype to the User Interface (UI). User Interface (UI) encompasses both graphical and non-graphical interactions with a software, making it a broader term than GUI. (GeeksforGeeks, 2018). In this work we will be using the term UI over GUI.

Mocks

In software engineering, a mock is a simulated object or module that mimics the behavior of a real object, commonly used in testing to isolate behaviors and ensure that the systems, objects or modules work as expected. (GeeksforGeeks, 2024a).

Modularity and Modular Programming

The word modularity is defined by the Cambridge Dictionary as "the quality of consisting of separate parts that—when combined—form a complete whole" (Cambridge University Press, n.d.). In modular programming, this concept is applied by breaking down larger systems into smaller, more manageable pieces called modules. Each module should be designed to perform a specific task independently, or combined with other modules to build a larger system. Modular programming offers benefits such as; easier code maintenance, update simplicity and reusability of code across projects (Kramer, 2024).

Pattern Library

While the terms "Pattern Library" and "Component Library" are often used interchangeably, there is a distinct difference between the two. A component library specifically focuses on

individual reusable UI elements, for example a button. In contrast, a pattern library consists of collections of UI elements organized into layouts addressing broader design solutions—for example a row of multiple buttons (Bergman, 2024; Fessenden, 2021).

Responsive Web Design

Responsive web design involves creating websites that adapt seamlessly to any device, ensuring that the content remains accessible and user-friendly across different platforms. This is achieved by designing flexible layouts that automatically scale according to the screen size .(GeeksforGeeks, 2024b).

Style Guides

Style Guides are most commonly focused on the brand identity, the product's "voice and tone" and visual elements associated with that branding—including but not limited to—color, typography, trademarks, logos and print media. (Fessenden, 2021). The term Style Guide and Design system are also commonly used interchangeably, even though some recognize there is a difference between the two (Bergman, 2024).

2. THEORY

The purpose of this chapter is to provide the reader with a deeper understanding of the core principles surrounding the Atomic Design methodology, along with design systems, which play a significant role in Brad Frost's book *Atomic Design* and the methodology itself. Component libraries are also discussed, as they were a key requirement for the project. The concepts discussed in this chapter are not tied to any specific programming language or other tools.

However, a design system does not exist in isolation; it is continuously influenced by a variety of external factors and cannot be reduced to a simple step-by-step-tutorial (Frost, 2019a). The way we describe design systems in this work, supported by various sources, may not align with the definition held by others, as interpretations can vary based on specific projects and contexts. In this work, "design system" refers to the definition outlined in this chapter.

2.1 Design Systems

A design system is a set of standards for digital products that creates a unified language, ensures consistency in user experience and reduces redundancy in patterns across the system (Bergman, 2024). Design systems often contain at least a component library, a pattern library and a style guide (Fessenden, 2021). Creating a functional design system is a long term commitment that requires a significant amount of time and effort to remain relevant over time, when a part of the design system no longer reflects the current state of the product it becomes obsolete (Frost, 2016). A design system should be cross-disciplinary, both in use and development. Limiting interaction with the design system—its style guide and design patterns—to only designers and developers restricts its potential. To maximize its usefulness, a design system should serve as a shared resource accessible to the entire organization (Frost, 2016).

Figma (2023) emphasizes that a design system should extend beyond just a style guide and pattern library; it should encompass the product's whole ecosystem. Figure 1 illustrates a possible interpretation of the various components of a design system and their respective functions, offering insight how these elements can contribute to a cohesive and integrated system.

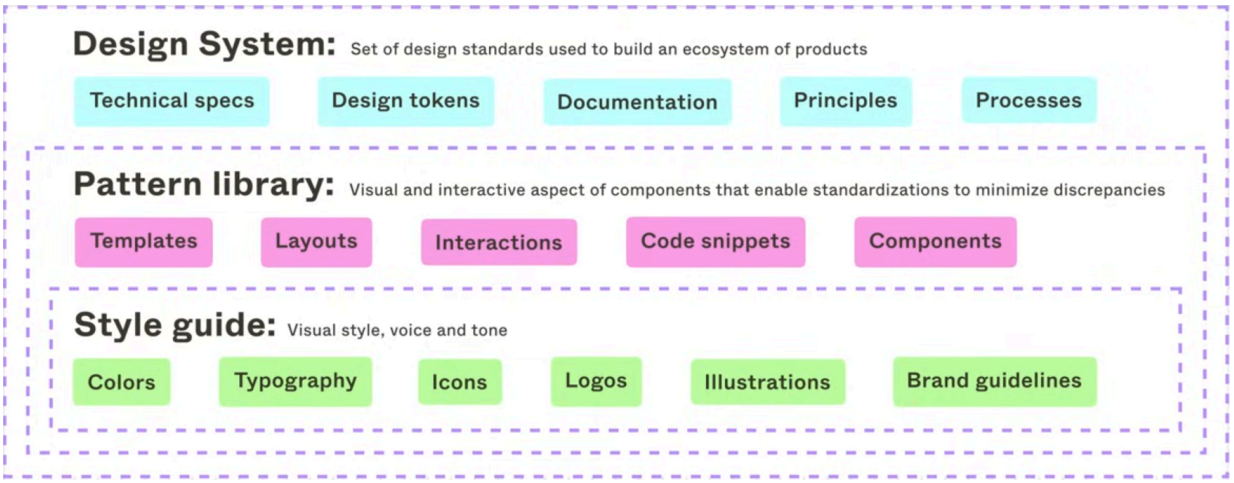

Figure 1: Example of a Design System Ecosystem taken from Youtube: "Welcome to Design System - Lesson 1: Introduction to design systems" by Figma (2023).

Frost (2019a) himself argues that the core of a design system is the library of reusable UI components that is essential to the development of functional software applications. Additionally, Frost (2019a) advocates to also utilize the design system to define organizational coding conventions and naming standards.

The benefits of a design system—as noted by Frost (2019a) and Figma (2023)—include ensuring consistency, reliability across platforms and faster development by eliminating repetitive coding of common UI components. Consequently, reducing quality assurance efforts and enhancing production quality, enabling teams to focus on iteration instead of reinvention. Acting as a single source of truth, it provides guidelines for developers and aids onboarding. Potentially, it can also simplify adopting new technologies as it harbors a future-friendly foundation for developers to modify and improve the system over time.

2.1.1 Building a Design System

Deciding whether to build a design system, and if so, when and how to begin building a design system can quickly become overwhelming. Figma (2023) claims that not everybody needs a design system, but a reliable indicator for identifying the need for a design system is to review the components in use and evaluate the presence of duplicate elements. If such occurrences are frequent, the need of a design system is apparent. On the other hand, Frost (2016, 2019a) argues that anyone utilizing reusable components could benefit by having a design system in place, and if one is not already implemented it should be prioritized. *"So when's the best time to establish an interface design system? Short answer: now" (Frost, 2016, p. 96)*. However, the approach for implementation should vary depending on where you are in a product's lifecycle.

If you already have established products, it is recommended to identify a suitable pilot project within the organization. A pilot project, much like a television pilot, serves as a test concept to assess its viability without committing significant resources to a full-scale implementation. The project should be redesigned by creating a component and pattern library to eliminate redundant design patterns and components through a front-end workshop. The final goal is to gradually incorporate every product into the design system, reusing already established components and patterns from the initial pilot project and adding new components patterns only when necessary (Frost, 2019a). Figma (2023) recommends audits to assess the current product state with a cross-platform team, offering insight into the need for a design system and helping build internal support without including actual implementation.

Additionally, Frost (2019a) believes there needs to be a designated design system champion to oversee the pilot project, advocate for its adoption and contribute to its ongoing evolution. Once established, a dedicated team should ensure its continued success (Frost, 2016). In contrast, Figma (2023) advocates selecting the team responsible for developing and maintaining the system from the beginning.

In conclusion, Figma and Frost approach the implementation of a design system with two different contexts. Figmas approach could be described as an ideal scenario, where there is

strong organizational support for building the system and the scope is either small or not a limiting factor. In contrast, Frost's approach could be described as a more flexible and pragmatic method, applicable regardless of the project's stage or scale, emphasizing the importance of adaptation to the existing circumstances.

Frost (2019a) also emphasises that an important part of developing a design system is avoiding stagnation in the prototype phase. It is essential to transition from a prototype to actually power real software. The design system is a byproduct of your product work and therefore there exists a circular flow of information between the two. By making that distinction one could argue that it is impossible to create a fully realized design system without an existing product. *"The design system informs our product design. Our product design informs the design system"* (Anne, 2015), referenced by Frost (2016, p. 151).

2.1.2 Maintaining a Design System

When the building dust has settled and the design system has been implemented, it is time to start thinking about how to maintain it. Having a pattern library, a style guide and a component library does not guarantee a design system a long term success. To ensure the design systems long-term success someone—ideally a cross-disciplinary full-time team—must take ownership of the design system and maintain, govern the system and offer support to its users. The maintainers of the design system and its users should collaborate closely to ensure that the patterns defined in the design system meet the product needs, while also ensuring the documentation is accurate and easy to understand. It can also be beneficial to involve external parties in reviewing the design system to gain insights that may not be easily identified by those within the system (Frost, 2016).

There needs to be a unified strategy on how to handle changes to patterns in the system. There are three types of changes that can happen to a pattern; modification, addition, and removal. These changes may arise from factors such as bug fixes, feature addition, visual design tweaks, code refactoring, and performance improvements. When adding new patterns, it is important to reflect whether the pattern is truly necessary, as unnecessary additions can lead to bloat, undermining the core purpose of the design system. New product needs are

often identified at the application level, but the solution should typically be addressed at the system level. This approach to managing new product needs presents an opportunity to enhance the overall framework and implement more deliberate changes (Frost, 2016).

Visibility is critical for the ongoing health of a design system, as it encourages the continued adoption and utilization of the system. Interactions with the systems and its documentation should be as frictionless as possible to also organically promote the use of it. The more visible and accessible the system is, the greater the opportunity for feedback and collaboration, enhancing its effectiveness and relevance over time. Additionally, visibility regarding changes within the design system at an organizational level is equally important and should be communicated to the entire organization to ensure future interactions and engagement (Frost, 2016).

2.2 Component Libraries

A component library is a collection of UI elements that can be reused across one or several projects. The purpose of a component library is to bring consistency and efficiency to the development process. While creating a component library requires an investment of time and resources, it leads to many long-term benefits, including faster development, easier maintenance, and improved product quality over time. Component libraries can exist in different forms, including code-based libraries or UI kits in tools like Figma. However, this thesis will mainly focus on the code-based solutions.

A component library can act as an integral part of a design system, bridging the gap between design and code by ensuring that the visual language and established design patterns are consistently applied to applications. A well-constructed component library provides a unified user experience, ensuring both visual consistency and adherence to brand guidelines (Karoulla, 2024).

By minimizing code duplication, component libraries follow the "Don't repeat yourself"[1] principle, which is a general principle of software design. This approach keeps the code maintainable and streamlines updates and bug fixes, as any changes to the components propagate throughout all instances where they are used. Having a component library forces developers and designers to carefully consider their decisions before they create a new component. It makes it easy to see what components already exist, and enables developers to reuse existing patterns (Lanciaux, 2021). Without a shared component library, teams may unintentionally create the same solutions multiple times, leading to wasted efforts and less maintainable code. The time saved can instead be redirected toward developing new features and improvements.

A component library may also be timesaving in other ways, as it can exist as a separate product within an organization and serve as a shared resource across multiple projects. This approach makes it easier to maintain design consistency across different applications, while reducing duplicated work efforts and allowing teams to create new applications more efficiently. As a shared resource, a component library can scale with an organization's needs, evolving alongside its products and adapting to new requirements as the organization grows (Martin, 2025).

In his book *Modern Front-end Architecture* (2021), Ryan Lanciaux presents the *Mise en place philosophy*, a concept borrowed from the culinary world. In French, *mise en place* translates to "everything in its place", referring to the practice of organizing and arranging the necessary ingredients and tools before the cooking begins. When applied to web development, this philosophy represents a systematic approach to preparation, organization and efficiency, establishing a solid foundation for a smooth and efficient workflow before writing code. Just as a chef benefits from having all ingredients prepared and ready, Lanciaux (2021) suggests that a developer can work more efficiently with established structures and resources—such as a component library—allowing them to focus on solving one problem at a time, rather than constantly shifting focus.

[1] Often referred as the DRY-principle

2.2.1 Third-party Component Libraries

Basic UI components often function similarly across different applications. Elements like buttons, dropdowns and input fields typically follow the same standard behaviors, making them ideal candidates for reuse. Instead of reinventing these components, using a well-established third-party library can significantly save development time.

With a wide variety of third-party component libraries available, developers can choose those that best suit their needs. Some libraries are designed for a specific framework (e.g. React, Vue, Angular), while others are framework-agnostic (Griffith, 2019).

Many third-party libraries have existed for several years and have been refined through extensive user feedback, making them a reliable source of high quality components. The more prominent and well-established ones often incorporate best practices, including accessibility standards, ensuring that the components are both inclusive and user-friendly (Karoulla, 2024).

Third-party libraries often aim to be customizable, which they achieve through various approaches with varying levels of customizability. The need for customization largely depends on the specific requirements of a project. Some libraries offer a base styling that allows for theming, enabling developers to adjust colors, typography and other variables (Griffith, 2019). Other libraries take a headless approach, meaning that the components only provide the necessary functionality without any predefined styles. This gives developers full control over the styling of the components, eliminating the need to override any existing styles, which makes it easier to integrate the components into a project's own design system (MUI, n.d.).

There are, however, potential downsides to using third-party libraries. The main limitation is that these libraries do not offer the same flexibility as custom-built components. If the component library comes pre-styled, those styles will need to be overridden to match the project's branding. Additionally, the functionality provided by the library may not always meet the project's specific needs. There are also situations where highly specialized

components are required. For instance, if the project requires data to be visualized in a certain way, it might be better to create the component in-house (Karoulla, 2024).

Furthermore, using a third-party may add unnecessary bloat to a project if not all components and functionality are used, resulting in larger bundle sizes and lowered performance. Relying on external libraries also adds a dependency where future updates could possibly introduce breaking changes (Karoulla, 2024).

2.2.2 Component Workshops

When creating a component library, it can be a good idea to develop components in isolation. This can be done by using a front-end workshop, such as Storybook or Pattern Lab. This approach separates the components from the business logic of the application, ensuring they remain modular and reusable. Lanciaux (2021) highlights three key benefits of component workshops:

1. Components will be designed to be truly reusable, focusing on their own responsibility rather than the overall page logic.
2. Developers can focus on refining one component at a time, leading to a higher quality component and, in turn, a more robust application.
3. A component workspace acts as a centralized reference, providing developers, designers and stakeholders with a clear view of all available components, which reduces code duplication.

