Aalto University
School of Science

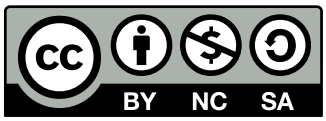
Master's Programme in Computer, Communication and Information Sciences

Impact of React component libraries on developer experience

An empirical study on component libraries' styling approaches

Ossian Rajala

Master's Thesis
2024



Aalto University
School of Science

Author Ossian Rajala

Title Impact of React component libraries on developer experience — An empirical study on component libraries' styling approaches

Degree programme Computer, Communication and Information Sciences

Major Computer Science

Supervisor University Lecturer Otto Seppälä

Advisor Doctoral Researcher Juho Vepsäläinen

Date July 23, 2024 **Number of pages** 76 **Language** English

Abstract

Over the past three decades, web design has transitioned from simple static pages to complex, interactive applications. Modern web development relies heavily on JavaScript frameworks and reusable components, often utilizing third-party, pre-styled component libraries. Customizing the styles - which are typically implemented with CSS (Cascading Style Sheets) - of such libraries to meet specific design requirements is a common task. This customization creates a need for effective styling solutions that enhance developer experience and maintainability.

Despite its widespread use in web development, CSS remains under-researched, particularly in the context of how it affects developers' day-to-day work with component libraries. As technologies continue to evolve, understanding and improving the developer experience becomes increasingly crucial, as it can positively impact software development outcomes.

This thesis seeks to understand how React component libraries' theming approaches impact customization, particularly ease of implementation, reusability of styles, and flexibility to accommodate various design requirements. Furthermore, the study investigates how the aforementioned factors and the library's documentation support or hinder the developer's work. These factors were investigated by porting the same application to four pre-styled component libraries. The implementation process, methods used, and challenges encountered were documented and analyzed.

As a result, the study found that meeting the design requirements was possible with all compared libraries: Ant Design, Chakra UI, shadcn/ui, and Mantine. However, the ease, intuitiveness, and maintainability of required solutions varied. Especially as one of the libraries, Ant Design, was tightly coupled with a specific design system. Color customization and dark and light modes were frequent sources of confusion. Overall, the libraries offered a positive experience and allowed the desired design to be reached without resorting to style hacks. Still, all the libraries would benefit from improved documentation, namely more detailed examples and explanations of the customization options.

Keywords component library, React, CSS, developer experience, Ant Design, Chakra UI, shadcn/ui, Mantine

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Ossian Rajala

Työn nimi React-komponenttikirjastojen vaikutus kehittäjäkokemukseen — Empiirinen tutkimus kirjastojen tyylittelyratkaisuista

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Tietotekniikka

Työn valvoja Yliopistonlehtori Otto Seppälä

Työn ohjaaja Tohtorikoulutettava Juho Vepsäläinen

Päivämäärä 23.7.2024 **Sivumäärä** 76 **Kieli** englanti

Tiivistelmä

Viimeisten kolmen vuosikymmenen aikana web-suunnittelu on siirtynyt yksinkertaisista staattisista sivuista monimutkaisiin, interaktiivisiin sovelluksiin. Moderni web-kehitys nojaa vahvasti JavaScript-kehyksiin ja uudelleenkäytettäviin komponentteihin hyödyntäen usein valmiiksi tyyliteltyjä kolmannen osapuolen komponenttikirjastoja. Usein näiden kirjastojen tyylejä, jotka toteutetaan yleensä CSS-tyylikielellä (Cascading Style Sheets), on tarpeen muokata tyylivaatimusten mukaisiksi. Tämä luo tarpeen tehokkaille tyyliratkaisuille, jotka parantavat kehittäjäkokemusta ja ylläpidettävyyttä.

Huolimatta sen keskeisestä roolista web-kehityksessä, CSS:ää tutkitaan edelleen vähän - etenkään sen vaikutusta kehittäjien päivittäiseen työhön komponenttikirjastojen parissa. Teknologioiden kehittyessä jatkuvasti on yhä tärkeäämpää ymmärtää ja parantaa kehittäjäkokemustä, sillä se voi vaikuttaa myönteisesti ohjelmistokehityksen lopputulokseen.

Tämä diplomityö pyrki selvittämään, miten React-komponenttikirjastojen tyylittelytavat vaikuttavat tyylien mukauttamiseen, erityisesti toteutuksen helppouteen, tyylien uudelleenkäytettävyyteen ja kykyyn mukautua erilaisiin suunnitteluvaatimuksiin. Lisäksi tutkimuksessa tarkastellaan sitä, miten edellä mainitut tekijät ja kirjaston dokumentaatio tukevat tai haittaavat kehittäjän työtä. Näitä tekijöitä tutkittiin muuntamalla sama sovellus käyttämään neljää valmiiksi tyyliteltyä komponenttikirjastoa. Toteutusprosessi, käytetyt menetelmät ja haasteet dokumentoitiin ja analysoitiin.

Tutkimuksen tuloksena havaittiin, että suunnitteluvaatimusten täyttäminen oli mahdollista kaikilla vertailluilla kirjastoilla: Ant Design, Chakra UI, shadcn/ui ja Mantine. Kuitenkin tarvittavien ratkaisujen helppous, intuitiivisuus ja ylläpidettävyys vaihtelivat. Erityisesti Ant Design, oli tiukasti sidottu tiettyyn suunnittelujärjestelmään (design system). Värien mukauttaminen sekä tumman ja vaalean tilan toteuttaminen aiheuttivat usein hämmennystä. Kokonaisuudessaan kirjastot tarjosivat kuitenkin positiivisen kehittäjäkokemuksen ja mahdollistivat halutun ulkoasun saavuttamisen ilman tyylikikkoja. Kaikki kirjastot hyötyisivät kuitenkin paremmasta dokumentaatiosta, erityisesti mukautusvaihtoehtojen yksityiskohtaisemmista kuvauksista ja esimerkeistä.

Avainsanat komponenttikirjasto, React, CSS, kehittäjäkokemus, Ant Design, Chakra UI, shadcn/ui, Mantine

Preface

I have received support and guidance from various people in my life throughout the process of writing this thesis - and even before this specific work ever began.

I want to thank my supervisor Otto Seppälä and my advisor Juho Vepsäläinen for their guidance, support, encouragement and especially patience throughout the thesis process. My belief in both the thesis itself and myself was at times wavering, but their support helped me to push through and finish the thesis.

Thank you to my project team at Gofore for their flexibility and understanding while I needed time off to work on this thesis. And while I resisted at times, in retrospect I am grateful for times they told me to stop working and focus on my thesis.

I also owe a big thank you to my family and friends for their support, encouragement, and help throughout the process.

Lastly I must thank my friend Lucky whose words stuck with me giving me the will to finally start and to also finish this thesis. Lucky, you made me see this project in a new light and I am grateful for that.

Turku, July 2024

Ossian Rajala

Contents

Abstract 3

Abstract (in Finnish) 4

Preface 5

Contents 6

1 Introduction 9
- 1.1 Changing challenges of web styling . . . 9
- 1.2 Motivation . . . 10
  - 1.2.1 Personal motivation . . . 11
- 1.3 Scope and research questions . . . 12
- 1.4 Structure of the thesis . . . 12

2 Developer experience 13
- 2.1 Prior research into developer experience of styling . . . 13
- 2.2 CSS challenges described in working diaries . . . 13
- 2.3 Resorting to CSS hacks . . . 14
- 2.4 Conclusion . . . 14

3 From patterns to design systems 15
- 3.1 Design systems . . . 15
- 3.2 Atomic Design . . . 16
- 3.3 Responsive design . . . 17
- 3.4 Conclusion . . . 18

4 CSS and types of UI libraries 19
- 4.1 CSS . . . 19
- 4.2 Types of UI libraries . . . 20
- 4.3 CSS frameworks and utility classes . . . 20
- 4.4 CSS Preprocessors . . . 21
- 4.5 Component libraries . . . 21
  - 4.5.1 Benefits of component libraries . . . 21
  - 4.5.2 Downsides of component libraries . . . 22
- 4.6 Conclusion . . . 22

5 React and styling React components 23
- 5.1 Why React was developed . . . 23
- 5.2 Philosophy of React . . . 23
- 5.3 Early usage of React . . . 23
- 5.4 Component-based architecture . . . 24
- 5.5 Introduction of JSX . . . 24
- 5.6 Styling React components . . . 25