A component workshop environment can also serve as documentation for the usage of the component library, providing essential information about how the components should be utilized.

2.2.3 Creating Components for a Component Library

Often, a component starts out as a visual design, typically created in a tool like Figma. Translating the design into a functional component requires thoughtful planning. It is a good idea to plan out different variants the component may have, as well as its behavior in different

states—such as focused, active or disabled states—and the props and inputs necessary to control its behavior. Additionally, components should be designed with responsiveness in mind, ensuring that they adapt across different platforms and various viewport sizes (Martin, 2025).

A common way to achieve visual consistency across components is by utilizing the design tokens from the design system. These tokens standardize design elements—such as colors, typography and spacing—to ensure a cohesive and scalable design. They are typically implemented as CSS variables, which can be reused throughout the styling (UXPin, 2024a).

When creating a component library, the goal is generally to create reusable components that can be used across various applications. To ensure reusability, it is essential to create components that are decoupled from the business logic and data fetching of the application. The aim is generally to create components that are presentational, in other words they should be solely focused on UI-related behavior.

It is advisable to minimize the internal state and allow consumers to control the behavior by passing props or inputs. However, certain types of components inherently require an internal state. For example, dropdowns and other complex interactive elements often require an internal state to handle user interaction.

A useful concept when designing the individual components for a component library is *the single responsibility principle*. Lanciaux (2021) points out that a component should have a single, well-defined purpose, quoting the Unix philosophy originally stated by Doug McIlroy as “Write programs that do one thing and do it well. Write programs to work together.”

Another related concept is *separation of concerns*, which is an architectural principle that emphasizes organizing code into distinct areas of responsibility, such as presentation, business logic and data fetching. This principle can also be applied to components, by grouping related functionality within one unit, while separating unrelated concerns into different components (Ramsier, 2019). Adhering to this principle has multiple benefits,

including making components easier to test and reducing code complexity, resulting in more maintainable code. It also enhances the reusability of the components as they are more likely to be able to be used in different contexts.

A common pattern in container based front-end architecture that facilitates the separation of concerns is the *container-presentational pattern*, as popularized in the React community by Dan Abramov (2015) in his article *Presentational and Container Components*. While Abramov wrote his article in the context of React applications, the pattern can also be applied in other frameworks. The pattern separates components into two categories, each with its own area of responsibility:

1. **Presentational components**, also called dumb components, with their single responsibility being to render UI elements. They are purely presentational and do not know how the data is fetched or managed, only how to display it.
2. **Container components**, also called smart components, are responsible for handling data fetching, state management and business logic. They act as a wrapper around presentational components, providing them with the necessary data.

Since the addition of hooks was introduced in React, this pattern may not be as relevant anymore, as the separation of concerns can now be handled by creating custom hooks, but the core idea of keeping business logic separate is still valuable, especially when designing a component library. The point is that the components in a component library should be "dumb"—in other words, they should not contain any application specific logic, as it would hinder the reusability across different projects. The responsibility for data management, state management and business logic should instead lie with the consumer of the library.

Although separation of concerns is generally advisable, too much decomposition can be counterproductive. The process of hiding complex implementation details by extracting parts of the code into separate functions, classes, components or modules is called abstraction (Lanciaux, 2021). Over-abstraction of components can be problematic, as it may lead to an unnecessarily fragmented codebase that is harder to understand and maintain. Lanciaux

(2021) states that it is often better to duplicate code until it is known whether the abstraction will actually be useful. According to Lanciaux (2021), a practical rule of thumb is that separating components is worthwhile if the encapsulated functionality is expected to be reused in different parts of the application.

2.2.4 Maintaining a Component Library

A component library will typically undergo several iterations, as new features and improvements are added. Proper maintenance is crucial to keep the library functional and adaptable to new requirements. Maintaining a component library requires a structured approach that includes managing dependencies, version control, testing, clear communication and comprehensive, up-to-date documentation.

To ensure that the component library remains functional, it is good practice to implement automated testing, which reduces the need for manual regression testing. For component libraries this generally involves unit testing, to ensure that each component and its internal methods functions as intended (Lanciaux, 2021). Additionally, since changing the styling of elements can accidentally affect other parts of the UI, it can be a good idea to implement visual regression testing, where the appearances of components are compared to older versions to detect unintended changes (Lanciaux, 2021). Both unit tests and visual regression tests can be configured to run automatically in the CI/CD pipeline.

Clear communication of changes is a crucial part of maintaining a component library. To keep teams informed of changes made to the components, updates should be shared through channels that are visible for all involved, for example through communication tools like newsletters or internal communication channels like Slack (Frost, 2016). Additionally, it is necessary to keep the documentation up to date, to ensure that the components can be utilized efficiently by the developers.

2.3 Atomic Design

Atomic Design is a methodology for creating design systems for user interfaces, introduced for the first time in a blog post by Brad Frost in 2013, which later was refined into a book published in 2016. Drawing inspiration from the vocabulary of chemistry, Atomic Design organizes components into five distinct stages; atoms, molecules, organisms, templates and pages, with the final two not being included in the chemistry analogy. The methodology aims to create user interface design systems in a more deliberate and hierarchical manner. By breaking down the interface into smaller, reusable components, Atomic Design encourages scalability and consistency across the design system (Frost, 2016).

In chemistry, everything can be broken down into a set of atomic elements. Frost (2016) argues that a similar approach can be applied to the HTML elements used to build user interfaces, using Duck's example shown in Figure 2. However, the chemistry analogy has its limits and should not be extended beyond the points mentioned here, as doing so could lead to misunderstandings about the true nature of Atomic Design.

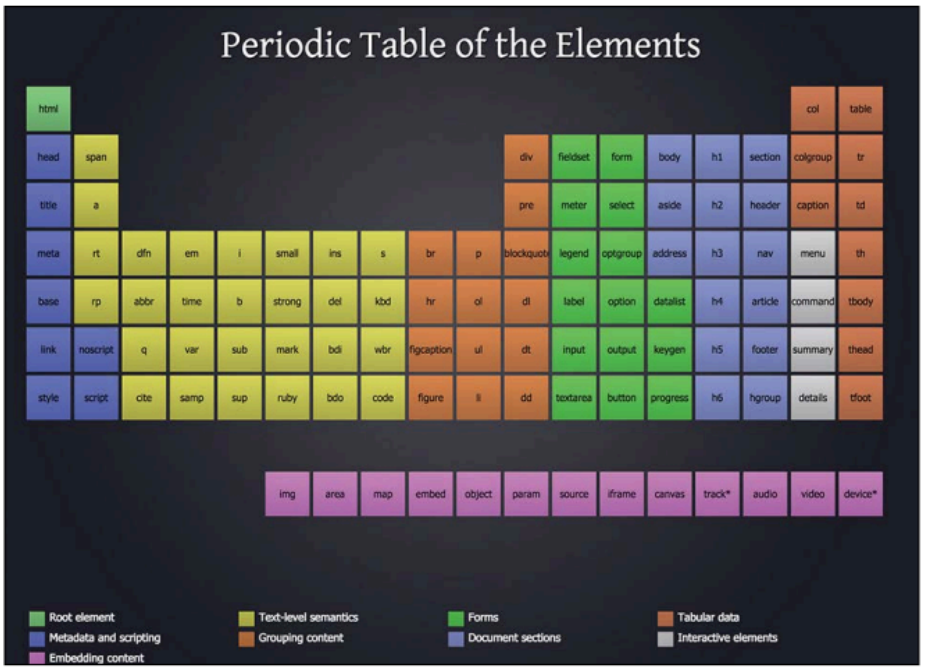

Figure 2: Josh Ducks Periodic Table of HTML Elements taken from the Atomic Design Book by Frost, B. (2016, p. 41).

Even though Atomic Design often is presented in a linear way, it is not a linear process. It is a mental model that can help developers focus on both the broader structure and the individual components and how these elements interact with each other (Frost, 2016).

In practice, Atomic Design allows for an iterative approach, where components may evolve and be refined over time. For instance, the earlier stages—such as atoms or molecules—may be revisited when new requirements or insights arise, ensuring that the design system remains flexible and adaptable to changes throughout the product's lifecycle (Frost, 2016).

2.3.1 The Building Blocks of Atomic Design

This project will only focus on implementing atoms and molecule components, but to understand Atomic Design it is important to present all the different components that should be used for a successful and complete implementation.

Atoms

Atoms represent the smallest and most fundamental building blocks of a user interface. In the terms of the web, this includes foundational HTML elements, for example a button, which cannot be broken down further without losing its functionality (Frost, 2016). Some other examples of atoms include links, labels, input fields and simple selectors such as radio buttons and checkboxes. An example can be viewed in Figure 3.

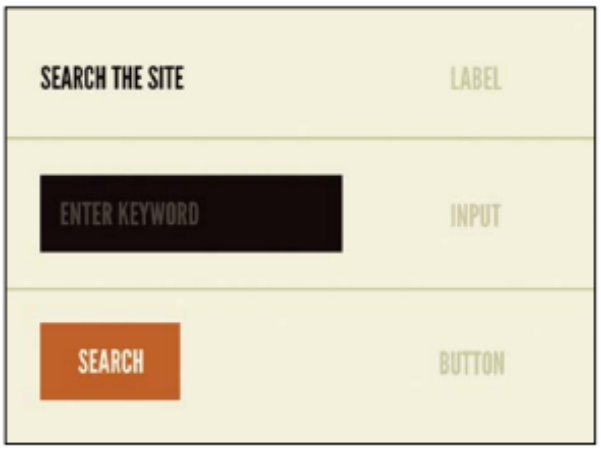

Figure 3: Brad Frost's atom example from the Atomic Design Book, by Frost, B. (2016, p. 43).

Molecules

Molecules represent a simple grouping of atoms that together take on a distinct new functionality. For example, a simple search bar molecule could be composed of three components: a button atom, an input field atom and a label atom (Frost, 2016). An example can be viewed in Figure 4.

Figure 4. Brad Frost's molecule example from the Atomic Design Book, by Frost, B. (2016, p. 45).

Organisms

Organisms are more complex than their predecessors and can be composed of groups containing molecules, atoms, or other organisms. They are important for providing designers and developers with a sense of context by combining smaller components into functional, reusable sections of a UI. Examples of organisms include a product list, a navigation bar or a form. An example can be viewed in Figure 5.

Figure 5. Brad Frost's organism example from the Atomic Design Book, by Frost, B. (2016, p. 46).

Templates

Templates are page-level objects which define the design's underlying content structure and place the front-end component—the organisms and molecules—in a layout. Their purpose is to shape how elements are arranged rather than determine the actual page content (Frost, 2016). An example can be seen in Figure 6.

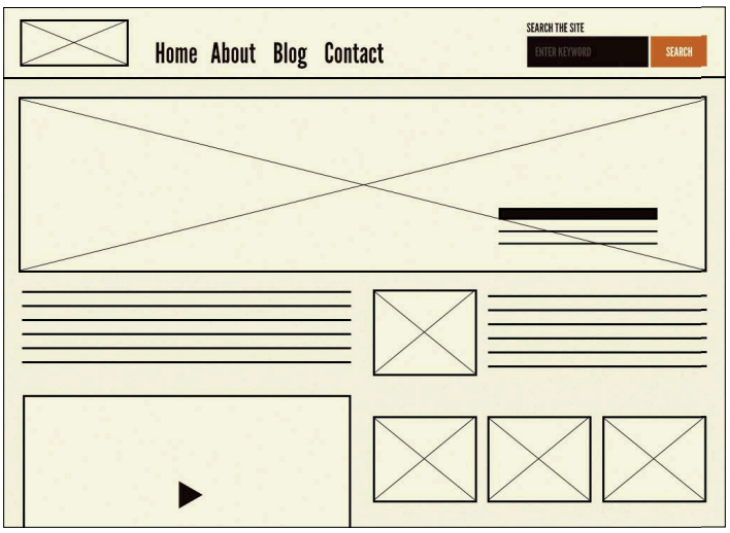

Figure 6. Brad Frost's template example from the Atomic Design Book, by Frost, B. (2016, p. 49).

Pages

Pages represent a fixed instance of templates, where dynamic content is incorporated into the predefined structure (Frost, 2016). In Frost's (2016) example, the transition from template to page involves populating the template with text, images and video media to create the final showcase page. This is the most concrete stage of atomic design, as we can see all the components coming together to form a fully functional UI experience. An example can be viewed in Figure 7.

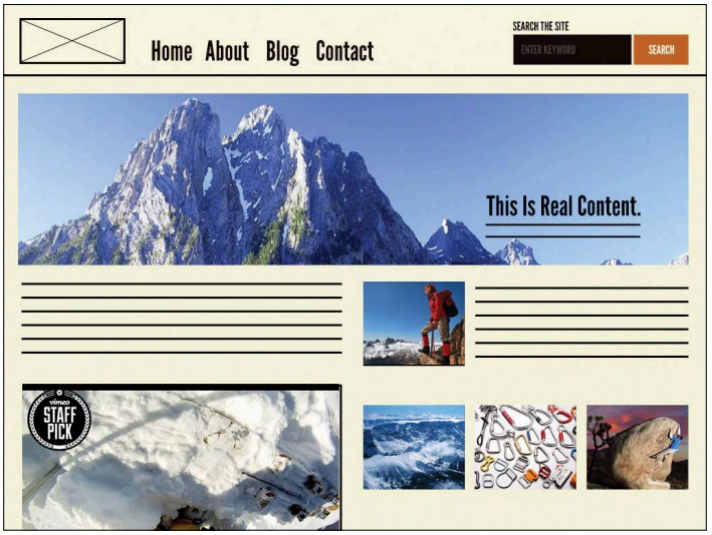

Figure 7. Brad Frost's page example from the Atomic Design Book, by Frost. B. (2016, p. 53).

At this stage the developer is able to see if earlier patterns are holding up when real content is applied. If something is not working, the developer can circle back to previous stages and modify them to fit the content's needs. Testing the resilience and adaptability of the design is a critical aspect of this phase, as it ensures the system can handle diverse content and maintain consistency (Frost, 2016).

2.3.2 New Extensions to Atomic Design

In 2019, Frost addressed some new concepts from the evolving field of design systems and their relationship with Atomic Design in a dedicated blog post.

In the context of design systems, design tokens are reusable values that define visual properties such as color, spacing, typography, border radius and shadows. They ensure consistency across a design system and serve as the foundation of a design language. A well-defined style guide typically references design tokens and provides clear guidelines for their effective use, ensuring consistent implementation across a product or brand. Frost (2019b) means that design tokens are not the same as atoms, and should rather be seen as subatomic particles, as illustrated in Figure 8, to expand on the chemistry analogy. Unlike atoms, design tokens do not exist independently in a user interface; they only appear as properties of other elements.