5.6.1 Inline styles . . . 25
5.6.2 CSS Style sheets . . . 25
5.6.3 CSS Modules . . . 25
5.6.4 CSS-in-JS libraries . . . 26
5.7 Plain CSS in React components . . . 26
5.8 Styling component libraries . . . 26
5.8.1 Centralized theme definition . . . 27
5.9 Conclusion . . . 27

6 Implementation **29**
6.1 Author's developer background . . . 29
6.2 Library selection . . . 30
6.3 Overview of selected libraries . . . 36
6.3.1 Ant Design . . . 36
6.3.2 Chakra UI . . . 37
6.3.3 Mantine . . . 37
6.3.4 Shadcn/ui . . . 38
6.4 Reference application . . . 39
6.4.1 Questions page . . . 39
6.4.2 Candidate page . . . 39
6.5 Base implementation . . . 39
6.5.1 Project setup . . . 40
6.5.2 Matching reference application's styles . . . 40
6.5.3 Color palette . . . 40
6.5.4 Testing . . . 41
6.6 Setting up library versions . . . 41
6.6.1 Ant Design . . . 42
6.6.2 Chakra UI . . . 42
6.6.3 Mantine . . . 43
6.6.4 Shadcn/ui . . . 44
6.7 Conversion from base implementation . . . 45

7 Results **46**
7.1 Ant Design . . . 46
7.1.1 Limited component customizability . . . 46
7.1.2 Global design tokens hindering customizability and ease of implementation . . . 47
7.1.3 Customizability of color palette and dark mode . . . 47
7.1.4 Documentation . . . 49
7.2 Chakra UI . . . 50
7.2.1 The ease of implementation: Dark mode . . . 50
7.2.2 Ease of implementation: Customizing components . . . 52
7.2.3 Documentation . . . 52
7.3 Mantine . . . 53
7.3.1 Customizability: Color palette . . . 54

7.3.2 Customizability: Component customization 55
7.3.3 Discoverability of documentation 55
7.4 shadcn/ui 57
7.4.1 Customizability via code ownership 57
7.4.2 Ease of implementation 57
7.4.3 Documentation 58
7.5 Use of style sheets 59
7.6 Screenshot test results 60
7.7 Summary of results 60
7.7.1 Amount of CSS 60
7.7.2 Time spent on each library 61

8 Discussion **63**
8.1 Naming style and component properties 63
8.2 Automating colors 63
8.3 Differences in dark mode approaches 64
8.4 Semantic HTML 64
8.4.1 Headings 65
8.4.2 Changing rendered HTML element 65
8.5 Choosing between theming and component-level styling 66
8.6 Limitations 66
8.6.1 One developer's perspective 66
8.6.2 Simple application 66
8.6.3 Author's time management 67
8.6.4 Conclusion 67

9 Conclusion **68**
9.1 Main findings 68
9.2 Future work 69

References **70**

1 Introduction

For the past three decades, creating and implementing website designs and reusing website elements has been a constantly evolving yet challenging process. In the early days of the World Wide Web, or the web, web pages were simple and static and included minimal styling and interactivity [14]. Bare HTML (Hyper Text Markup Language) did not offer a way to reuse web page sections. The introduction of JavaScript in 1995 and CSS in 1996 drastically expanded the capabilities of front-end web development [36, 4]. Over time, JavaScript made it possible to turn static web pages into interactive applications by dynamically updating the page content without the user reloading it. Meanwhile, CSS expanded styling options for web pages and made it possible to separate styles from the content and reuse the same style definitions across multiple pages [4].

1.1 Changing challenges of web styling

Though CSS made it easier to style web pages, it also introduced a new set of challenges. The browser wars of the late 1990s and early 2000s created a turbulent environment, making it difficult for developers to write CSS that functioned consistently across different browsers [96]. Early versions of CSS also lacked many of the features that are now considered essential, such as flexbox and grid for layout control[1]. As a result, developers often had to resort to hacks and workarounds, like using tables for layout, to achieve the desired designs [96].

Standardization efforts by the World Wide Web Consortium (W3C) and the Web Standards Project (WaSP) helped to improve browser compatibility and promote the adoption of web standards [96]. Though browser compatibility is still a consideration, especially regarding newer CSS and HTML features, the situation has improved significantly. Standardization has reached such support that in 2013, WaSP announced that it had fulfilled its mission, and the group disbanded [59].

After browser wars, the next big challenge for front-end developers was the rise of mobile devices and the need to accommodate different screen sizes. While the earlier solution was to create separate mobile versions of websites, the introduction of responsive design in the early 2010s made it possible to create websites that adapt to different screen sizes.

The earlier static website layouts had already required designers to shift their thinking compared to the unchanging pixel-perfect print design some may have been used to. In contrast to print media, on the web the dimensions of the page is controlled by the user - their device and settings - and not the publisher [46, 97]. Responsive design was as big if not a bigger leap, requiring designers to consider how a website would look and function on a wide range of devices and screen sizes, where the

[1] CSS grid and flexbox are powerful tools for creating responsive and adaptive layouts. CSS Grid allows for two-dimensional layouts with intuitive control over rows and columns. It also enables responsive designs with fractional units, and the use of named template areas for simplified coding. On the other hand, flexbox is great for one-dimensional layouts, providing flexible item sizing, alignment, distribution, and the ability to reorder items without altering the HTML structure.

content could be rearranged and resized fluidly [45]. Instead of a single fixed layout, a website is an adapting collection of elements that move, resize, change appearance, and sometimes disappear entirely to accommodate different devices, screen sizes, and color scheme preferences.

Hence, in modern web development dominated by JavaScript frameworks, the building blocks of a web application are often reusable components. These components can be as simple as buttons, inputs, and text elements or as complex as modals, dropdowns, and data tables. Though companies and organizations do also develop their own components from scratch, using third-party component libraries - collections of components built by other developers and organizations - is common. But using a component library made by someone else is always, to some degree, a trade-off between effort and control.

Even though many libraries offer ways to customize the component styles, the ease, flexibility, and intuitiveness of customization vary between libraries. While the same result may be achievable with different libraries, the developers' experience of getting there can vary significantly. Developer experience, a concept akin to user experience, captures how developers think and feel about their work [26]. Improving developer experience can positively impact not only the developers themselves but also the outcomes of the software development projects they work on [33].

1.2 Motivation

JavaScript frameworks, such as React, Angular, and Vue, receive research attention, especially in the form of theses. But, the focus is rarely on styling or the use of component libraries. Even in works describing practical implementations, styling is mentioned either briefly in passing or not at all. This is unsurprising considering that CSS, the language at the heart of styling the web and thus also component libraries, has received relatively little attention in academic research. Marden and Munson [54] wrote already in 1999 that "style sheet languages are terribly under-researched". Other authors have echoed this sentiment [65]. Unfortunately, 25 years later, the situation has not changed.

A major portion of research into CSS has focused on refactoring tools and opportunities [57, 56, 64, 55, 34]. As a related topic, CSS code smells have been studied in the context of different frameworks [16] as well as to create predictive models [31]. CSS code smells are patterns in the code that are not bugs but indicate weaknesses in the design and may cause issues in program comprehension and maintenance in the long term [31]. Both refactoring and code smells relate to mistakes or improvement opportunities in the CSS code. However, the focus is on the code itself or the tools to manage it, but not on the experience of the developer who writes the CSS. As will be discussed in Chapter 2, the developer experience is a crucial aspect of software development, and styling is no exception.

Outside of academia, developers' fraught relationship with CSS is a well-known and discussed issue. While some of the discussion does not go beyond venting and memes such the classic "CSS is awesome" [22], plenty of developers are also trying to

help and educate others, for example, Rachel Andrew[2], Chris Coyier[3], Adam Argyle[4], Kevin Powell[5], and many others. They advocate for better CSS practices, tools, and education, and they try to demystify CSS for others.

Repairing developers' relationship with CSS is vital because, though some modern tools put distance between the developer and bare CSS, ultimately, as long as one is styling web applications in a browser, CSS cannot be avoided. More than 96 % of today's websites use CSS [85]. Though there's no shortcut to getting CSS right (meaning you need to learn it and resist the urge to slab `"!important"` in styles), well-designed tools can improve the developer experience and thus the maintainability of the application's styles.

If a component library is intended to be customized, the styling solution and component structure should be such that a developer is not forced to wage specificity wars or resort to hacks to achieve the desired look. Better yet, the library can aid the developer on this mission or even solve some common style problems for the developer, such as focus outlines - without forgetting customization. However, as Järveläinen [44] points out in her thesis about creating a React Native component library, when the needs of the consumer are not known, the developer of the library must take into consideration different possibilities and balance between versatility and simplicity. Finding such balance is tricky; ultimately, it is impossible to anticipate and meet all users' needs.

1.2.1 Personal motivation

Based on work experience, I've found that a well-designed and easy-to-use component library theming solution can significantly speed up styling, ease maintenance, and allow for the easy sharing of styles within an organization. However, even an extensive and theoretically well-made library theme (styling solution) can be challenging to use if poorly documented.

While working with the same customer on multiple projects, I found that with a good style guide in hand, I could sit down and define large parts of the application's theme (including component variants) right at the beginning of the project. With slightly more funding and time, it would have been easy to define a shareable base theme that each project could have pulled from a repository and extended (especially as a shared project template repository already existed and was in use). This experience made me curious about how the themes of other component libraries compare to the one I was using (Chakra UI).

[2] https://rachelandrew.co.uk/, visited 1.6.2024
[3] https://chriscoyier.net/, visited 1.6.2024
[4] https://nerdy.dev/, visited 1.6.2024
[5] https://www.kevinpowell.co/, visited 1.6.2024

1.3 Scope and research questions

Based on the above, I have chosen to focus on the following research questions:

1. How does the theme implementation and structure impact theming and customization in terms of ease of implementation and reusability of styles?
2. How flexible is the theme to accommodate various design requirements?
3. How do the provided theme documentation, examples, and type definitions (TypeScript) support development?

By "theme" and "theming" I refer to the combination of styling methods used and provided by the library to customize the appearance of components. As many component libraries provide some centralized customization mechanisms, the term theme is often used to describe not only the overall look of the components but also the structure for styling them. Additionally, all component libraries investigated in this research already come with default styles, which can also be referred to as themes due to their coherent nature.

In this thesis, I touch on but do not extensively cover, some of the other concerns and angles that can be used to evaluate and select a component library. Instead, I focus on the topic through a narrow lens, specifically looking at development and styling. However, it is essential to note that criteria such as accessibility, performance, usability, and maintenance of the project are likely more important considerations when selecting a component library.

1.4 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 offers an overview of term developer experience and how it has, or rather has not, been previously studied in context of styling. Chapter 3 discusses evolution and benefits of design systems and how those concepts relate to developers' work. In Chapter 4, overview of CSS and types of UI libraries is provided. In Chapter 5 React and ways to style React components and component libraries are introduced. Chapter 6 details the practical implementation at the heart of this thesis, including selection of compared libraries, overview the reference application, the base implementation and library-specific versions. Chapter 7 presents the results of the comparison in relation to the research questions. In Chapter 8 I discuss other themes that arose from the experiment as well as limitations of the study. Finally, Chapter 9 summarizes the key findings and potential future research directions.

2 Developer experience

In 2012, Fagerholm and Münch [26] described developer experience as "a concept that captures how developers think and feel about their activities within their working environments, with the assumption that an improvement of the developer experience has a positive impact on software development project outcomes". Following a model used to describe the mind in psychology, Fagerholm and Münch [26] divide developer experience into three dimensions: cognition, affect (feeling, emotion), and conation (impulse, desire, volition, striving). Each of the dimensions consists of several aspects, but can be summarized as following questions:

Cognition
: How do developers perceive the development infrastructure?

Affect
: How do developers feel about their work?

Conation
: How do developers see the value of their contribution?

The research questions presented in Section 1.3, focus mostly on the cognition element of the developer experience. Though perceived difficulty or ease of implementation can also relate to affect.

2.1 Prior research into developer experience of styling

To my knowledge, developer experience of styling web applications has not been explicitly - or at least not widely - studied. However, some studies touch on the subject. To understand the cognitive challenges that developers working with JavaScript frameworks (but all respondents were using React) face, Yunita [95] conducted interviews with ten developers and analyzed the results. Component creation was the most widely mentioned challenges, especially when it comes to styling and maintaining responsiveness.

2.2 CSS challenges described in working diaries

Small snippets of developers' experience of styling can be found in working diary theses written in universities of applied sciences. For example, while reflecting on areas of future learning and improvement, Kivinen [48] writes that her "skills in CSS still need a lot of work" and she would "like to concentrate more on knowing what to do, instead of trial and error methods". While this is just an individual example, it echoes the sentiments found in Yunita's [95] interviews and heard by the author. Instead of leaning on in-depth knowledge of CSS, some developers simply try different things to see what works. This is not only frustrating and inefficient but leads to uninformed design decisions and potentially poor code quality. As another student [49] writes about his solution to a CSS issue: "This probably isn't the best solution, but the results were exactly what I want".

2.3 Resorting to CSS hacks

Resorting to "fixes" to get the desired styling result is a common theme, especially among students. As part of their study on teaching web development at universities, Alston et al. [9] interviewed five university lecturers to identify concepts their students struggled with. CSS came up in answers, especially specificity and cascading, and lecturers described their students resorting to "fixes" to get their desired styling results. Using hacks or not understanding styles can lead to a situation where developers are afraid to touch the styles. As Kollin [49] writes in his working diary: "One small change could destroy the whole look of everything".

2.4 Conclusion

In conclusion, CSS and styling in general are areas where developers, especially inexperienced ones, struggle. As a technology that, on its surface, seems simple, CSS can be surprisingly complex and hard to master. If developers are not taught the best practices and principles of CSS, they can resort to the aforementioned "fixes" to get the desired styling results. This creates technical debt in the form of poorly structured and hard-to-maintain styles, making styling even more challenging and frustrating, creating a vicious cycle.

3 From patterns to design systems

The idea of identifying common design problems and providing reusable, adaptable solutions precedes the world wide web by almost two decades. In 1977, architect, design theorist, and professor of architecture Christopher Alexander, with his students Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel published a book called *A Pattern Language* [8], a collection of 253 "patterns" for built environments ranging from towns to individual rooms. Each pattern consists of a common design problem and a core solution. Linked to each other, the patterns form a kind of language.

This concept of describing design challenges and solutions as "patterns" has influenced various fields, but especially computer science [90]. Alexander's work inspired Kent Beck and Ward Cunningham to write their first small pattern language in 1987 to guide Smalltalk programmers [13]. Another well-known example of design patterns in software engineering is the book *Design Patterns: Elements of Reusable Object-Oriented Software*, also known as the Gang of Four book, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [30].

3.1 Design systems

Design systems are a natural continuation of this pattern thinking. Fessenden [27] describes a design system as "a complete set of standards intended to manage design at scale using reusable components and patterns". A design system often consists of both intangible and tangible elements [39]. Fessenden even considers the design system to include the people who maintain the system. Design systems can consist of several parts, such as style guides, a component library, and a pattern library [27]. In the tech world, some of the most well-known or at least recognizable design systems are Material Design by Google[6] and Apple's design system Apple Human Interface Guidelines[7].

Style guides often contain design principles, color palettes, typography, visual references and other guidelines for creating a consistent visual identity. Commonly, style guides focus on brand elements, such as logos and colors, but they can also include guidelines on content, such as tone of voice and writing style, or interaction design [27].

Component libraries (also sometimes referred to as **design libraries**), are collections of reusable UI elements. Component libraries will be discussed in more detail in Chapter 4.

The term **pattern library** is sometimes used interchangeably with "component library" [90], but there is a difference between the two. While component libraries specify individual elements, pattern libraries are collections of UI elements or layouts [90]. Pattern libraries can thus be less detailed and contain rough, though still reusable, patterns.

[6] https://m3.material.io/, visited 5.3.2024
[7] https://developer.apple.com/design/, visited 5.3.2024

In conclusion, a design system's content can vary depending on the organization's needs and the system's purpose. However, in the context of this thesis, it is sufficient to understand that a design system is a comprehensive system of guidelines and implementations (such as component libraries) that follow those guidelines. Additionally, a component library by itself does not necessarily form a design system if it does not contain above described the standards and practices for using said components.