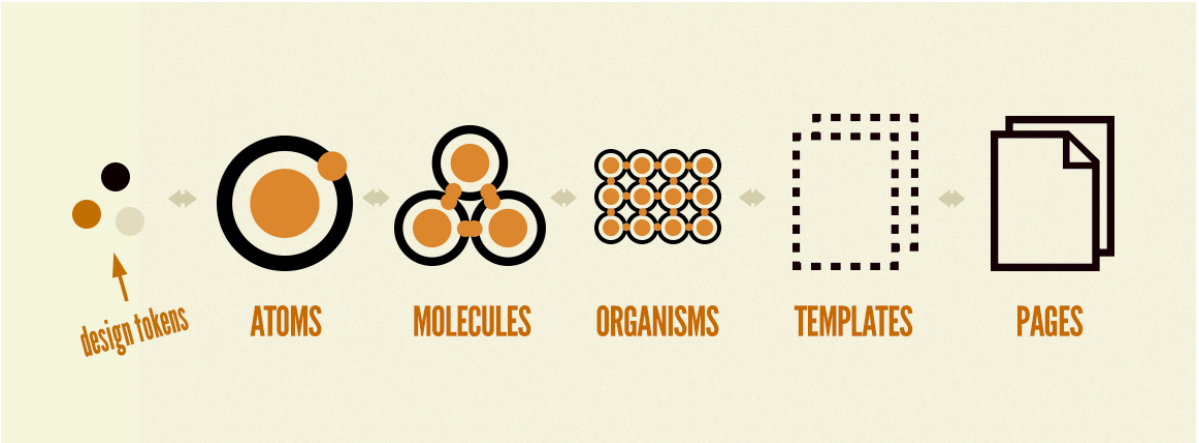

Figure 8: The Atomic Design hierarchy with design tokens by Frost (2019b).

Another more recent illustration by Frost (2019b), shown in Figure 9, depicts the design system ecosystem within the context of the Atomic Design methodology. It clarifies what parts belong to the design system—atoms, molecules and organisms—and which belong to the product—template and pages. These two areas, the system and the product, should influence one another, establishing a circular dependency. While this important iterative relationship within design systems has been discussed in previous chapters, it has not yet been explored within the specific context of Atomic Design. The figure does a good job of showcasing how the Atomic Design methodology interacts with a design system (Frost, 2019a, 2019b).

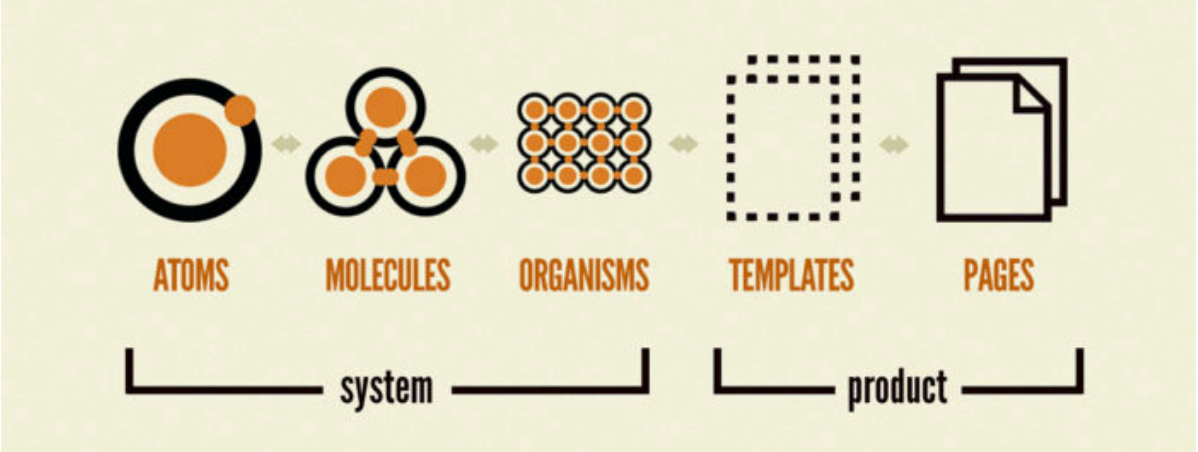

Figure 9: The Atomic Design component types in relation to the product and the design system by Frost (2019b)

2.3.3 Benefits of Atomic Design

According to its creator Frost (2016) the main advantage of Atomic Design is its ability to shift between the abstract and the concrete. That is to say, this approach allows developers to simultaneously view the interface at an atomic level and observe how these individual components combine into the final experience. "*The parts of our designs influence the whole, and the whole influences the parts. The two are intertwined, and atomic design embraces this fact*." (Frost, 2016, p. 57). Frost (2016) also highlights the methodology's ability to distinctly separate structure (templates) from content (page) while still recognizing that the template and page phases influence one another and must work together in harmony.

Atomic Design also shares the same strengths as design system and component library, focusing on reusability and encouraging teams to focus on building new features rather than repeatedly reinventing existing ones, leading to a more streamlined and effective development process.

Another benefit is the approachable taxonomy. The terms used in Atomic Design, atoms, molecules, organisms, templates and pages implies a clear hierarchy, which can be easily understood by individuals with a basic knowledge of chemistry. This naming structure allows for comprehension of the relationships and order of the components by the naming alone. Atomic Design's naming of hierarchical components also provides a shared language for both developers and designers, facilitating improved communication and collaboration across roles. However, if the terminology used in Atomic Design does not resonate with a team wanting to adopt its principles, the naming conventions can easily be adjusted to suit the team's preferences (Frost, 2016).

2.3.4 Criticisms and Limitations

According to Martinez (2023) there are certain contexts where Atomic Design may not be the most suitable approach, where Atomic Design introduces unnecessary complexity. These contexts include:

- Smaller projects with simpler requirements and short timelines.
- Projects with unique design that deviates significantly from predefined pattern.
- Single-purpose websites.
- Projects with evolving requirements.
- Projects already adhering to a well-established design system.
- Projects with a tight deadline.

Martinez (2023) also comments that the hierarchical structure and its intricate relationship between elements—atoms, molecules, organisms, templates and pages—can become a difficult learning curve, making a swift adoption to the system harder for new developers.

It should be emphasized that Atomic Design does not provide a methodology for how to structure a whole software project. Instead, it is solely focused on the design system and presentation layer of an application, without providing any guidance on business logic (Levin, 2025).

Atomic Design has faced some criticisms for being outdated in certain regards, particularly in its categorization of components. Since the framework's introduction in 2013, UI design has evolved significantly, and according to Jankowski (2024), the methodology needs an update to better align with the increased complexity of modern interfaces. Jankowski, for example, argues that a button today is more complex and should be considered a molecule rather than an atom to preserve the concept of the atom as truly indivisible.

Frost (2024) himself has also offered criticisms and improvements to his work in later years, which is relevant for anyone wanting to create their own design system. The problem Atomic Design originally aimed to solve gets pushed "forward". Instead of duplicated components we get duplicated design systems. After more than 10 years of Atomic Design, the same issues persist—everyone is still creating their own solutions: the same buttons, accordions, login forms and so on. In response, Frost presents the solution of a singular global design system that can serve as a stylefree base for the common components.

3. TECHNOLOGIES

The following chapter presents the different frameworks and tools used to implement the project. As previously mentioned, Angular was utilized as the framework for development, in accordance with the project requirements. The use of Storybook was also a requirement, while Nx was chosen independently, based on its usage in similar projects within the company. Netlify was utilized to deploy a static website of a Storybook instance, bypassing the complexity of the company's standard deployment pipeline. Given that the project was a proof of concept, it was determined that resources and time would be better allocated to other aspects of the implementation.

3.1 Angular

Angular is a framework for building web applications, maintained by Google (Angular, n.d.-c). Angular is open source, and the codebase can be viewed on the cloud-based version control platform GitHub (Angular, n.d.-c; Github, n.d.).

Angular uses a component-based architecture, making development modular and scalable. An Angular component consists of four main parts:

1. a `@Component` decorator that defines configurations for Angular
2. a TypeScript class that manages component behavior and logic
3. an HTML template for dynamic rendering of content in the DOM[2]
4. a CSS selector that defines how Angular identifies and applies the component within the DOM (Angular, n.d.-a).

Additionally, Angular incorporates dependency injection through services, which facilitate logic and state sharing between components (Angular, n.d.-b).

[2] Document Object Model, a programming interface that represents the web page's structure

3.2 Storybook

Storybook is a front-end workshop for building UI components in isolation. It allows developers to visualize and test components individually, effectively separating the user interface from the application's business logic, data and APIs (Storybook, n.d.-f, n.d.-h). Storybook serves multiple purposes, including UI development, documentation, as well as testing. It is especially useful for testing component states that would be difficult to recreate within the application (Storybook, n.d.-f).

Storybook is designed to be compatible with a range of web frameworks, including more well-known ones such as React, Vue, Angular and Svelte (Storybook, n.d.-g).

3.2.1 The Storybook User Interface

Storybook provides a user-friendly interface that can be accessed through a web browser. The interface is designed to be used by designers, developers and testers alike, allowing them to view components in various states and modify their input via the Storybook's user interface, without the need to modify the code (UXPin, 2024b).

The Storybook interface, as presented in Figure 10, maintains a consistent layout across all implementations, with variation based on the components and the developers choices regarding their display configurations and addons. The interface is composed of several key elements, this is a simplified explanation of the most prominents ones:

1. **Sidebar**: indicated by the blue border, enables navigation between components and documentation stories.
2. **Toolbar**: indicated with a red border, providing additional controls for tweaking the canvas.
3. **Canvas**: indicated by the yellow border, is the central preview area where the selected component and story is displayed and can be interacted with.
4. **Panels**: indicated with a black border, is an area for different types of panels that provide additional controls or information within the Storybook interface. The control

panel, which is the active panel Figure 10, enables real time adjustments of arguments passed to the story.

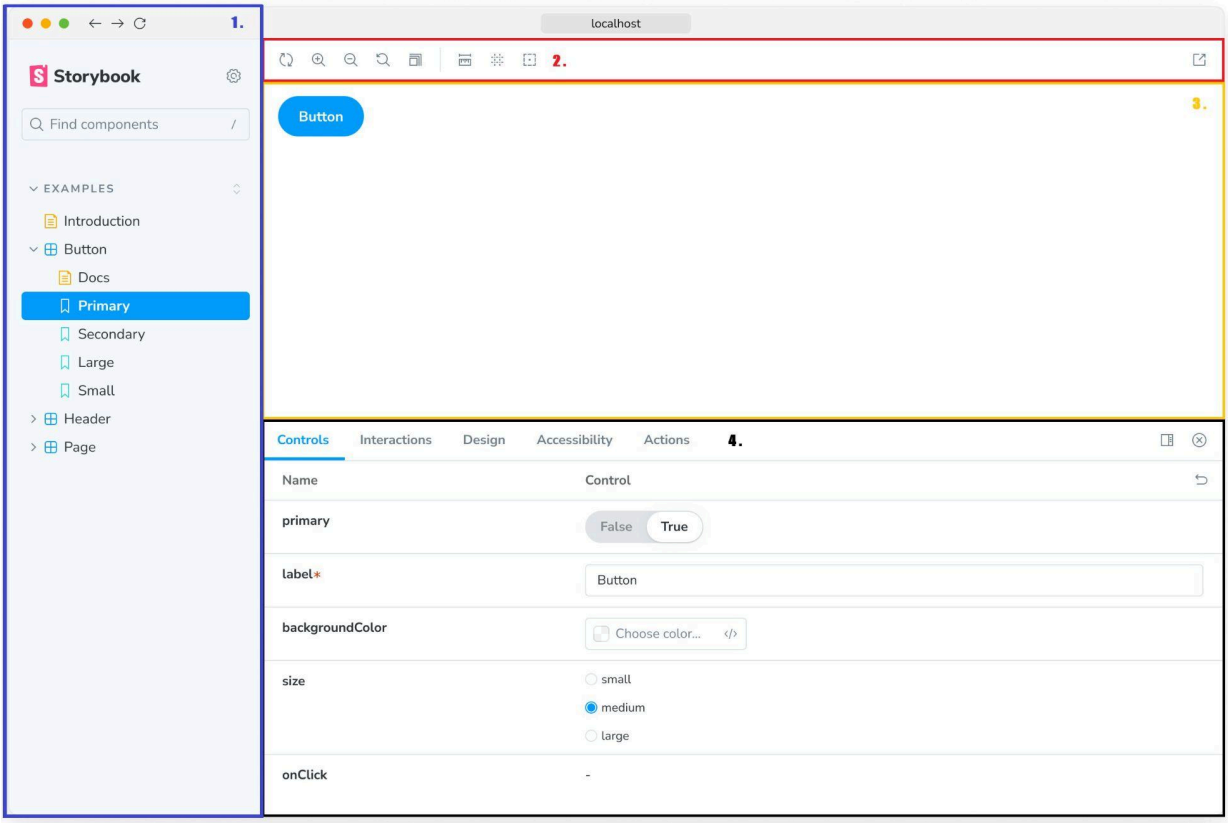

Figure 10: The Storybook interface with colored borders highlighting key elements processed from Storybook's official website (Storybook, n.d.-l).

3.2.2 Stories

Stories are representations of a state of a UI component and can be previewed in the Storybook canvas. It is common to define multiple stories for the same component to display different states. The stories are defined in a separate file from the component, usually with the file extension .stories.js or .stories.ts, and are written in Component Story Format[3], based on the JavaScript ES6 module standard (Storybook, n.d.-l).

To create a component state, the developer can define a set of arguments[4] which are used as inputs to the story, where it can be passed down to the component within it. The values of

[3]Can be referred to as CSF
[4] *In code referenced as args*

these args can be manually adjusted in the controls panel in the Storybook UI, a feature that comes preinstalled with Storybook's *addon-essentials* package. The controls panel provides a user-friendly interface that can be used by both developers and designers to interact with the components without the need to change the underlying code. When running Storybook in developer mode, it is also possible to use the controls to change the arguments values and save them as a new story without the need to write the changes directly as code (Storybook, n.d.-l).

The stories are rendered in a separate iframe from the rest of the Storybook application, which means that the story preview exists in its own HTML document separate from the Storybook manager surrounding it (Storybook, n.d.-i). As a result, it is possible to inject separate CSS files into the story preview's DOM, ensuring that the styles only affect the preview and not the rest of the Storybook application, and vice versa.

3.2.3 Addons

Storybook addons are plugins designed to extend the functionality of Storybook. They allow for greater customization by enabling modification of the platform's behavior and can add extra tools to its interface, making it more adaptable to specific needs. The functionality they provide covers a wide range of use cases, including theming, automated documentation and various testing tools.

The Storybook documentation mentions two general types of addons; UI-based and presets. UI-based addons are designed to make changes to the Storybook UI, by, for instance, adding custom tools to the toolbar or new panels to the Storybook interface (Storybook, n.d.-k). It is also possible to change the rendering of stories by using addons that inject decorators, which wrap the story in extra markup (Storybook, n.d.-b).