3.2 Atomic Design

Atomic design is a design methodology for creating design systems developed by web designer Brad Frost. It helps designers construct design systems as reusable patterns that better translate to reusable components. Frost introduced the concept in 2013 in his blog post [28] and later published a book under the same name [29]. Inspired by chemistry, atomic design describes the fundamental building blocks of a design system as atoms [29]: "If atoms are the basic building blocks of matter, then the atoms of our interfaces serve as the foundational building blocks that comprise all our user interfaces". As Figure 1 illustrates, these atoms can then be combined to form increasingly complex structures: molecules, organisms, templates, and pages.

1. **Atoms** are a design system's smallest, functional building blocks. These can be abstract elements such as colors and font styles or fundamental user interface elements such as buttons, headings, or input fields. Atoms cannot be broken down further without losing their functionality or meaning.

2. **Molecules** are groups of atoms forming relatively simple, reusable components with a specific function, such as a search form.

3. **Organisms** are relatively complex components consisting of molecules and atoms - and potentially even other organisms.

4. **Templates** are "page-level objects that place components into a layout and articulate the design's underlying content structure." [29]. They are page skeletons that focus on defining the page's overall layout and content structure. The actual content will be included in the next stage.

5. **Pages** are instances of templates with actual representative content. They mimic the final, concrete implementations of the design system.

It is important to note that atomic design is not a linear process but a mental model that allows designers "to concurrently create final UIs and their underlying design systems" [29]. Atomic design offers a structured approach to design systems and user interfaces, easing navigation between individual parts and the whole UI and providing language to discuss UI patterns and content placed in them. This shared terminology can ease collaboration between designers and developers [47].

Besides a helpful mental model and language, atomic design has other benefits. By encouraging designers to create, reuse, and maintain this library of atoms and

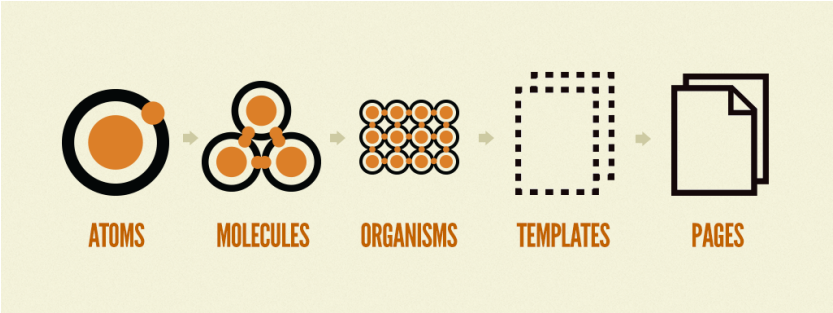


Figure 1: Levels of atomic design illustrated by Brad Frost [28]

molecules and use them across products, the atomic design promotes consistency, efficiency, and maintainability [47]. I would also argue that these concepts translate well into development even without a design system and can help also front-end developers to create more modular and maintainable UI components.

3.3 Responsive design

When designing for print, the designer knows the exact dimensions of the medium they are working with and has complete control over the precise positioning of each element. However, on the website, the page size is set by the user's browser window instead of the designer. In the early days of the web, this was not as much of a problem as most users had similar screen sizes and resolutions. Hence, early web designs attempted to mimic the precise control of print design by using fixed widths, tables for layouts, and pixel-based measurements. However, as the variety of screen sizes and devices increased, this approach was no longer viable.

Responsive websites adapt to different screen sizes and devices, such as desktops, tablets, and smartphones. Instead of building multiple websites to serve different devices, the same site is designed to change based on available screen size and provide a good user experience regardless of the device used. While the most visible changes often happen in layout and typography, considering other device differences, such as interacting with the UI via touchscreen, is also essential.

I argue that responsive design has been a similar leap for designers as the original transition from print to the web. As designer Laura Kalbag wrote in 2012 [45]: "Responsive web design brought us back to pre-table layout fluid sites that expanded or contracted to fit the viewport".

However, I believe that moving towards component-based design and development has made responsive design easier. When components are first designed in isolation and then combined into larger structures, they are easier to adapt to different layouts. Given that each component has certain non-negotiable properties, such as minimum and maximum widths, switching between different screen sizes becomes a matter of defining layouts that serves as boundaries for components to exist in.

3.4 Conclusion

In this chapter, we have explored the evolution from architectural patterns into pattern thinking and the creation of design systems. Design systems consist not only of their concrete implementations, such as component libraries but also of the principles and guidelines that help teams create consistent and scalable user interfaces. Principles such as atomic design can help create maintainable and scalable design systems that map seamlessly to actual component implementations. Finally, as the number of devices and screen sizes continues to grow, responsive design is a necessary part of designing user interfaces. Due to the fluid nature of responsive design, designing user interfaces out of reusable and adaptable components allows easier adaption to different screen sizes without the need for separate implementations for each device.

4 CSS and types of UI libraries

This chapter offers a brief overview of the language of web styling, CSS and some styling methods relevant to the libraries discussed in this thesis. However, styling specifically React components is discussed in more detail in Section 5.6. Finally, some terms related to UI libraries are clarified.

4.1 CSS

CSS (Cascading Style Sheets) is a style sheet language used to define the presentation of documents written in markup languages like HTML and XML. CSS rules consist of selectors, properties, and values. A selector targets elements in the DOM, and the properties and values define how the elements should be styled.

```
selector { property: value; }
```

For example, the following CSS rule sets the color of all <h1> elements in the rules scope to red and underlines them:

```
h1 {
    color: red;
    text-decoration: underline;
}
```

Selectors can target elements based on their tag name, class, id, or for example, their state or relationships to other elements. Combining selectors allows for more specific targeting or applying styles to multiple elements at once.

These days, it is also possible to nest CSS rules. Previously, this was only possible with preprocessors like Sass or Less, which compiled the nested rules into flat CSS. Now, newer versions of all major browsers can natively parse nested CSS rules [12]. When used in moderation, nesting can make CSS easier to read and maintain [86, 62]. Repetition of selectors is also reduced, which can reduce the file size [86, 62].

Which styles apply to an element is determined by three mechanisms: the cascade, specificity, and inheritance [24]. The cascade defines the order in which styles are applied, and specificity determines which styles take precedence. Cascade and specificity determine which rule is applied if multiple rules target the same property of the same element. Inheritance defines which styles are passed down from parent to child elements. Some properties are inherited by default, while others are not.

Additionally, some properties, such as `color` for text color, only change one aspect of the element. In contrast, others, such as `display` or `position`, can impact the element's layout and behavior in multiple ways. Understanding, at least on a basic level, how these mechanisms work is crucial for successfully writing style rules. But in the context of this thesis, it is sufficient to know that this complex interplay of several mechanisms exists and determines which styles are applied to an element.

4.2 Types of UI libraries

Before diving into specifically component libraries, it is worth clarifying some related and overlapping terminology as terms around UI-related packages and systems are often used interchangeably, overlap in meaning, and can lack clear definitions. Hence, the definitions provided here are neither exhaustive nor universal. They have been compiled from various industry articles and blog posts to provide a general understanding of the terms in the context of this thesis.

Firstly, by the term "UI libraries", sometimes called UI frameworks, I refer to any package that provides UI styles (including layout utilities), components, or both. I see "UI libraries" as an umbrella term that encompasses, for example, CSS frameworks and component libraries.

The term design system is often also used to refer to a technical implementation of a design system, namely a component library that follows the design system's guidelines. But as discussed in Section 3.1, a design system is a broader concept that can include a component library as one of its elements.

4.3 CSS frameworks and utility classes

CSS frameworks are collections of predefined styles, usually in the form of CSS classes that users can apply their HTML - or JSX for React - elements. CSS frameworks can be categorized based on the framework's complexity and features.

Simpler frameworks might only provide a grid system, while more complex ones offer styles for typography, buttons, forms, and other common HTML elements. The benefit of a more rudimentary, grid-only framework is that it is lightweight and does not impose a specific style on the website.

The line between CSS frameworks and component libraries can also be blurry. For example, several of the CSS frameworks that Kopperi [50] compares in his thesis, such as Bootstrap and UIKit, include custom components and JavaScript to enhance functionality.

Some CSS frameworks, such as Tailwind CSS[8], provide utility classes for styling elements. Utility classes are not a specific technology or library but rather a way of writing CSS - that can then be gathered together to form a framework. Utility classes are small, single-purpose classes that apply a particular style to an element. They are often used to apply common styles like margins, padding, text alignment, and colors. Utility classes can be combined to create more complex styles, and they can help keep the style sheet small and maintainable by reducing redundancy.

Utility classes, even as frameworks, are not particularly new technology. Bootstrap[9], developed by Twitter, has been in public use since 2011 and offers utility-first CSS but with a more opinionated styling [51]. The aforementioned Tailwind CSS, released in 2017, is a utility-first CSS framework that provides a comprehensive set of utility classes for styling elements but does not prescribe a specific visual identify [10]. Tailwind CSS has gained popularity for its flexibility and ease of use [10].

[8] https://tailwindcss.com/, visited 3.4.2024
[9] https://getbootstrap.com/, visited 3.4.2024

4.4 CSS Preprocessors

CSS preprocessors, such as Sass, Less, and Stylus, are tools that extend CSS with mixins, nested selectors, loops, and other features. Especially before the introduction of CSS variables, preprocessors were used to make writing CSS easier and more maintainable. Preprocessors can help reduce redundancy, improve organization, and make style sheets more maintainable.

4.5 Component libraries

Component libraries are collections of ready-made components, such as buttons, inputs, modals, text elements, etc. They serve as the building blocks of an application's user interface. Often, component libraries include predefined styles that can be customized to a varying degree. How opinionated and customizable a component library is depends on its origin, purpose, and styling methods.

Some may be originally intended for an organization's internal use and later released to the public, while others are built from the ground up to be open-source and widely used. Some libraries, for example Bootstrap, have recognizable look and feel and may follow a set design system or language, such as **Material UI**[10], that follows Google's **Material Design**[11] principles.

Using a component library offers developers several benefits but also comes with some downsides. Though many of these upsides and downsides also apply to self-created component libraries, the next section focuses on the use of third-party libraries. This means that someone else has created and maintains the library and the user has no direct control over the library's development. Thus, using someone else's component library is always, to some degree, a trade-off between effort and control.

4.5.1 Benefits of component libraries

One of the primary advantages of using a component library is increased development speed [87]. Libraries often come with pre-built, complex components that can be customized, saving developers significant time and effort compared to building the same components from scratch. Effort is saved, especially when it comes to more intricate elements.

If a component library's style fits the design requirements, the library can provide an easy way to create a visually appealing application [87]. And even if customization is needed, using a component library promotes stylistic consistency across an application [87, 91]. Developers can ensure a cohesive visual style throughout the application by customizing and reusing components.

Using a third-party library can also reduce maintenance effort [3]. As Järveläinen [44] pointed out, a self-created library requires continuous maintenance just to keep up with newer versions of dependencies (such as React versions) that its consuming applications will want to use. With a third-party library, this maintenance

[10] https://mui.com/, visited 5.3.2024
[11] https://m3.material.io/, visited 5.3.2024

effort falls on the library's developers instead of the consumer. This can significantly reduce the ongoing maintenance burden on developers.

Furthermore, if the library includes comprehensive tests, it can reduce the the testing effort the consuming application requires, allowing developers to focus on testing business logic and custom functionalities [44]. Similarly, a well-established library can provide better accessibility and browser compatibility than a self-created library, as achieving these takes effort and expertise [87, 91].

Finally, using a component library can provide support from the user community. If a library is widely used, it is more likely to have an active community of users and contributors who can help with issues and provide guidance on best practices [3].

4.5.2 Downsides of component libraries

However, there are also downsides to using third-party React component libraries. One potential issue is the increased bundle size, especially with large libraries with many styles and custom behaviors [87]. Although techniques like tree-shaking can mitigate this problem, it remains a concern.

Additionally, accessibility issues can arise, and fixing these issues may be challenging if the library's maintainers are overworked or slow to accept pull requests. While using a well-maintained library can resolve many common bugs and concerns, there can still be friction when addressing issues within the library.

Finally, one downside can be the inflexibility and lack of customization options in some libraries. Highly opinionated libraries might not match the application's requirements visually, necessitating further customization efforts. To mitigate some of these customization and styling challenges, developers might consider using headless UI libraries, such as Headless UI[12], as those provide more flexibility and control over the appearance of components due to minimal initial styling, but still offering other benefits listed above.

4.6 Conclusion

As mentioned in the beginning of this chapter, lines between different types of UI libraries are not always clear. Terms such as CSS frameworks, UI libraries, and component libraries overlap and a specific product might contain features of all three. However, the main difference between a CSS framework and a component library is that a CSS framework focuses more on styles and layout, while a component library provides components that are styled and often interactive. The term "UI library" could then be consider an umbrella term that covers both - and maybe some additional - terms.

[12] https://headlessui.com/, visited 5.3.2024

5 React and styling React components

React.js, or simply React, is a JavaScript library developed by Meta (formerly Facebook) for building user interfaces. Though the terms library and framework are often used interchangeably, React is a library, not a framework [7]. It provides neither a complete toolset for building an application nor a strict structure one must follow.

5.1 Why React was developed

React was born out of Facebook's need to improve synchronization between application's state and the user interface, particularly in managing Facebook Ads [41]. Traditionally, updating the UI required checking what data changed and then manually manipulating the DOM (Document Object Model) with JavaScript, which was a tedious and error-prone process [41, 40].

Jordan Walke, a product engineer on the Facebook Ads team, played a pivotal role in developing React [37, 35]. He had been improving upon Facebook's JavaScript MVC framework Bolt.js, creating in 20211 a prototype project called FaxJS[13], a predecessor to React. Improving on FaxJS, Walke created the first version of React in 2012 [37].

5.2 Philosophy of React

Walke's innovative idea was to re-render the UI whenever data changed. React works by keeping a virtual representation of the user interface, called **the Virtual DOM**, in memory and only updating necessary changed content in the browser in the "real" DOM - the content rendered in the browser - when the application's state changes [88]. React's approach is declarative, meaning that developers describe what they want the UI to look like, and React takes care of how [92]. This not only abstracted away the DOM handling simplifying developer's work, but also managed to improve performance [41].

5.3 Early usage of React

React was first used to implement comments in Facebook's news feed [37]. As Facebook acquired Instagram in 2012 and there was a desire to adopt React at Instagram, the decision was made to open-source React [35]. The library was released to the public in May 2013 on GitHub[14] and presented at JSConfUS by Jordan Walke and Tom Occhino [1]. Though the initial reaction to React was mixed, with some developers viewing the re-rendering approach as a step back [37], the library quickly gained interest and an active developer community. Within a couple of years of its public release, React was adopted by companies such as Airbnb and Netflix [35]. React's popularity has continued to grow, and it has become a cornerstone of modern

[13] https://github.com/jordwalke/FaxJs, visited 3.7.2024

[14] React's initial public release commit on GitHub, visited 3.6.2024

web development. In 2021, React surpassed jQuery as the most popular front-end technology or framework according to Stack Overflow's developer survey [78].

5.4 Component-based architecture

Virtual DOM, one-way data flow, and declarative programming helped simplify developers' work and improve performance, thus contributing to React's popularity. Yet another core factor in React's popularity is its component-based architecture, where the user interface is broken into small reusable components [92]. This supports reusability, simplifies management of complex UIs, and improves code maintainability and testability [92]. React documentation's quick start guide states, "React apps are made of components. A component is a piece of the UI (user interface) with its own logic and appearance. A component can be as small as a button or as large as an entire page." [23]

Though breaking UI into components is not unique to React, it has always been a core part of the library's philosophy. When first presenting React at JSConfUS, Walke stated that the "primary focus of React is allowing you to build custom components" [1].