Presets are a special type of addon that enables Storybook to integrate with other tools and services. This is achieved by injecting custom webpack configuration, which changes the build setup of Storybook (Storybook, n.d.-k).

Addons come from a variety of different sources; some are developed and maintained by Storybook's core team, who also makes sure that they are kept up to date with the latest Storybook's updates, while others are third-party, created by the Storybook community (Storybook, n.d.-e). It is also possible to create custom addons to fit specific project needs.

Storybook comes prepackaged with a set of addons called *addon-essentials*, which provides functionality such as controls for changing arguments passed in to the story, documentation, logging of user actions and a toolbar menu. A key feature of this addon package is the introduction of global variables that can be accessed throughout the Storybook application. These variables can be consumed in stories or within other addons, and can be set dynamically, for example via a toolbar item. Globals can also be read and used in decorators to control the rendering of stories. Changing the value of a global triggers a re-render, ensuring that the story and decorator rerun with the updated values (Storybook, n.d.-j).

3.2.4 Documentation

With the *addon-docs* addon, it is possible to create documentation pages to display additional information about the stories. The Autodoc feature offers preconfigured documentation for stories, with a customizable template documentation that can be modified in the documentation configuration using doc blocks, that provide premade components for creating documentation (Storybook, n.d.-c).

Addon-docs also makes it possible to create standalone documentation pages using MDX[5], a markdown format based on JSX[6]. These standalone pages are useful for showcasing design tokens, like colors and typography, or for displaying guidelines and introduction pages (Storybook, n.d.-d). Storybook offers a range of pre-made blocks, such ColorPalette, Markdown and IconGallery, to simplify building the documentation pages. These blocks can only be used within documentation, including the Autodoc pages, but not inside individual stories (Storybook, n.d.-c).

[5] Markdown is a markup language for formatting plain text, markdown files have the extension .md.
[6] JavaScript XML, a syntax extension to JavaScript. Used in React to define UI elements in a HTML-like format.

Documentation pages can also display previews of stories, which by default are rendered inline, meaning they are embedded directly within the documentation page, in the same HTML context, without isolation. This behavior can be changed to use a separate iframe by modifying the configuration. However, according to the Storybook documentation, this could lead to issues with the story not being updated when the controls change (Storybook, n.d.-a).

3.3 Nx

Nx is an open-source development toolset and build system for creating web applications. It offers many features to improve productivity and speed up build times and testing, both in local development and in continuous integration pipelines.

Nx has a powerful toolset for managing monorepos, enabling developers to efficiently manage multiple projects within a single repository (Nx, n.d.-d). The projects can share dependencies, maintain a centralized `package.json` and use libraries shared between applications. This simplifies code management, reduces code duplication and improves consistency between projects.

One of Nx's prominent features is its build system which optimizes build times by running only the necessary tasks impacted by code changes. Nx achieves this by tracking the project dependencies and caching task results, reducing redundant calculations and significantly speeding up execution time (Nx, n.d.-c).

Beyond build optimization, Nx boosts developer productivity through code generation and scaffolding of new applications, libraries and components. By incorporating best practices, Nx ensures high code quality and consistency across projects (Nx, n.d.-a). Nx also supports advanced testing workflows and can detect flaky tests to automatically rerun them as necessary (Nx, n.d.-e).

The tool is particularly popular within the JavaScript and TypeScript ecosystem, and is compatible with various frameworks, including Angular, React and Vue (Nx, n.d.-b). Nx also

enables easy integration of tools such as Storybook, Tailwind into projects, making it a versatile tool for web development (Nx, n.d.-e).

3.4 CSS

Cascading Style Sheets (CSS) is a standard styling language used to design layout and visual aspects of a web page, describing the presentation of HTML elements. A CCS rule consists of a selector and a declaration block, where each declaration is a property-value pair. The selector determines which HTML element the rule applies to. An overview of the rules is provided in Table 1.

Table 1. Overview of CSS selectors, their syntax and usage (CSS Syntax, n.d.).

Selector	***Syntax***	***Usage***
Universal selector	*	Selects all elements.
Element selector	e.g. p, h1	Targets specific HTML element types.
Class	.class-name	Applies to elements with a specified class attribute.
ID	#id-name	Should be unique within a page, applied to elements with a specific id attribute.
Pseudo-classes	e.g. :hover, :focus, :first-child, :last-child	Applies styles based on element states (e.g. :hover, :focus) or structural relations (e.g. :first-child, :last-child).

The term “cascading” in Cascading Style Sheets refers to how CSS rules are applied when there are multiple rules targeting the same element. CSS follows a specific hierarchy to determine which rule takes priority. The following is a simplified hierarchy of the rule application process outlined in MDN Web Docs (2025):

1. **!important rule** – Rules with the `!important` annotation take precedence over all others.

2. **Origin type** – Styes can have different origins and are applied in this order, from highest precedence to the lowest:
    a. inline styles (`style=""`)
    b. internal styles (`<style>` tags)
    c. external stylesheets
    d. browser defaults
3. **Specificity** – More specific selectors override less specific ones. For example, IDs (`#id`) take precedence over classes (`.class`), which in turn override element selectors (e.g. `div, p`).
4. **Order of appearance** – When rules share the same specificity and origin type, the last one takes precedence.

It is good practice to use low specificity, particularly when creating a component library with base styles that are intended to be easily customized and overridden (Serban, 2023). High specificity can create conflicts and make it difficult to apply custom styles without using cumbersome overrides and forceful methods, such the use of `!important`.

While low specificity promotes flexibility, the rules should be carefully managed to avoid unintended side effects, as broadly defined styles can leak out and affect other parts of the application. To prevent these issues, developers could use namespacing with class-based selectors or CSS modules to keep styles localized to their components. Naming conventions such as BEM[7] can be useful to keep styles modular and reusable, while reducing the risk of conflicts. Proper scope management is crucial to avoid unintended overrides and ensure that the styles remain contained within the intended scope (Serban, 2023).

3.4.1 CSS Variables

CSS variables, also known as custom properties, provide a way to store reusable values in a stylesheet. They help ensure design consistency and simplify updates, as the value only needs to be changed in the declaration for the change to take effect everywhere the variable is used. CSS variables are declared using a syntax starting with two dashes followed by the name of

[7] Block Element Modifiers

the variable and can be referenced using the var() notation. As shown in Figure 11, CSS variables are usually declared in the pseudo-class `:root` to be globally accessible throughout the stylesheet, but can also be declared in other selectors if different scoping is required (MDN Web Docs, 2024).

```css
:root {
  --primary-color: #3498db;
  --secondary-color: #2ecc71;
  --font-size: 16px;
  --spacing: 8px;
}

body {
  background-color: var(--primary-color);
  font-size: var(--font-size);
  margin: var(--spacing);
}
```

Figure 11: Example of CSS variables declared in :root and applied inside the body tag selector.

3.5 Sass

Sass is a preprocessor scripting language that extends the functionality of CSS. Sass files have the .sass or .scss file extensions and need to be compiled into CSS before they can be used in web applications. Some of the extended functionalities of Sass include the ability to use partials and mixins, which help code organization and reusability (Sass, n.d.-a).

Partials allow styles to be split into multiple files for better maintainability. A partial file has a file name with a leading underscore to indicate to the compiler that the file should not be generated into its own CSS file. These files are typically imported into other Sass files using the `@use` annotation which loads the file as a module. This also makes it possible to use variables and mixins defined in the included file (Sass, n.d.-a).

Mixins are groups of CSS declarations that can be reused throughout a project. They function similarly to variables, but with the added benefit of accepting input values, which makes them useful for creating dynamic styles, such as theme variations (Sass, n.d.-a).

3.5.1 Sass Variables

While CSS variables can be used in Sass stylesheets, Sass also has its own type of variables with different syntax and functionality. The syntax for Sass variables uses a dollar-sign symbol ($) as a prefix followed by the name of the variable. The variables do not need to be defined inside a selector, as opposed to CSS variables (Sass, n.d.-b).

One of the main differences between CSS and Sass variables is that Sass variables are processed during compilation, where their actual values are being inserted into the final CSS output. In contrast, CSS variables are directly processed by the browser at runtime, and therefore do not need to be compiled (Sass, n.d.-b). After compilation, Sass variables are static and cannot be modified dynamically using JavaScript. CSS variables, however, offer more flexibility, as they can be changed at runtime, making them more suitable for theming and dynamic design adjustments (Sass, n.d.-b).

Another difference is that CSS variables are declarative, while Sass variables are imperative. This means that the value of CSS variables can be overridden and the change affects earlier uses as well. Sass variables, on the other hand, will use the current variable declaration, so a change in value will only affect the code that appears after it (Sass, n.d.-b), as demonstrated in Figure 12.

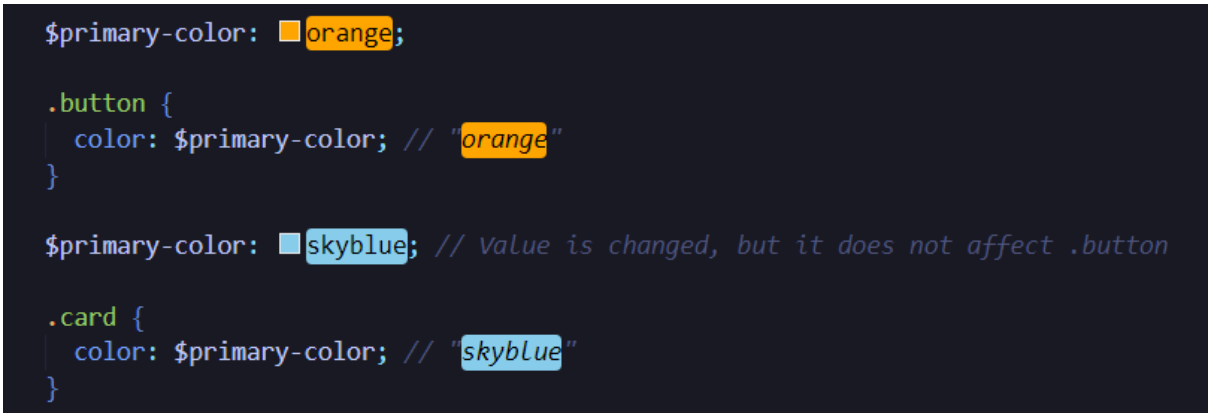

Figure 12: Code snippet illustrating the imperative nature of Sass variables.

3.6 Netlify

Netlify is a cloud computing platform that simplifies web hosting and deployment (Rafalski, 2025). They describe themselves as *"a framework-agnostic composable platform that empowers enterprises and cross-functional teams to build high-performing modern web experiences that can scale, innovate and ship to production faster." (Netlify, 2024).*

They offer a tiered pricing structure with a free tier and an enterprise solution. Some key features of Netlify include continuous deployment, serverless functions, static website hosting, designed for speed enhancements and security and ease of management (Rafalski, 2025).

4. METHOD

This chapter outlines the strategies employed to effectively plan and organize the project, offering insight into both the preparatory stages and the practical steps taken to carry out the project successfully.

4.1 Pre-project Preparations

The initiation for the project was conducted during working hours. Since two separate teams were involved, it was necessary to consult with the respective team managers from each group. An internal discussion regarding the subject for the project was initiated, although we were not involved in the beginning. The subject “Atomic Design” was later presented to us as an option, and we deemed it a suitable topic for the thesis.

After the subject had been decided, we scheduled a kick-off meeting with the two Team Managers, a System Architect, an Interaction Designer and a Consultant who had prior experience with Atomic Design and Storybook. The meeting primarily focused on defining the project scope, aligning with the requirements set by the school and confirming that the project was indeed feasible as a thesis project.

Before any implementation took place, an additional meeting was held with the Interaction Designer and System Architect to further define the requirement specification in more detail and to align our visions. In preparation for the project, we also thoroughly read Brad Frost’s book Atomic Design, since we knew it would be essential to the project that we had the methodology fresh in mind.

4.2 Project Organization and Planning

Development occurred one to two days a week—from October 2024 to February 2025. In the beginning of development, we—the two developers—worked tightly together to form the

project foundation and file structure, encompassing both the Angular code file structure and the Storybooks interface and sidebar structure. Once a stable file structure was established, tasks were individually assigned based on priority and development work became more focused on individual implementation and on distinct functionality.

Prior to each development session, we coordinated our schedules to identify mutually available days for collaboration. During these sessions, we consistently worked side by side in our designated office in Mariehamn, simplifying communication about implementation strategies and problem solving. This setup encouraged ongoing cooperation—even when working on different functionalities—fostering a collaborative environment for sharing insight and resolving challenges. A document was maintained to record the development work completed each day, along with notes explaining the rationale behind some coding decisions.

4.2.1 Development Environment

IntelliJ IDEA was chosen as our development environment—drawing on prior familiarity from earlier development work—in combination with Nx for generating boilerplate[8] code and tests. To maintain code quality and enforce an uniform coding style, ESLint and Prettier were integrated into the IDE.

4.2.2 Bitbucket

The version control of the code was managed using Bitbucket, which is a Git based code hosting collaboration tool created for teams (Atlassian, n.d.-a).

We had full control over the repository where our code resided, which made adding new features more efficient by allowing us to work outside the usual code review rules of the organization. The version control system also enabled us to work in parallel, with each team member able to develop features independently.

[8] Standardized setup code that is repeatedly used with little to no variation

The implementation strategy involved creating a separate branch for each new feature, with the intention of merging the completed feature back into the main branch. Additionally, feature branches were often maintained independently for a period of time, in case there was a need for the isolated feature.

4.2.3 Jira

To track tasks—including their status and priority—we used Jira, as it is the company's designated project management tool. Jira is a project management software to help teams plan, organize, manage and track their work made by Atlassian. It is not designed for any specific type of team or industry and can be applied to any project (Atlassian, 2025).

We were given the flexibility to define our own tasks within Jira. In an effort to maintain an agile approach, we initially created foundational tickets that we anticipated would be necessary. As the implementation progressed, existing stories were updated and new ones added. Additionally, we created some tickets with the knowledge that they would have low priority, but they were kept in the backlog for future consideration.

4.2.4 Agile Feedback Meetings

Feedback meetings were held at the project's midpoint and conclusion. While these meetings were not initially planned, it felt both natural and necessary to have some form of feedback meeting to gain external perspectives and ensure ongoing alignment. For the midpoint meeting we again met up with the System Architect and the Interaction Designer to review the progress and address potential adjustments. The conclusion meeting included all the participants from the initial kick-off meeting and the final result of the project was presented.