5.5 Introduction of JSX

```
export default function TodoList({ todos }) {
    if (todos.length === 0) {
        return <p>No todos!</p>;
    }
    return (
        <div>
            <h1>Todo list:</h1>
            <ul>
                {todos.map(todo => (
                    <li key={todo.id}>{todo.text}</li>
                ))}
            </ul>
        </div>
    );
}
```

Listing 1: Simple React component written using JSX

To support and simplify the creation of components, the React team also introduced an optional syntax extension **JSX** (JavaScript Syntax Extension) that allows writing HTML-like markup inside JavaScript files [1]. As shown in Listing 1, JSX looks similar to HTML, but it has some stricter rules and allows writing JavaScript expressions such as variables, loops, and conditional statements in JSX markup [93]. Though React components can be written without JSX - and JSX will be turned into JavaScript objects

- most React developers prefer to write components in JSX due to its conciseness and readability [93].

5.6 Styling React components

Styling React components is not all that different from styling static HTML pages. While the markup is JSX instead of HTML, and hence some syntax details differ, the underlying language CSS, discussed in more in Section 4.1, remains the same. Next, we will briefly discuss different ways styles can be applied to React components.

5.6.1 Inline styles

The most basic way is to use inline styles, where styles are defined directly in the component's prop `style` as a JavaScript object, as demonstrated in Listing 2. As the property names are object keys, CSS properties that normally contain a hyphen (e.g., `font-weight`) need to be written in camelCase (e.g., `fontWeight`) [66].

```
const WarningMessage = (message) => {
    return (
        <p style={{ color: "red", fontWeight: "bold" }}>
            {message}
        </p>
    );
}
```

Listing 2: Inline styles in a React component

While this approach is simple and safe as it is scoped to the component, it offers only a subset of what CSS can do and can - in large quantities - make the component code harder to read. Also, applying the same inline styles to too many elements is not ideal from a performance perspective [53].

5.6.2 CSS Style sheets

Traditional CSS style sheets can be used to style React components. Simply import the CSS file into a component or application and use class names to apply styles. This method provides a separation of concerns, making it easier to maintain and scale style sheets across your application.

However, global style sheets can lead to naming conflicts, specificity issues, and a lack of encapsulation. Additionally, it can be hard to track which styles are being used where, especially in larger applications.

5.6.3 CSS Modules

CSS Modules allow scoping CSS class names to a specific component, avoiding naming conflicts and providing better encapsulation. When you import a CSS module

in a JavaScript file, you get an object containing mappings of class names to unique identifiers, which you can then use in your JSX.

5.6.4 CSS-in-JS libraries

CSS-in-JS libraries, such as styled-components, Emotion, and JSS, allow writing CSS directly within JavaScript code using tagged template literals or JavaScript objects. This approach offers component-level styling and enables writing styles in the same file or within the component. As it is possible to use JavaScript in style rules, styles can be modified based on component props and state. This flexibility can reduce duplication and allow writing highly customized yet reusable styles [53]. However, CSS-in-JS has issues, such as runtime overhead, increased bundle size, and increased complexity [53].

5.7 Plain CSS in React components

Unlike Vue.js, React does not allow placing plain CSS in the same file as the rest of the component code, as CSS and JavaScript have to go in separate files [53]. Vue offers Single-File Components (SFCs), a custom file format with extension `*.vue`, where the component's template, scripts, and styles can all be defined [72]. As shown in Listing 3, the files typically contain three main sections: `<template>` for component markup, `<script>` for JavaScript logic, and `<style>` tag(s) for styles. SFCs are like HTML files but without the `<head>` section, and `<template>` replacing HTML `<body>` [77]. This format allows one to keep all related code in one place, making it easier to maintain and understand. Additionally, the styles defined in the `<style>` tag can scoped to the component, which avoids global style conflicts and keeps styles from bleeding to a component's children [71].

In React, styles can be defined in the component files using the CSS-in-JS libraries mentioned earlier. However, this approach requires additional dependencies and has its trade-offs, as discussed in Section 5.6.4.

5.8 Styling component libraries

In principle, any of the methods described in Section 5 can be applied to React component libraries. However, a component library may have a preferred way of styling components and offer library-specific variations of the methods for styling.

Traditionally, libraries also have some initial styles that, depending on the library, users can customize fully or partially. However, there is a trend towards unstyled, so-called **headless libraries**. These libraries contain styles to position component parts but offer no or minimal visual styling. Headless UI[15] by Tailwind Labs and Ariakit[16] by Diego Haz are examples of such libraries. Additionally, some libraries that are styled by default offer an option to use the library as a headless one. These

[15] https://headlessui.com/, visited 3.4.2024
[16] https://ariakit.org/, visited 3.4.2024

```
<template>
    <h1 class="example-title">{{ title }}</h1>
</template>

<script>
export default {
    data() {
        return {
            title: 'Hello world!'
        }
    }
}
</script>

<style>
.example-title {
    color: blue;
}
</style>
```

Listing 3: Vue Single-File Component example

approaches allow developers to fully customize the appearance of the components, making them more flexible and easier to integrate into existing design systems.

5.8.1 Centralized theme definition

A relatively common practice among React component libraries is to provide some centralized theme definition and a set of design tokens such as colors, typography, spacing, and more. Such a definition often takes the form of a JavaScript object that can be applied to the application via a context provider or a hook.

Depending on the library, the theme object can often be used to customize, for example colors, typography, spacing, and other design tokens. The theme object might also include a central way to customize components' base styles and variants. Examples of such definitions will be presented in results sections, in Listings 10 and 12.

5.9 Conclusion

React, originating from Facebook's performance needs, has become a popular and influential library. Its component-based approach has contributed to its popularity and helped to popularize component-driven development. Component-based architecture enhances code reusability and maintainability and simplifies the management of complex user interfaces.

Styling in React is similar to styling traditional HTML pages, and each available

method has advantages and trade-offs. However, integrating styling into JSX instead of HTML has some syntax differences and opens up the possibility of using JavaScript to define styles. With component libraries, developers might be constrained by the library's preferred styling method. However, headless libraries and libraries that offer headless options allow more freedom in styling choices.

6 Implementation

To compare the different component libraries, I will create the same relatively simple web application with all of them. The underlying logic will be the same for all, but UI components will be replaced with the corresponding component from the library. Theming is done via the library's theming solution or a styling solution used or recommended in the library's documentation.

To simulate a situation where a developer is following a mock-up created by a designer, I decided to create an imitation of an existing web application, Finnish Broadcasting Agency Yle's election barometer (in Finnish known as "vaalikone"). Vaalikone is a tool that helps voters find candidates by answering a series of scale-based questions. In addition to the questions, the tool displays the candidates' answers and background information. It calculates how closely the user's answer matches different candidates (and potentially political parties - depending on the type of election). Due to the writing of the thesis coinciding with the Finnish presidential election of 2024, specifically said version of the election barometer [94] was used as a reference.

The election barometer was chosen as the reference application because it:

- Has a good variety of UI elements
- Has a consistent, unique look that requires custom styling
- Can be built with a reasonable amount of effort and without a backend. Instead, data stored on the front end can be used.
- Even if features (such as graphs) are left out, and the scope of the implementation is narrowed down, the application remains usable and offers a good variety of styles to implement.

Details of the features and components that will be implemented in the imitation will be presented in the Section 6.4.

6.1 Author's developer background

To put both the implementation and evaluation into context, it is worth briefly discussing my, the author's, background in software development. I have been working as a developer for over six years, though roughly half of that time has been part-time alongside either bachelor's or master's studies in computer science. Earlier in my career, I worked fixing and migrating legacy websites with bare HTML and CSS and customized WordPress and LibGuides sites. Since 2019, I've worked in a consulting company, first on internal Slack bots (2019-2020) and later in various web development projects. Though I have some personal and professional experience in back-end technologies such as Node.js, most of my work has been in front-end roles, usually with React and TypeScript.

I've always been drawn to styling and design, which has led me to learn CSS preprocessors, such as Sass and SCSS, and to experiment with CSS art and animations. Hence, it is justifiable to say that I am more interested in and patient with CSS than

the average developer. This has also led me to take on more design-related tasks in projects and to work closely with UI/UX designers whenever possible. Clearly, this background has also influenced the choice of topic for this thesis. Thus, as discussed later in Section 8.6, my affinity for CSS and design will affect the generalizability of the results of this thesis.

6.2 Library selection

Before searching for libraries, React had already been selected as the base technology due to its wide usage and the author's prior work experience. Based on that choice, potential component libraries were sourced from recent articles and blog posts [76, 6, 61, 58, 67, 2], as well as from Reddit's React community [15]. Additional candidates were found by searching for the tag "components" on npm and sorting by popularity[17].

Only open-source libraries that are free to use were considered for the comparison, which excluded, for example, Retool[18] and Syncfusion[19]. Libraries that required sign-up or had a time limit on the free trial were rejected (e.g. KendoReact[20]).

Initial search and filtering resulted in a list of 27 libraries. These were then compiled into Tables 1 and 2 to compare their popularity and activity on npm[21] (Node Package Manager) and GitHub, respectively. Additionally, if a library had a longer pause in releases or a low number of open issues and pull requests (PRs), recent GitHub activity was checked manually to evaluate the state of a project maintenance and user engagement. The goal of the analysis was to find actively maintained and used libraries and to get a sense of their popularity.

Out of the 27 libraries, 12 were ruled out for the following reasons:

- **Rebass**: No longer actively maintained, latest release over 4 years ago in 2019
- **NextUI**: Built on top of another candidate, React Aria
- **Gestalt**: Team prioritizes internal use at Pinterest and does not accept issues from external developers
- **Carbon Design System** and **React Spectrum**: Based on their website, underlying company's design system is heavily emphasized, and hence may not be suitable when heavy customization is needed.
- **Core UI React**, **Sanity UI**, **Evergreen**, and **Grommet**: At the bottom in terms of npm or GitHub popularity
- **Ariakit**: Some of the examples are behind a paywall[22]

[17] https://www.npmjs.com/search?q=keywords%3Acomponents%20react&ranking=popularity, visited 16.3.2024
[18] Retool: https://retool.com/components, visited 18.3.2024
[19] Syncfusion: https://www.syncfusion.com/downloads/react, visited 18.3.2024
[20] KendoReact: https://www.telerik.com/kendo-react-ui, visited 18.3.2024
[21] https://www.npmjs.com/, visited 18.3.2024
[22] https://ariakit.org/plus, visited 18.3.2024

- **PrimeReact**: Documentation indicated that in near future, the library's theming architecture will be redesigned. Hence comparison results might soon become irrelevant.
- **MUI Base UI**: The library's API is being revised and no new features or components will be added to the current implementation [60, 52]. Thus the comparison results will soon be obsolete.

The remaining 15 libraries were reviewed further to understand their styling approach and customization options. The goal was to find libraries with different approaches to styling and with support for customization. Due to the large number of libraries and limited time, the evaluation and selection process was quite surface-level and based on browsing the libraries' websites, documentation, and GitHub repositories.

The Table 3 summarizes the findings from the review of the libraries' styling approaches. In the interest of space and readability, not all methods are listed, such as the availability of render props or slots for components. Also, whether modifying or adding custom classes should be considered a way to customize a global theme is up for interpretation - namely referring to React Bootstrap and Blueprint.

Component library	Package on npm	Weekly downloads	Latest release (March 18th 2024)
MUI Base UI	@mui/base	4 677 825	5 days ago
Material UI	@mui/material	3 753 843	5 days ago
React Bootstrap	react-bootstrap	2 026 557	same day
Headless UI	@headlessui/react	1 705 948	3 days ago
Ant Design	antd	1 377 839	2 days ago
Radix UI	Packaged by component, e.g. @radix-ui/react-tooltip @radix-ui/react-accordion	 1 267 767 848 139	 4 days ago 10 months ago
Chakra UI	@chakra-ui/react	569 160	3 days ago
React Aria	react-aria	441 705	same day
Mantine	@mantine/core	355 331	6 days ago
Semantic UI React	semantic-ui-react	292 793	79 days
daisyUI	daisyui	244 680	9 days ago
Blueprint	@blueprintjs/core	204 779	21 days ago
Sanity UI	@sanity/ui	204 532	6 days ago
PrimeReact	primereact	125 058	33 days ago
React Suite	rsuite	91 725	3 days ago
Next UI	@nextui-org/react	90 659	2 days ago
Fluent UI	@fluentui/react-components	84 642	same day
Ariakit	@ariakit/react	73 655	11 days ago
Theme UI	theme-ui	70 737	28 days ago
shadcn/ui	shadcn-ui	56 672 *	62 days
Rebass	rebass	55 929	4 years ago
CoreUI React	@coreui/react	54 677	11 days ago
Carbon Design System	@carbon/react	49 114	5 days ago
Grommet	grommet	35 174	5 days ago
React Spectrum	@adobe/react-spectrum	17 100	same day
Gestalt	gestalt	11 176	3 days ago
Evergreen	evergreen-ui	10 616	9 months ago

* Package shadcn-ui available on npm is a command line tool for adding component to a project [75]

Table 1: Weekly downloads of the libraries on npm from March 11th to 17th 2024

Component library	Stars	Forks	Contributors	Open issues	Open PRs
Material UI	91.1k	31.2k	2928	1571	183
Ant Design	89.8k	45k	2081	970	83
shadcn/ui	52.2k	2.7k	184	571	368
Chakra UI	36.3k	3.1k	626	251	54
daisyUI	29.8k	1.2k	178	63	2
Mantine	23.8k	1.7k	503	27	16
Headless UI	23.8k	994	86	26	3
React Bootstrap	22.2k	3.6k	470	133	51
Blueprint	20.3k	2.1k	379	648	41
Next UI	18.7k	1k	143	298	50
Fluent UI	17.5k	2.6k	814	432	211
Radix UI	13.7.k	627	47	258	75
Semantic UI React	13.2k	4k	330	176	22
Evergreen	12.3k	826	157	40	38
React Spectrum / React Aria *	11.4k	950	273	604	69
Grommet	8.3k	1k	374	262	32
React Suite	8.1k	916	113	203	26
Rebass	7.9k	657	51	46	51
Ariakit	7.5k	363	115	69	20
Carbon Design System ***	7.4k	1.7k	438	475	37
PrimeReact	5.5k	897	228	222	25
Gestalt	4.2k	329	119	- **	11
Theme UI	5.2k	671	140	49	17
CoreUI React	637	247	5	5	0
Sanity UI	114	16	15	14	0
MUI Base UI	70	12	15	91	19

* React Aria is part of multi-package react-spectrum repository
** Gestalt is Pinterest's open source design system but its small team is prioritizing in-house needs and thus not open to issues from external developers.
*** Multipackage repository

Table 2: GitHub statistics of the component libraries as of March 18th 2024. Sorted by star count.

Component library	Default styles/theme	Customize global theme	Component-level styling
Headless UI	Unstyled	No	Prop className, Tailwind plugin
React Aria	Unstyled	No	Props className & style CSS variables, Tailwind plugin
shadcn/ui	Two theme types 12 main colors all in dark & light	CSS variables Tailwind config	Tailwind classes (or user's choice)
React Bootstrap	Light (default) and dark	No	Utility classes Modify and add classes
Blueprint	Light (default) and dark	No	Library's utility classes Predefined style variables Props, Adding custom classes
React Suite	Light, dark, and high contrast	Override Less or CSS variables	Component-specific props, e.g. appearance, menuStyle
Semantic UI React	1 default	Less variables	Assign component theme Props (often Enum) Less variables
Ant Design	1 default (& automatic dark theme)	Theme design tokens	Component design tokens Component props
Fluent UI	5 themes 2 light, 2 dark, & high contrast*	Theme tokens	Component-specific props Prop `appearance` (limited values) Griffel makeStyles() (CSS-in-JS)
Theme UI	14 presets **	Theme object	Prop `sx`, theme object, variants
Chakra UI	Light (default) & dark	Theme object	Theme object Style props and `sx` props CSS-in-JS
daisyUI	32 themes	Tailwind config	Utility classes
Mantine	Light & dark	Theme object (Unstyled option)	Component props & Variants Style props, CSS Modules React Style prop (extended)
Material UI	Light and dark	Theme object or CSS variables	Inline styles via prop `sx` styled() utility (CSS-in-JS) State and component classes
Radix UI	Light (default) and dark	Override CSS variables	Classes, CSS-in-JS Data attributes Extending a primitive

* Fluent UI's high contrast theme is available but not recommended for use.
themes: https://rsuitejs.com/guide/official-themes/
** Theme UI includes option presets package @theme-ui/presets

Table 3: Library styling methods

As a result of the second review round, the following five libraries were ruled out:

- **Blueprint**: Based on the documentation, styling is limited to the predefined colors, typography styles, and other style variables and utility classes. Seemingly, those are not intended to be customized.
- **Semantic UI**: The library is React integration for Semantic UI and does not have its own styling system. Hence, the actual library under scrutiny would be Semantic UI. Also, most of the styling documentation is on the Semantic UI's website. This impeded already the initial research. Thus, the library was ruled out.
- **React Suite**: Documented customization options are limited to a set of global variables and a narrow set of component props.
- **Radix UI**: The styling section of the documentation states that "the components in Radix Themes are relatively closed—they come with a set of styles that aren't always easily overridden" [81]. Such a statement does not encourage selecting the library for a heavily customized real-life project, and thus it was not selected for the experiment either.