Following the midpoint feedback meeting, some adjustments were made to the project requirements. Initially, the project began with the requirement that the themes should be distinctly different, but discussions led to the idea of exploring how a base styling could be applied to the components, while still allowing for flexibility through customer-specific customizations. Another new feature that was proposed was the possibility of locking the

write access of atoms and molecules, ensuring that only authorized individuals could modify the fundamental front-end components.

4.3 References and Assessing Reliability

Since the start of this thesis, it was clear that sourcing reference would pose a significant challenge, since Atomic Design is a singular methodology and concept primarily documented in a single book by one author. However, a key advantage has been that Atomic Design draws upon pre-existing concepts, allowing for parallels to be made with ideas presented in other works. The reliability of certain concepts explained in this thesis can be questioned in this context, for instance whether Figma's interpretation of design systems aligns with Frost's definition. There is also the risk of misinterpreting information, potentially treating similar concepts as equivalent when they are not.

Another challenge in sourcing references arises from the fast-paced nature of the IT industry, where technologies can evolve in a matter of months, quickly rendering information outdated. To address this, we tried to use official documentation wherever possible, especially in Chapter 3. Technologies. We also faced difficulties finding relevant academic sources, which despite being considered more reliable, are similarly affected by the industry's rapid advancements. As a result, most of our resources were gathered from the internet, as well as recorded lectures and presentations on YouTube.

When sourcing information from the internet, we made an effort to evaluate the reliability of the websites and determine whether the information might be AI-generated, based on publication date and the language used. However, accurately assessing whether information is AI-generated feels nearly impossible as of today, as AI has become highly proficient in closely resembling human-produced content. Nevertheless, we made a conscious effort to keep this in mind when gathering information.

5. DEVELOPMENT

This chapter details some key practical implementations of the project, focusing on implementation where the development approach and underlying thought process may provide value to others looking to build similar solutions in the future. It aims to share lessons learned, showcase decision-making and offer insight into encountered challenges based on project goals and chosen technologies.

5.1 Stylesheet and Theme Implementation

A significant part of the project involved the implementation of theming to accommodate the different customization needs of the mock customers. We chose Sass for the styling implementation, due to its extended feature in comparison to standard CSS, as well as its ability to simplify managing complex stylesheets in large projects—a necessity if the application would be implemented in a real context in the future.

Throughout this project, the approach to styling evolved based on shifting requirements. Initially, the implementation followed a headless approach that completely left out any predefined styling to allow for maximum flexibility when defining styles for the applications. However, as these requirements changed, a base styling for the component library was introduced, with theme support through CSS variables. While this approach provides a structured starting point and simplified usage of the components, it inherently limits—or at least complicates—the styling flexibility compared to a fully headless approach. However, through careful management of CSS specificity, styles can still be made to remain overridable when necessary.

As our styling approach evolved, we came to understand that different projects benefit from different strategies. Base styling suits standardized applications or solutions where the customization needs can be solved through themes, while a headless approach offers flexibility for strict branding requirements. An important realization was that these approaches do not have to be mutually exclusive. Instead of choosing between base styling

and a headless approach, it is possible for a component library to provide them both. With this in mind, we chose to make the base styling optional, so that developers can select the approach that best fits the project's requirements. This way the component library could support a wider range of use cases.

To accomplish this, we deliberately avoided directly importing the styling into the component files, which is otherwise a common practice in Angular development. This approach keeps the component implementation separate from the styling rules and allows the component library to be used in a headless manner, giving developers the flexibility to decide whether they want to import the base styling or define their own styling from the ground up.

5.2 Dynamic Documentation Pages

As different customers were expected to have their own distinct design token values, we needed a way to display customer-specific documentation in Storybook based on the selected theme. Our goal was to have a single Storybook page per design token category, rather than creating separate pages for each customer. However, we were unable to find a way to make Storybook’s default MDX format support dynamic content, making it impossible to tailor a single page to different customers with runtime theming. To overcome this limitation, we decided to implement our own custom solution. Our workaround involved creating an Angular component that accepts the current theme value as input and renders the corresponding child component with documentation specific to the selected customer.

Although this method does not follow standard Storybook practices, it was necessary to ensure that the documentation adapted dynamically to the different customer themes and design tokens for each customer. However, this solution comes with a drawback: it prevents the usage of Storybook’s built-in code blocks, as they are limited to MDX. As a result, this setup increases development time when creating the documentation, as the solutions provided by the code blocks would have to be recreated manually in Angular.

During development, we explored several alternative solutions for implementing dynamic documentation in Storybook:

- **Displaying MDX in Angular:** We initially considered displaying MDX inside a wrapper Angular component. This approach was ruled out, as Angular does not natively support rendering of MDX. It might be possible to compile the MDX to HTML before building the Storybook, although this idea was not not explored in practice.
- **Adding React components:** Another idea was to use React components for the documentation rendering, since React is JSX based and supports MDX. However, Storybook does not currently support multiple frameworks within the same Storybook instance.
- **Creating a composite Storybook:** We also looked into using a composite Storybook which merges multiple Storybooks together and displays them as a single instance. However, according to the official Storybook documentation, composite Storybooks limits the functionality of addons. Since we did not implement this solution, the exact limitations remain unclear.

5.3 Challenges During Development

Throughout the development process, we faced several challenges that impacted the progress of the project. These challenges were primarily related to gaps in documentation, difficulties in theme management and issues with configuration.

One of the challenges stems from the chosen framework, Angular. Several features of Storybook are not available when using Angular, resulting in limited access to the full range of Storybook's functionality. Additionally, the official Storybook documentation for Angular integrations contains numerous deprecated references, further complicating the implementation process with Angular.

Another significant challenge emerged with the use of CSS and Sass variables. Initially, we aimed to use CSS variables to manage themes across the application. In standard CSS, variables are typically defined within the `:root` pseudo-class to ensure global accessibility. However, this approach conflicted with how the addon we were using—the *@storybook/addon-themes*, created by the Storybook core team—worked. The addon applies a class with a name based on the current theme to the `<html>` tag of the Storybook story, which prevents the CSS variables from being applied correctly to the theme addon classes. At first, we were unsure whether investing significant time into developing a custom solution would be worthwhile, so we initially worked with the existing setup. However, as the project progressed we were able to allocate time to develop our own theme configuration which added a custom decorator to successfully resolve the issue.

Additionally, we encountered difficulties in managing separate Sass files for different themes. For our own custom theme decorator, our intention was to keep the different themes in separate files and apply them dynamically to the HTML. However, despite specifying the desired configuration, Storybook consistently bundled all compiled styles into a single main.css file during the build process, which lessened our control over the theming. As a result, we had to separate the Sass compilation process from the Storybook build and instead compile them into separate CSS files using custom CLI scripts.

Another issue encountered during the development was styling leaking into the documentation UI when *addon-themes* and *addon-docs* were used together in Storybook. This seems to be a known issue and is documented on the Storybook GitHub page as issue #29569. The problem arose because the documentation pages and the stories exist in the same iframe. Since the class is applied to the `<html>` element of the iframe, styles intended for the stories also affected the documentation UI. This was especially troublesome when the themes included a mix of light and dark themes, as a light text color made the documentation UI unreadable against the default white background of the Storybook. While Storybook does provide a way to change the background color in its toolbar, it felt unnecessarily cumbersome and resulted in a poor user experience.

6. RESULT

This chapter presents the result of the project, detailing the implementation of the proof-of-concept Storybook application and the limitations of the chosen tech stack.

The completed project successfully met the original requirement set by the case company. These requirements were to create a single deployed Storybook instance, enabling users to flip through three different styles based on the chosen mock customer. Additionally, the project required the development of a shared UI library containing components at the atomic and molecular level developed with Angular, with a base style that can be overridden with customer specific styling.

The result became a *themable* component library and a Storybook instance, designed to be shared between the three mocked customer applications. The implementation follows Atomic Design principles and includes components at atomic and molecular level. The component library features a base styling which can be utilized by all mock customers, with the option of customer-specific customization through CSS variables. The project was implemented using Angular 18.2.0 with TypeScript, Storybook 8.4.1, Nx 20.0.6 and Sass 1.77.6.

6.1 Project Structure

The project was structured as a monorepo using Nx, containing three different applications—one for each mocked customer. Additionally, a single shared library of UI components was created and integrated with Storybook.

The folder structure is illustrated in Figure 13. All customer applications follow the same structure, with customer 1 shown as an example. The `lib/components` folder contains base components within the shared-ui library. The `override-styles/components` folder contains customer-specific overriding styles.

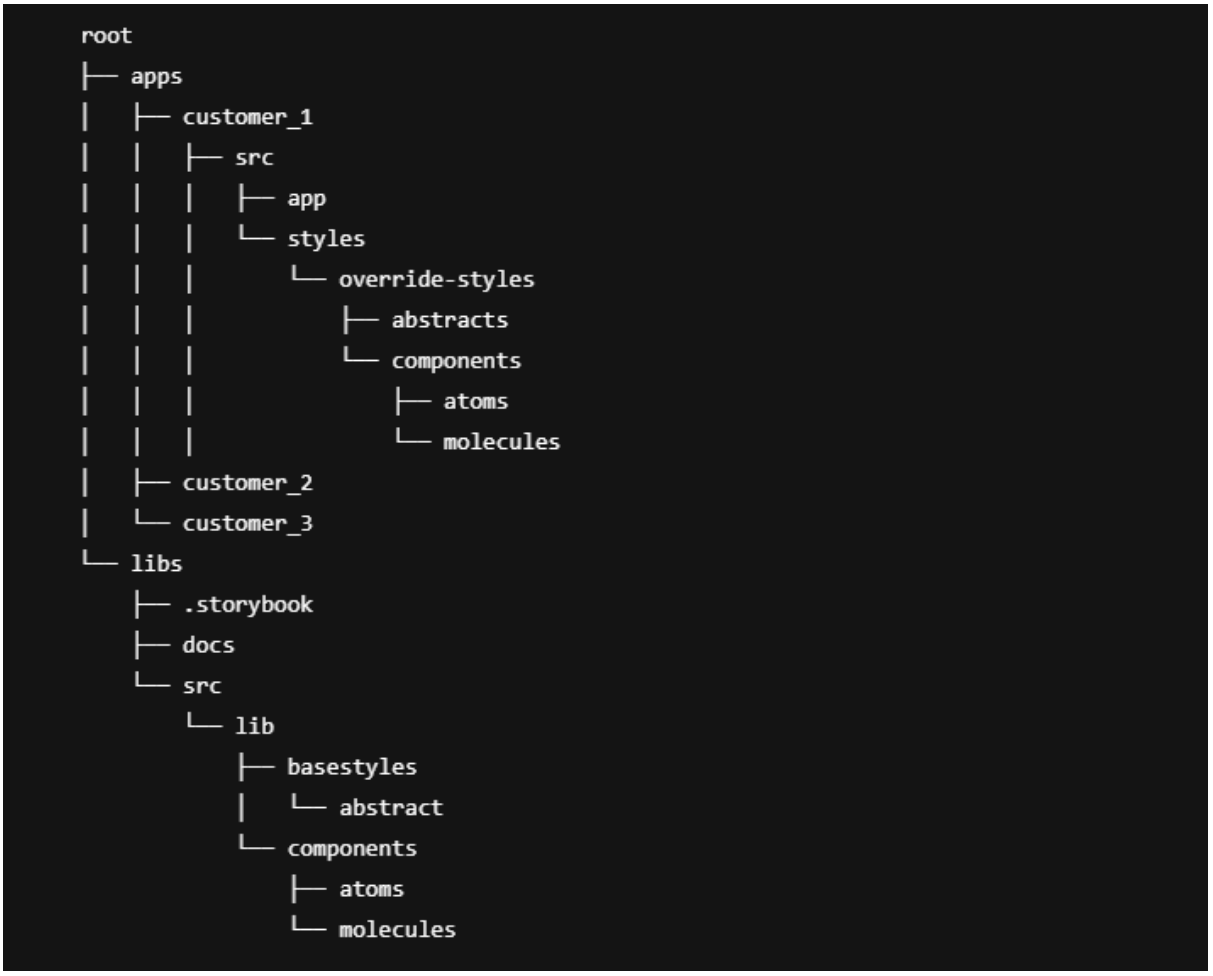

Figure 13: Simplified representation of the project's folder structure.

6.2 Component Implementation

Rather than defining the element in the Angular template, the atom components were developed using an attribute selector, a special type of component selector, to create a component based on a standard native element. This approach leverages the standard APIs and accessibility features inherent to HTML elements and maintains their expected browser behavior. By using the `host` configuration in the component's `.ts`-file the components, properties, such as `disabled`, could be bound directly to the element. An example of this can be seen in Figure 14.

```typescript
@Component({
  selector: 'button[shared-ui-button]',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './button.component.html',
  host: {
    '[attr.disabled]': 'disabled ? true : null',
    '[class]': 'buttonClasses',
  },
})
export class ButtonComponent {
  @Input() size: 'small' | 'medium' = 'medium';
  @Input() style: 'primary' | 'secondary' | 'tertiary' = 'primary';
  @Input() disabled = false;

  get buttonClasses(): string {
    return `shared-ui-button shared-ui-button--${this.style} shared-ui-button--${this.size}`;
  }
}
```

Figure 14: The .ts-file of the button component, containing the class and @Component decorator with configs.

An example of Atomic Design principles can be seen in Figure 15 and 16, where the search form molecule contains an input atom and a button atom, showcasing the reusability of the atom components.

Figure 15: Screenshot of the search form molecule.

```html
<form class="search-form" (submit)="onSubmit($event)">
    <input shared-ui-input [placeholder]="placeholderText" />
    <button type="submit" shared-ui-button [disabled]="disabled">
      {{ buttonText }}
    </button>
</form>
```

Figure 16: The template of the search form component contains an input atom and a button atom.

6.3 Stylesheet and Theme Implementation

The styling was implemented using Sass. The component library features an optional base styling, customizable through CSS variables and can be included in the application by adding it to the `styles` array in the application's `angular.json` configuration file. By not importing the base styling, the component library can also be used in a headless manner, without any predefined styles. The styling was separated into different files using Sass partials, to create a well-organized and maintainable architecture. Each application has its own dedicated styling directory, located within its respective application directory.

For the base theme, a set of CSS variables was defined to manage design tokens such as colors, spacing, typography, borders and more. Figure 17 displays the beginning of the variable file, showcasing the color variables.

```
:root {
  // Colors
  --color-primary: #377477;
  --color-secondary: #70b3bf;
  --color-tertiary: #377477;

  --color-background-primary: #FFFFF;
  --color-background-secondary: #f1f1f1;
  --color-background-input: var(--color-background-secondary);

  --color-text-primary: #272c2c;
  --color-text-secondary: #555555;
  --color-text-inverse: #fdfdfd;

  --color-error: #e74c3c;

  --color-border: var(--color-primary);

  --color-button-text: #ffffff;
  --color-button-disabled-bg: #d6d6d6;
  --color-button-disabled-text: #9e9e9e;
  --color-label-disabled: #757575;
```

Figure 17: CSS variables for the base theme.

These base variables can be selectively overridden to create a customer-specific theme. An example of this can be seen in Figure 18, where the configuration for Customer 1 overrides some of the default styles.

```
:root {
  // Colors
  --color-primary: #4A90E2;
  --color-secondary: #F5A623;
  --color-tertiary: #7B92A3;

  --color-background-primary: #F4F4F9;
  --color-background-secondary: #E1E8ED;

  --color-error: #D9534F;

  --color-button-text: var(--color-text-inverse);
  --color-button-disabled-bg: #D6D6D6;
  --color-button-disabled-text: #9E9E9E;
  --color-label-disabled: #A1A1A1;

  // Typeface
  --font-family-base: 'Arial', sans-serif;
```

Figure 18: The theme configuration of Customer 1 overrides some of the base style variables.

The base styles can also be overridden by creating new CSS rules, if the theme variables do not fully meet the customer's customization needs. Figure 19 showcases the different customer themes applied to the search form component, as well as the base style.

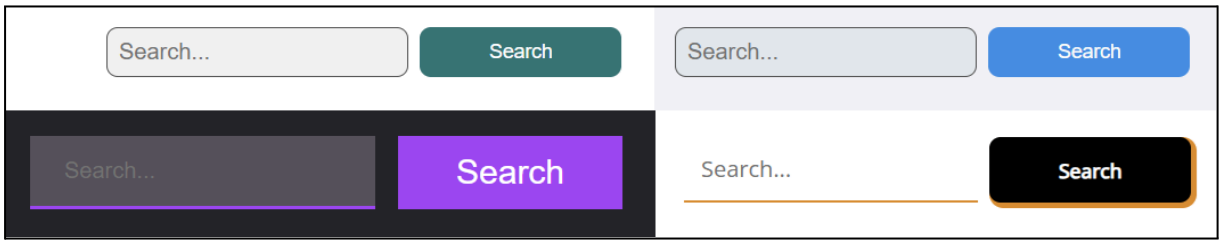

Figure 19: The search form component with different customer themes, with the base style on the top left side.

6.4 Storybook Implementation

The Storybook setup consists of two top-level categories: Design Tokens and Components. The Component category is further divided into two subcategories—Atoms and Molecules—to represent the components hierarchy based on the Atomic Design methodology. Additionally, a standalone introduction page, written in MDX, was configured to be the first page displayed when visiting the Storybook, as can be seen in Figure 20. This page contains general info about the project and guidance on how to navigate the Storybook. The project also utilizes Storybook's *addon-docs* Autodoc feature, which automatically generates documentation for all story files.

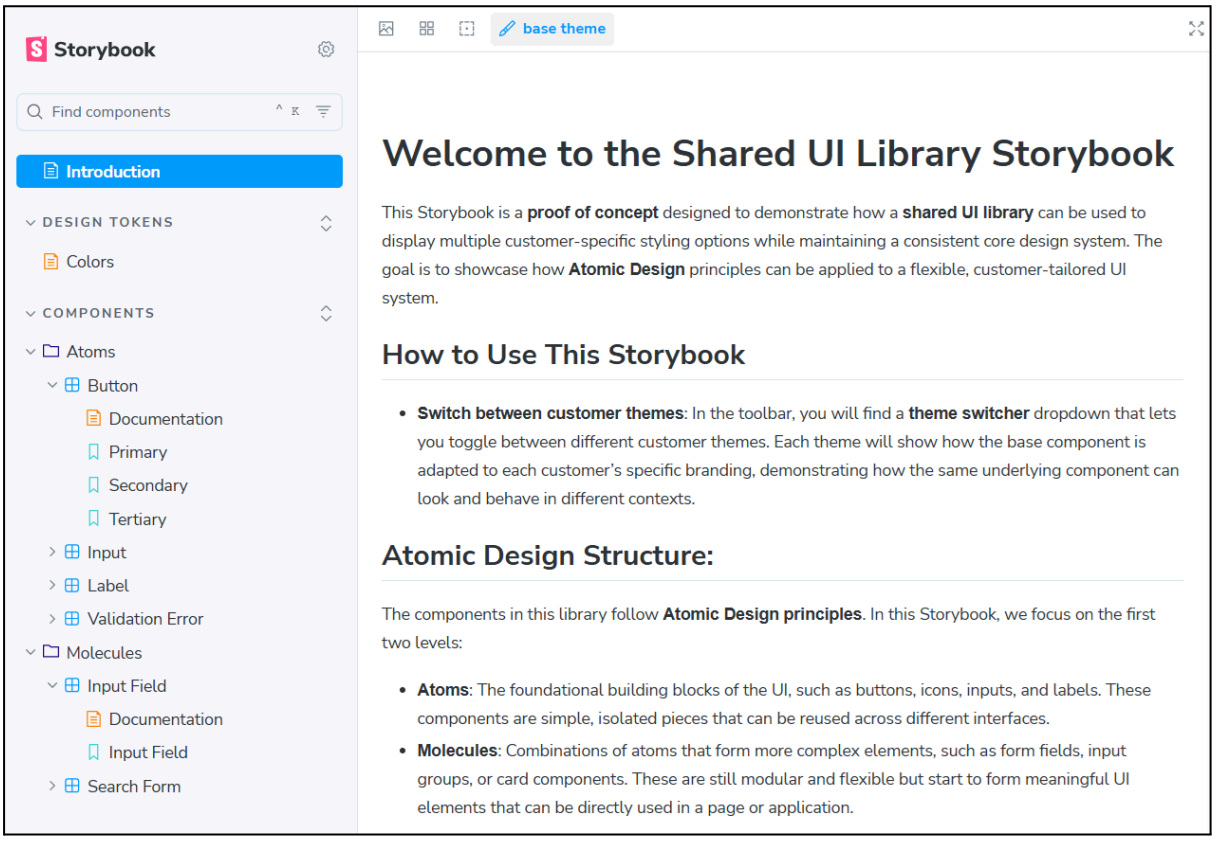

Figure 20: Screenshot of the implemented Storybook instance.

Each component in the library has its own story file, located in the same folder as the Angular component, following standard structure. Most of the story files contain multiple stories to showcase different variations of the components. An example of this can be seen in Figure

21, where the documentation lists the available story variants—primary, secondary and tertiary—of the Button component.

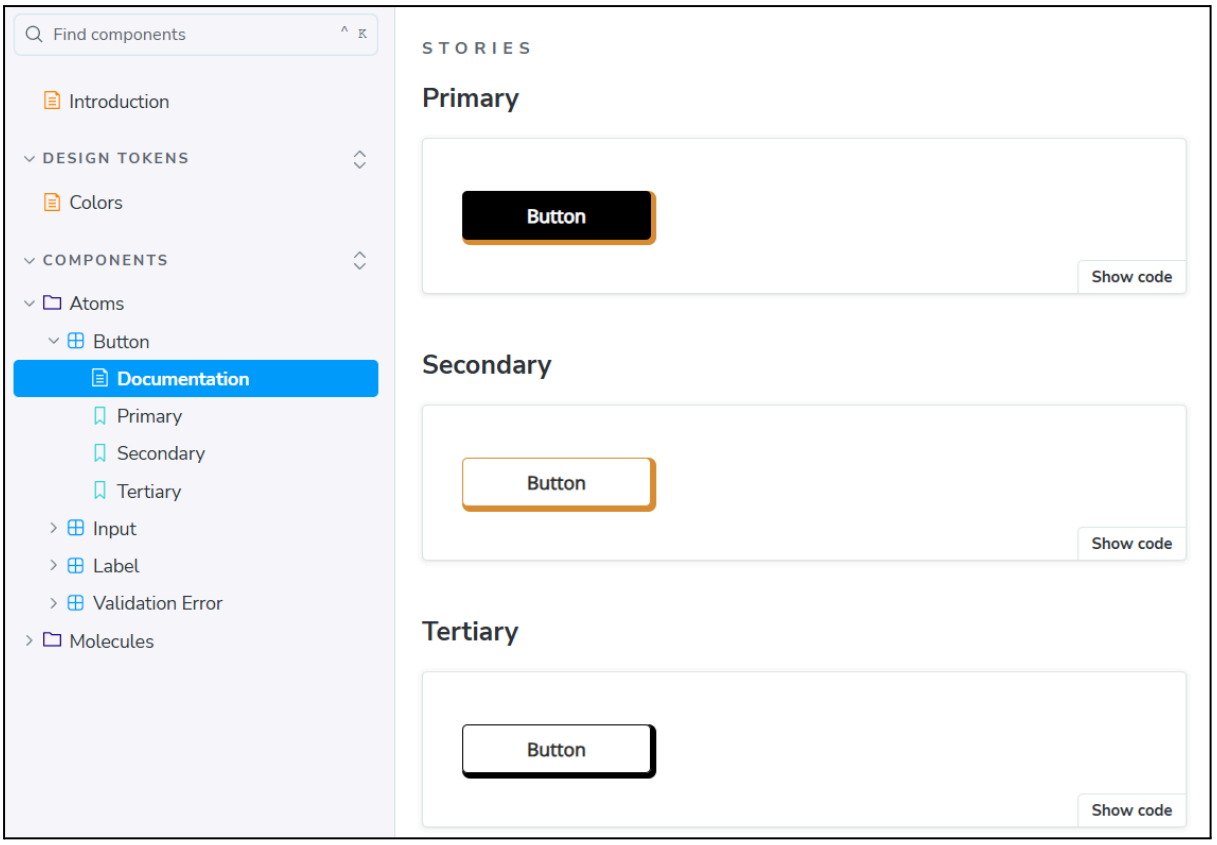

Figure 21: Storybook Documentation page for the Button stories, showcasing different variations of the Button component.

The Storybook controls interface allows users to display the stories in different states and additional variants, as seen in Figure 22.

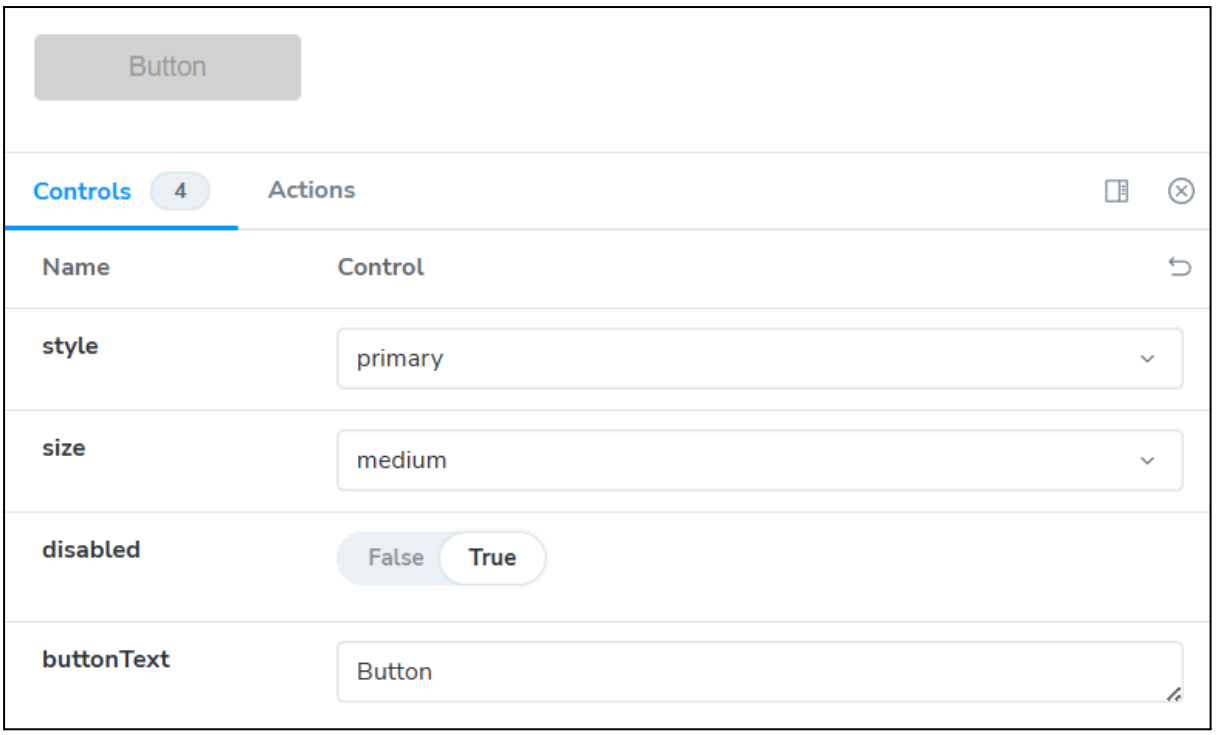

Figure 22: The Primary Button story and its associated controls, allowing users to modify the button's variant and state.

6.4.1 Extended Arguments

While not part of the initial requirements, we also explored the possibility of allowing stories to accept arguments that are not tied to the predefined inputs of the Angular component they represent. This was successfully achieved by extending the args with an additional interface, which allowed for additional inputs to the story, as seen in Figure 23. This example allows users to input text in the controls panel, which is then dynamically displayed as a text string in the inner HTML of the component.

```ts
import { Meta, StoryObj } from '@storybook/angular';
import { LabelComponent } from './label.component';

// Define extra args for the story (label is not an @Input in LabelComponent)
interface ExtraArgs {
  labelText: string;
}

const meta: Meta<LabelComponent & ExtraArgs> = {
  component: LabelComponent,
  argTypes: {
    disabled: { control: 'boolean' },
    labelText: {
      control: 'text',
      description:
        'Text that will be displayed inside the label element',
    },
  },
};

export default meta;
type Story = StoryObj<LabelComponent & ExtraArgs>;

export const Label: Story = {
  render: (args) => ({
    props: args,
    template: `<label shared-ui-label for="idOfInputHere" [disabled]="disabled">
    {{ labelText }} </label>`,
  }),
  args: {
    disabled: false,
    labelText: 'Label text',
  },
};
```

Figure 23: Code example of a story with extended args.

6.4.2 Theme Switcher Implementation

To handle the theme selection and presentation of the different customer themes, two different implementations were created:

1. **addon-themes:** This addon dynamically applies a class corresponding to the currently selected theme to the HTML element of the story's iframe. The theme classes are

specified in a single `styles.scss` file that imports the customer specific themes into their respective classes.

2. **Custom decorator:** This approach enables a more structured way to handle themes by dynamically switching between different stylesheet files. The stylesheet file of the selected theme is injected to the iframe using a `<link rel="stylesheet">` tag. It allows CSS variables at `:root`, as it lets the different themes exist in completely separate files, enabling a more standard styling implementation. The Sass files are compiled into CSS using a command executed via the CLI[9], the scripts can be observed in Figure 24.

```
"sass-all": {
  "executor": "nx:run-commands",
  "options": {
    "commands": [
      "sass apps/customer1/src/styles/styles.scss compiled-css/customer-1.css",
      "sass apps/customer2/src/styles/styles.scss compiled-css/customer-2.css",
      "sass apps/customer3/src/styles/styles.scss compiled-css/customer-3.css",
      "sass libs/shared-ui/src/lib/styles.scss compiled-css/base.css"
    ],
    "parallel": true
  }
},
"sass-watch-all": {
  "executor": "nx:run-commands",
  "options": {
    "commands": [
      "sass --watch apps/customer1/src/styles/styles.scss compiled-css/customer-1.css",
      "sass --watch apps/customer2/src/styles/styles.scss compiled-css/customer-2.css",
      "sass --watch apps/customer3/src/styles/styles.scss compiled-css/customer-3.css",
      "sass --watch libs/shared-ui/src/lib/styles.scss compiled-css/base.css"
    ],
    "parallel": true
  }
},
```

Figure 24: Custom scripts for compiling Sass to CSS files executed by CLI. The "sass-all" script generates a static build intended for deployment, while "sass-watch-all" monitors file changes and dynamically updates the build during development.

[9] Command-line interface

Both implementations have the same functionality: the theme switcher is configured to display the base styling as the default theme, and users can change the theme by selecting an option from the dropdown menu located in the toolbar. The theme dropdown from our custom solution is shown in Figure 25.

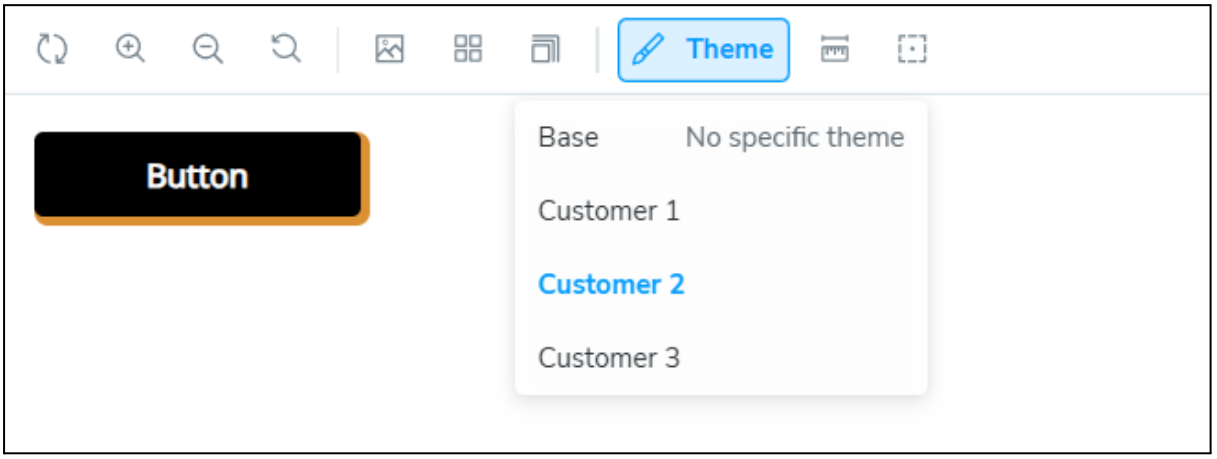

Figure 25: The dropdown menu for selecting the theme.

To accommodate the design tokens of different customers, the content of the documentation pages updates dynamically based on the selected customer theme. Because Storybook's default documentation format (MDX) does not support dynamic content, an alternative solution was implemented using Angular components. The component receives the current theme value as an input from the story, which in turn receives the current theme from the global Storybook context. Based on the theme value, the component renders the appropriate child component containing the documentation specific to the selected customer.

6.4.3 Deployment

The Storybook application was successfully deployed on the cloud-based platform Netlify. The deployment process involved generating a static build executed through the CLI[10]. Once the build was complete, the resulting dist folder was simply dragged and dropped into the Netlify dashboard for a quick and easy deployment as seen in Figure 26.

[10] Command-line interface

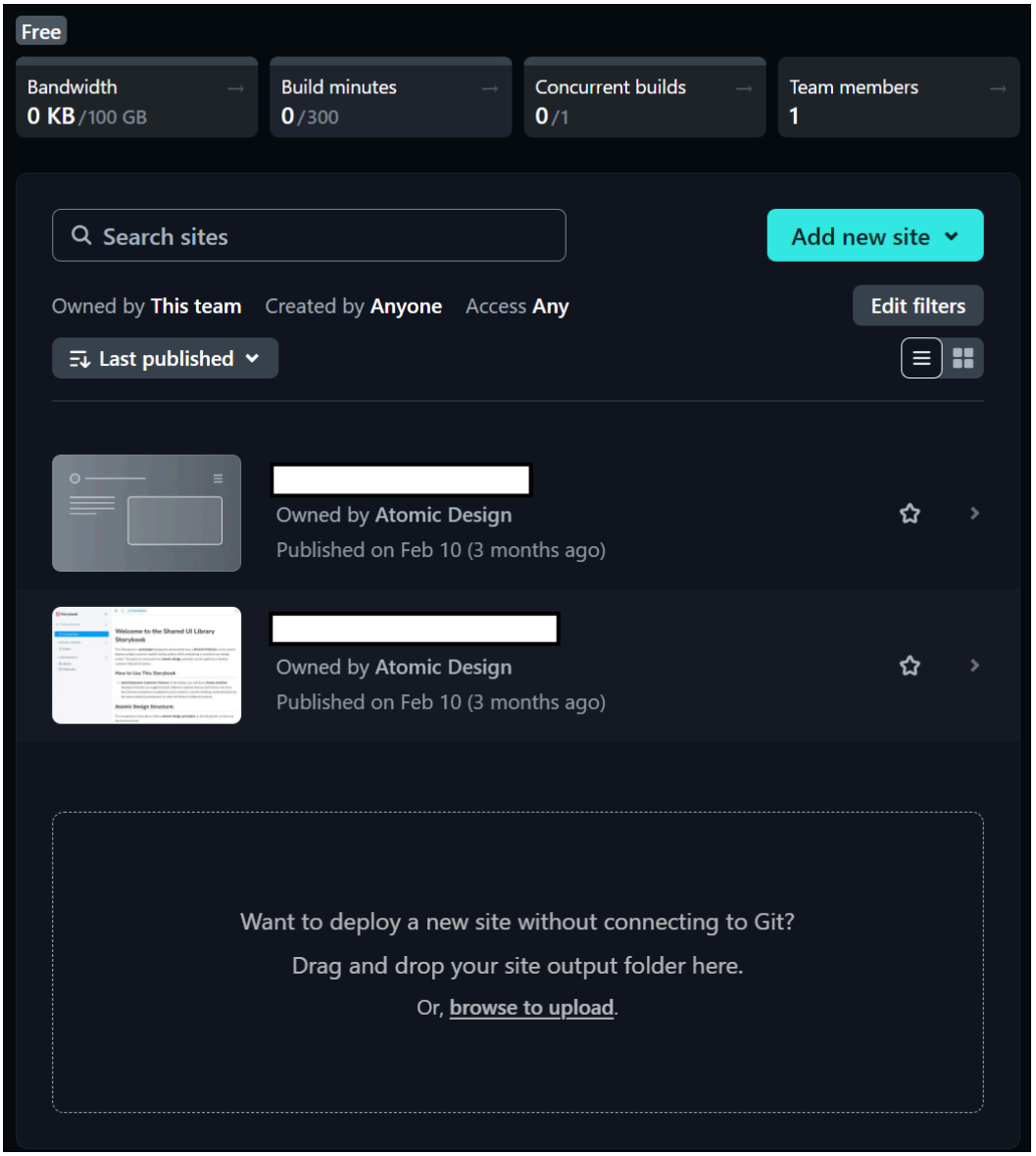

Figure 26: Netlify drag and drop deployment dashboard

6.5 Component Library Write Access

As a result of a discussion during one of the Agile meetings, time was allocated to investigate the feasibility of locking front-end component libraries to prevent unauthorized changes to critical components. The initial consideration was to implement access control at the directory level within Bitbucket. However, this was quickly ruled out, as Bitbucket does not natively support access control on directory level. Two alternative solutions were identified:

1. **Git commit hook:** If the component library should exist within the same project, a Git commit hook can be configured to restrict write access at the file or directory level by verifying whether the user is included in a list of approved users.
2. **Separate repository:** Alternatively, if it is acceptable that the component library exists in a separate project, it can be placed in its own repository. This allows for write access restrictions to be applied at the repository level in Bitbucket. It would also offer the possibility to assign specific individuals as code reviewers, allowing other teams to suggest changes to the code while ensuring that changes are only merged with approval from the designated reviewers.

6.6 Identified Limitations of the Chosen Tech Stack

A crucial aspect of creating this proof-of-concept application was to identify the potential limitations of the chosen tech stack when attempting to meet the project requirements. This was done in the hopes of providing useful insights for future iterations or alternative solutions for a full-scale implementation. The following limitations were identified:

- Storybook's use of MDX does not support dynamic documentation without the use of React, which means it is unable to show different content based on the selected customer theme. The alternative solution of using an Angular component does not allow for the usage of Storybook code blocks.
- Theme styling leaks into the documentation content and the documentation interface of *addon-docs.*

7. CONCLUSION

This chapter summarizes the key outcomes of the project, reflects on them in relation to the research questions, and outlines potential directions for future work. By revisiting the initial objectives and the chosen methods and technology, we assess how effectively they were addressed throughout the course of the project.

7.1 Result

The result of this thesis became a successfully implemented proof of concept, consisting of a configured themable Storybook instance that accommodates three different mocked customers along with a base styling mode. We were able to fulfill the initial requirements, as well as additional requirements that were introduced later in the process. Limitations were also identified both in the project implementation and within the Atomic Design methodology itself.

In the Introduction chapter, we outlined two core research questions to guide the thesis. The first question—*How can a shared component library focused on reusability be implemented to accommodate the customization needs of different customers?*—has largely been answered through the implementation itself, which demonstrates a viable approach to building reusable components adaptable to customer-specific needs. The second question—*What are the benefits if using Atomic Design in this type of project*—is more difficult to answer. While the theoretical benefits of Atomic Design have been discussed, identifying project-specific, empirically verifiable benefits proved challenging. It could be argued that the project did not provide sufficient opportunity to fully explore the benefits of Atomic Design. The requirement was to implement atoms and molecules, which did demonstrate some level of reusability, but not enough to showcase the full potential of the methodology.

As Brad Frost notes, Atomic Design is best applied to established products, which was not present in our case. Additionally, Atomic Design relies heavily on the implementation of a design system, which was outside of the project's requirements and arguable beyond our

responsibility as developers. Understandably, us implementing such a project would likely not have provided significant value from a company perspective. The limited scope and timeframe constrained our ability to evaluate Atomic Design's long-term impact. A broader study would be needed to draw stronger conclusions, ideally one focused on qualitative evaluation and long-term observations.

There are several areas we would have liked to explore and refine further, had the project's allocated time frame allowed for it:

- Assets, such as images or icons, could be provided to the customer themes. The same component could dynamically use different icons depending on the chosen customer.
- Although CSS best practices were considered, they were not consistently prioritized, resulting in some minor inconsistencies in class and variable naming.
- The Atomic Design integration could have been taken further by applying atomic naming convention to both components and CSS and Sass naming.
- Since the project initially intended to keep completely separate styling for each customer, the addition of a shared base style was implemented in a rush. As a result, the current setup could benefit from refining the exposed variables of the base theme.
- Extracting Storybook into a standalone application could contribute to a cleaner architecture and a clearer separation of concerns.
- Choice of priorities and our limited experience of the organization's deployment tech stack, we did not implement a continuous deployment within the CI/CD[11] pipeline. For future development, such a pipeline should be implemented, and automated tests should be incorporated into the process.

In retrospect, more thorough planning at the beginning of the project could have led to a stronger outcome, particularly if the goal was to align more closely with Brad Frost's recommendations for implementing Atomic Design. Following Frost's vision, it might have been more valuable to shift the objective toward developing a pilot project that could serve as the foundation for a future design system.

[11] Continuous integration/ Continuous delivery or deployment

That said, we believe the project still holds value as a proof of concept, especially in demonstrating the technical feasibility of managing multiple contexts within a single Storybook instance. However, to fully explore and benefit from the Atomic Design methodology, the development of a complete and integrated design system would have been essential. As it stands, it feels as though we have only scratched the surface of what Atomic Design has to offer.

7.2 Methods and Technologies

We are satisfied with the methods and the workflow we employed throughout the project. The processes were pragmatic and realistic and we upheld a high standard of professionalism throughout, despite the considerable freedom in how we could structure our work and development choices.

The combined use of Angular, Storybook, Sass, CSS and Nx provided a solid foundation for scalable and modular front-end development. However, some integration challenges were encountered, particularly between Angular and Storybook. As previously mentioned, Storybook and the addons used did not fully support our specific use cases, requiring creative workarounds. This raises the question whether alternative frameworks, such as React—which enjoys broader integration support within Storybook—could have offered a smoother development experience.

7.3 Atomic Design: Thoughts and Takeaways

From the perspective of a new developer, Atomic Design might initially appear to be solely about creating thoughtful reusable components. However, a significant portion of the book revolves around building and maintaining a design system. The Atomic Design methodology builds on several pre-existing ideas, such as modularity, the single responsibility principle, the DRY[12] principle and component-based architecture. What sets it apart is its clear hierarchical structure, which enhances reusability, as well as its emphasis on design systems,

[12] Don’t repeat yourself

seamlessly integrated with the methodology. Additionally, it utilizes a shared language that makes it easier for designers and developers to collaborate.

Finding and comparing alternative methodologies to Atomic Design is a challenging task. Atomic Design blends design system philosophy, component-based development and naming conventions in a way that is fairly unique. There are methods for structuring code—some framework-specific, some not—but none feel directly comparable to Atomic Design. This is likely due to the methodology's broad scope, which makes it difficult to find a perfect substitute to the methodology.

It feels important to reiterate that implementation of Frost's Atomic Design methodology—along with its associated assets such as a design system, pattern libraries, style guide, component library, naming standards and so on—requires a *lot* of resources. Following the initial setup, it is ideal to establish a dedicated team for the ongoing maintenance of the system, also making it a long-term resource investment for the organization.

In Frost's book, he is an enthusiastic advocate for pursuing the work of implementing a design system, even without organization support. Because of the long term commitments that an implemented design system requires, it feels like a too idealistic dream that—in the worst case—can become a total waste of time for the company if no one takes responsibility for maintaining the system after its creation. In this regard, it is somewhat misguided for Frost to encourage this approach, especially if his intention is for us to build a lasting, sustainable system that will not become obsolete the moment development is done. While his book contains inspiring sentiments—such as "ask forgiveness not permissions"—this attitude can ultimately lead to challenges and headaches in the long run if the issue of maintenance is not properly addressed.

During development of both the thesis and the project, we gathered insights into the methodology, but were unable to find references that fully addressed some of the limitations we observed. These limitations regarding Atomic Design are discussed in the following paragraphs.

Firstly, it is worth questioning whether the methodology fully holds up in modern UI development. While it provides a useful framework for structuring components, it may be too simplistic to account for the complexities of the dynamic, interactive and highly responsive user interfaces of today. It works well for simpler or static components, but does not fully address the challenges of dynamic component composition and highly responsive structures that vary significantly between viewport sizes.

While the introduction chapter of the book acknowledges these challenges, it does not provide what we consider a sufficient satisfactory solution. Creating reusable components with the hierarchical Atomic Design structure addresses part of the problem, but does not fully resolve it. From our perspective as developers, this is also a coding challenge that must be addressed at the implementation level. The issue feels unsatisfactory because of another limitation of the book; it does not provide any clear guidelines on how to structure the code or approach the actual coding process. Which—to be fair— was likely an intentional choice by Frost to keep the methodology framework-neutral, but it still felt like a drawback in practical application.

Additionally, we observed some ambiguity in component classification, which can lead to confusion and inconsistency in component organization. The problem may partly stem from the methodolgy's age, an issue already addressed by Jankowski (2024) which was discussed in a previous chapter. However, it may also represent an inherent challenge that will persist, regardless of updates to the methodology, as individuals often interpret information differently. But ultimately, we perceive that there is potential for improvement in clarifying the classification between component stages.

It is also important to remember that Atomic Design ultimately is a methodology for organizing UI components within a design system. While the resulting structure can be used to inform architectural decisions for the code base, it is worth considering whether it could be counterproductive to translate the structure too literally into code. As mentioned in the Theory chapter, Lanciaux (2021) raises an important concern, although not directly tied to

Atomic Design specifically: Over-abstraction can lead to a cluttered and overly fragmented code base, especially if it results in abstractions that will not be reused or that only add unnecessary complexity. Arguably, this is not a flaw of the Atomic Design methodology itself, but rather a matter of it requiring thoughtful consideration of how and where it should be applied.

Despite these shortcomings, we are of the opinion that Atomic Design still can be a valuable resource in modern front-end development. However—to iterate—the successful implementation of the Atomic Design methodology requires a well-considered and deliberate effort. It is not merely a matter of breaking down UI elements into smaller parts, but also involves designing a whole system that establishes clear guidelines, maintains consistent naming conventions and fosters cross-disciplinary collaboration between designers and developers. Without thoughtful planning and a shared understanding of its principles, the methodology risks becoming fragmented and underutilized, ultimately limiting its effectiveness.

7.4 Closing Words

Reflecting on the project, one key insight that emerged was the central role of design systems in Atomic Design. Initially, our focus was primarily on understanding and applying the Atomic Design methodology, without fully considering its broader context. However, as the thesis progressed, the importance of design systems became increasingly clear. Although building a full design system was out scope for this project, exploring the theory proved to be valuable. It gave us a foundation to work from when evaluating the Atomic Design methodology and helped us make key decisions about what the Storybook should contain. For example, the inclusion of design tokens was a choice influenced by our understanding of design system principles. This insight might also be useful for us as developers in future work. It is easy to implement solutions without thinking about how it fits into the overarching design system, but doing so risks inconsistency and inefficiency.

There is potential for future development work to improve this project, some of which we have already discussed earlier in the chapter 7.1 Result. Looking ahead, if this project ever progresses beyond the proof of concept stage, we recommend exploring different theming strategies to create a more robust and flexible theming system. Another direction forward could be to make the component library framework agnostic by using technologies like Web Components, which would allow for reuse across different front-end frameworks and enhance the library's flexibility and long-term value.

We also recommend conducting a confidentiality assessment for the themable Storybook implementation. Currently, showcasing multiple customers within a single Storybook limits its use to intel purposes only, as it cannot be shared with individual customers without exposing themes and configurations meant for other clients, which could potentially violate confidentiality. If we want to align with Frost's approach to documentation, a solution that allows showcasing all customers without compromising confidentiality would be ideal. Overall, maintaining separate Storybooks for customer-facing documentation would be a more secure and appropriate solution.

Working solely with atoms and molecules limited our ability to fully experience the benefits of Atomic Design, reinforcing the idea that a true full-scale implementation requires a broader design system context. It is an "infinite project" in many ways—always evolving and continuously shaped by the needs of the product and the teams working on it.

Given the number of technologies and the project's time constraints, we focused primarily on delivering core functionality, with limited opportunity to explore each technology in depth. Additionally, we brought different levels of front-end development experience to the project; one developer primarily works with Java, while the other has a stronger background in Angular. This created individual challenges, but also created opportunities for mutual learning and complementary perspectives. Despite the challenges, we are grateful for the chance to work with such a diverse range of technologies and we look forward to applying the knowledge gained in future projects.

REFERENCE LIST

Abramov, D. (2015, March 23). *Presentational and Container Components*. Medium. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

Ålandsbanken. (2024, June 13). *Crosskey Banking Solutions Ab Ltd*. Ålandsbanken; #creator. https://www.alandsbanken.fi/sv/om-oss/fakta-om-alandsbanken/koncernen/crosskey-banking-solutions-ab-ltd

Angular. (n.d.-a). *Components*. Angular. Retrieved May 1, 2025, from https://angular.dev/essentials/components

Angular. (n.d.-b). *Dependency Injection*. Angular. Retrieved May 1, 2025, from https://angular.dev/essentials/dependency-injection

Angular. (n.d.-c). *What is Angular? (v19)*. Angular. Retrieved January 7, 2025, from https://angular.dev/overview

Anne, J. (2015, October 13). *The Salesforce Team Model for Scaling a Design System*. Salesforce Designer. https://medium.com/salesforce-ux/the-salesforce-team-model-for-scaling-a-design-system-d89c2a2d404b

Atlassian. (n.d.-a). *Bitbucket Overview*. Bitbucket. Retrieved February 28, 2025, from https://bitbucket.org/product/guides/getting-started/overview#a-brief-overview-of-bitbucket

Atlassian. (n.d.-b). *Design tokens explained*. Atlassian Design System. Retrieved May 18, 2025, from https://atlassian.design/tokens/design-tokens

Atlassian. (2025, January 16). *What is Jira? | Atlassian Answered* [Video]. YouTube. https://www.youtube.com/watch?v=Z-a1RB9HvDI&t=6s&ab_channel=Atlassian

Bergman, C. (2024, February 12). *What Is a Design System*. Figma. https://www.figma.com/blog/design-systems-101-what-is-a-design-system/

Cambridge University Press. (n.d.). *modularity*. Cambridge Dictionary. Retrieved January 18, 2025,

from https://dictionary.cambridge.org/dictionary/english/modularity

Crosskey Banking Solutions. (2024, January 1). *Crosskey Banking Solutions*. https://www.crosskey.fi/

Fessenden, T. (2021, April 11). *Design Systems 101*. Nielsen Norman Group. https://www.nngroup.com/articles/design-systems-101/

Figma. (2023, May 24). *Welcome to design systems - Lesson 1 : Introduction to design systems* [Video]. YouTube. https://www.youtube.com/watch?v=YLo6g58vUm0

Frost, B. (2016). *Atomic design*. Brad Frost.

Frost, B. (2019a, June 16). *The Technical Side of Design Systems* [Video]. YouTube. https://www.youtube.com/watch?v=TgWyyoofKIA

Frost, B. (2019b, July 10). *Extending Atomic Design*. Brad Frost. https://bradfrost.com/blog/post/extending-atomic-design/

Frost, B. (2024, January 9). *A Global Design System*. Brad Frost. https://bradfrost.com/blog/post/a-global-design-system/

GeeksforGeeks. (2018, April 6). *What is User Interface (UI) Design?* GeeksforGeeks. https://www.geeksforgeeks.org/user-interface-ui/

GeeksforGeeks. (2024a, January 8). *MOCK (Introduction) - Software Engineering*. GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering-mock-introduction/

GeeksforGeeks. (2024b, January 17). *Responsive Web Design*. GeeksforGeeks. https://www.geeksforgeeks.org/responsive-web-design/

GeeksforGeeks. (2024c, January 28). *What is Graphical User Interface (GUI)?* GeeksforGeeks. https://www.geeksforgeeks.org/what-is-graphical-user-interface/

GeeksforGeeks. (2024d, June 28). *Component-Based Architecture - System Design*. GeeksforGeeks. https://www.geeksforgeeks.org/component-based-architecture-system-design/

Github. (n.d.). *About GitHub and Git*. Github. Retrieved January 7, 2025, from https://docs.github.com/en/get-started/start-your-journey/about-github-and-git

Godbolt, M. (2016). *Frontend Architecture for Design Systems: A Modern Blueprint for Scalable and*

Sustainable Websites. “O’Reilly Media, Inc.”

Griffith, C. (2019, February 6). *What is a UI Component Library - Modern Frontend Development*. Ionic. https://ionic.io/resources/articles/what-is-a-ui-component-library

Jankowski, M. M. (2024, May 6). *Why so many Design Systems using Atomic Design fail to succeed?* Medium. https://medium.com/@michalmjankowski/why-so-many-design-systems-using-atomic-design-fail-to-succeed-07c01b7451e9

Karoulla, A. (2024, August 13). *What is a Component Library? When to Build Your Own and When to Use Someone Else’s*. freeCodeCamp.org. https://www.freecodecamp.org/news/what-is-a-component-library-when-to-build-your-own/

Kramer, N. (2024, April 10). *What is modular programming?* https://daily.dev/blog/what-is-modular-programming

Lanciaux, R. (2021). *Modern Front-end Architecture* (1st digital ed.). Apress.

Levin, Y. (2025, January 3). *Atomic Design and Its Relevance in Frontend in 2025*. DEV Community. https://dev.to/m_midas/atomic-design-and-its-relevance-in-frontend-in-2025-32e9

Martin, A. (2025, February 19). *Component-Based Design: Complete Implementation Guide*. Studio by UXPin; UXPin. https://www.uxpin.com/studio/blog/component-based-design-complete-implementation-guide/

Martinez, B. (2023, November 13). *Dissecting the Disadvantages of Atomic Design in UI Design*. Medium. https://medium.com/@martinezhillaryfrosh/dissecting-the-disadvantages-of-atomic-design-in-ui-design-8247e1c33f6f

MDN Web Docs. (2024, December 19). *Using CSS custom properties (variables)*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/CSS/Using_CSS_custom_properties

MDN Web Docs. (2025, January 24). *Introducing the CSS Cascade*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade

MUI. (n.d.). *Base UI: Unstyled React components and low-level hooks*. Retrieved April 27, 2025, from https://mui.com/base-ui/

Netlify. (2024, October 2). *What is Netlify?* Netlify Docs. https://docs.netlify.com/platform/what-is-netlify/#composable-web-architecture

Nx. (n.d.-a). *Generate Code*. Nx. Retrieved April 26, 2025, from https://nx.dev/features/generate-code

Nx. (n.d.-b). *Intro to Nx*. Nx. Retrieved January 1, 2025, from https://nx.dev/getting-started/intro

Nx. (n.d.-c). *Mental Model*. Nx. Retrieved April 26, 2025, from https://nx.dev/concepts/mental-model

Nx. (n.d.-d). *Monorepos*. Nx. Retrieved April 13, 2025, from https://nx.dev/concepts/decisions/why-monorepos

Nx. (n.d.-e). *Why Nx?* Nx. Retrieved January 1, 2025, from https://nx.dev/getting-started/why-nx

Rafalski, K. (2025, January 8). *What is Netlify? The Ultimate Guide for Web Developers*. https://www.netguru.com/blog/what-is-netlify

Ramsier, T. (2019, February 5). *Separation of Concerns: Component Deconstruction - Dictionary Engineering - Medium*. Dictionary Engineering. https://medium.com/dictionary-engineering/separation-of-concerns-component-deconstruction-c08f9dc057b

Sass. (n.d.-a). *Sass Basics*. Retrieved February 4, 2025, from https://sass-lang.com/guide/

Sass. (n.d.-b). *Variables*. Retrieved January 25, 2025, from https://sass-lang.com/documentation/variables/

Serban, P. (2023, August 27). *Advanced Tactics in the CSS Specificity Wars*. Paul Serban | Software Engineer. https://www.paulserban.eu/blog/post/advanced-tactics-in-the-css-specificity-wars/

Storybook. (n.d.-a). *Automatic documentation and*. Storybook. Retrieved February 3, 2025, from https://storybook.js.org/docs/writing-docs/autodocs

Storybook. (n.d.-b). *Decorators*. Storybook. Retrieved January 18, 2025, from https://storybook.js.org/docs/writing-stories/decorators

Storybook. (n.d.-c). *Doc blocks*. Storybook. Retrieved February 3, 2025, from

https://storybook.js.org/docs/writing-docs/doc-blocks

Storybook. (n.d.-d). *MDX*. Storybook. Retrieved February 3, 2025, from https://storybook.js.org/docs/writing-docs/mdx

Storybook. (n.d.-e). *Storybook Addons*. Storybook. Retrieved January 18, 2025, from https://storybook.js.org/docs/configure/user-interface/storybook-addons

Storybook. (n.d.-f). *Storybook: Build UIs without the grunt work*. Storybook. Retrieved January 9, 2025, from https://storybook.js.org/

Storybook. (n.d.-g). *Storybook: Framework support*. Storybook. Retrieved January 13, 2025, from https://storybook.js.org/docs/configure/integration/frameworks

Storybook. (n.d.-h). *Storybook: Get started with Storybook*. Storybook. Retrieved January 9, 2025, from https://storybook.js.org/docs

Storybook. (n.d.-i). *Story rendering*. Storybook. Retrieved January 27, 2025, from https://storybook.js.org/docs/configure/story-rendering

Storybook. (n.d.-j). *Toolbars & globals*. Storybook. Retrieved January 18, 2025, from https://storybook.js.org/docs/essentials/toolbars-and-globals

Storybook. (n.d.-k). *Types of addons*. Storybook. Retrieved January 18, 2025, from https://storybook.js.org/docs/addons/addon-types

Storybook. (n.d.-l). *What's a story?* Storybook. Retrieved January 26, 2025, from https://storybook.js.org/docs/get-started/whats-a-story

UXPin. (2024a, September 5). *What Are Design Tokens?* Studio by UXPin; UXPin. https://www.uxpin.com/studio/blog/what-are-design-tokens/

UXPin. (2024b, October 10). *How Storybook Helps Developers With Design Systems?* Studio by UXPin; UXPin. https://www.uxpin.com/studio/blog/how-storybook-helps-developers-with-design-systems/