The remaining 11 libraries were then grouped based on their primary approach to customizing an existing theme or styling an unstyled library. It's to be noted that though multiple libraries offered styling via primarily CSS, Less, or Sass variables (Semantic UI React, Radix UI, and React Suite), all of them were ruled out due to seemingly limited customization options. However, this is only the author's perception based on the library documentation.

- **Unstyled, so-called headless, libraries**: Libraries that by default include minimal styling to simply position component parts: Headless UI, React Aria, and shadcn/ui. However, each of them also offers documentation for integration with Tailwind CSS for styling or provides a list of CSS variables to define styles.
- **Utility class -based styling**: Bootstrap provides its own set of utility classes, and daisyUI depends on Tailwind CSS. The unstyled libraries listed above overlap with this category as they all recommend utility-class-based Tailwind as a styling option.
- **Nested theme object (terminology varies)**: These libraries contain a central theme object that is used to define for example colors, typography, breakpoints, and even component styles. Each library may offer additional or alternative styling methods (such as Material UI's CSS variables labeled "experimental" in spring 2024). Out of the remaining 11 libraries five belong to this category: Chakra UI, Ant Design, Theme UI, Mantine, and Material. Additionally, though Fluent UI's theme tokens are not defined in a nested structure, it could be considered part of this group.

To ensure a diverse set of styling approaches, at least one library was selected from each of the three categories. The final selection was as follows:

- **Ant Design**: Selected due to its apparent popularity.
- **Chakra UI**: Due to the author's prior experience with Chakra UI, it was included in the comparison to offer a familiar point of reference.
- **Mantine**: Based on a rough review, Mantine seemed to have a similar central theme object as MUI, but based on discussions on forums such as Reddit, it might be easier to customize and have a better developer experience.
- **shadcn/ui**: Selected partially due to not being a typical component library. While components are styled, ownership of the code gives the user freedom to customize or even remove the provided styles.

In the next section, the selected libraries will be introduced briefly to understand their background and maturity. A more detailed look into the libraries' theming approach and customization options will be presented when discussing comparison results in Section 7.

6.3 Overview of selected libraries

This section will present an overview of the selected libraries, Ant Design, Chakra UI, Mantine, and shadcn/ui, to provide context for libraries' origins. As can seen from Table 4, Ant Design is almost three times the age of Chakra UI and Mantine, whereas shadcn/ui is only in the second year and has yet to reach version 1. While Chakra UI and Mantine are roughly the same age, Mantine is significantly higher in version numbers. This indicates rather rapid changes throughout the library's lifetime.

	Ant Design	**Chakra UI**	**Mantine**	**shadcn/ui**
Current version	5.18.0	2.8.2	7.10.1	0.8.0
released on	Jun 3, 2024	Nov 9, 2023	May 30, 2024	Jan 16, 2024
First GitHub	0.7.0	0.2.6	1.0.0	0.0.4
release	July 21, 2015	Sep 7, 2019	May 3, 2021	Mar 8, 2023
Version 1.0.0	May 10, 2016	Nov 13, 2020	May 3, 2021	-
year(s) ago	9 years	3.5 years	3 years	

Table 4: Libraries' first releases and versions used in this thesis

6.3.1 Ant Design

Ant Design is developed by Ant Group, a Chinese technology company formerly known as Ant Financial. The library advertises itself as "the world's second most popular React UI framework" [11]. It is used by companies such as Alibaba, Tencent, Baidu,

and many more [42]. The React component library is the official implementation of Ant Design design system [42]. Implementations for other technologies, such as Vue, Angular, and Blazor also exist [42].

To support designing with Ant Design, the library's website has a variety of official and third-party design files, such as mobile components and desktop components, layouts, prototypes, and plugins for design tools such as Sketch (for official design resources), Figma, and Adobe XD [69]. Presuming that the design files are up-to-date and well-organized, access to ready-made resources can speed up the design process. However, most of the listed design resources are from third parties, and at least some of them are paid, for example, a UI kit for Adobe XD[23].

To help with customization, Ant also offers a theme editor[24] on their website. There, one can modify global and component-specific design tokens in a theme with a preview of how it impacts the components. Finally, the modified theme can be downloaded as a JSON file and copied to the project.

6.3.2 Chakra UI

Chakra UI, release officially in November 2020[25], is described on the library's homepage as "a simple, modular, and accessible component library that gives you the building blocks you need to build your React applications" [18]. Built by a Nigerian software developer Segun Adebayo and developed by him and six other core contributors [18], the library is not tied to a specific company or organization. Instead, its development is funded by both sponsoring organizations and individual sponsors [18] via platform Open Collective[26] and Adebayo's Patreon page[27].

Chakra UI's components are offered as a free Figma UI Kit[28]. Though based on recent comments on the kit's page, it seems to be slightly outdated and missing components from the current version of the library (2.8.2)[29]. Additionally, a paid Figma plugin[30] for generating Chakra UI components from Figma designs is offered for 12 dollars per month per user[31].

6.3.3 Mantine

Mantine is a relatively recent React component library, as its development started in January 2021 and version 1.0 was released in May 2021 [5]. It is built and maintained by developer Vitaly Rtishchev and more than 300 other contributors [5].

[23] https://antforxd.com/, visited 21.4.2024

[24] Ant Design theme editor: https://ant.design/theme-editor, 21.4.2024

[25] Chakra UI version 1.0.0 released on GitHub on November 13th 2020 https://github.com/chakra-ui/chakra-ui/releases/tag/%40chakra-ui%2Freact%401.0.0, visited 20.4.2024

[26] Chakra UI's Open Collective page https://opencollective.com/chakra-ui, visited 20.4.2024

[27] Segun Adebayo's Patreon: https://www.patreon.com/segunadebayo, visited 20.4.2024

[28] https://v2.chakra-ui.com/figma/ui-kit, visited 20.4.2024

[29] Chakra UI's Figma Kit: https://www.figma.com/community/file/971408767069651759/chakra-ui-figma-kit, visited 20.4.2024

[30] https://v2.chakra-ui.com/figma/plugin, visited 20.4.2024

[31] https://figma.chakra-ui.com/, visited 20.4.2024

Since its release in May 2021, Mantine has gone through major versions rather quickly. Mantine 2.0.0 was released already July 5th 2021 and version 3.0.0 in October 2021 [68]. In May 2024, Mantine was at version 7.9.0. According to the Mantine documentation, the library generally releases minor versions (1.X.0) once a month or every two months, and major versions (X.0.0) every six months or once a year [5]. This cycle seems to be faster than other libraries selected for comparison. While it indicates active development and frequent updates, it may also require more effort from users to keep up with the changes.

Mantine also offers Mantine UI[32], a free collection of 123 pre-made responsive components designed and built by Mantine maintainers and the community. The Mantine UI website displays interactive examples of the components and code snippets (component code and separate CSS module style file) for easy copy-pasting. Each example also links to the component's source code on GitHub.

6.3.4 Shadcn/ui

As stated earlier, shadcn/ui is not a component library and cannot be downloaded as a dependency. It is a collection of reusable components that one can copy and paste into their application. As stated in the FAQ section on the project's website, this approach was chosen to "give you ownership and control over the code, allowing you to decide how the components are built and styled" [43]. Despite not being a library, it will be referred to as such in this thesis for the sake of simplicity.

The project was built by a design engineer who goes by the name "shadcn" online [73] and works at Vercel[33] [74]. Shadcn has previously built a variety of projects and tools, for example, a styling library Reflexjs[34] and social media post-generation tool Dito[35] [73]. While shadcn is the developer behind the collection, pull requests - both bug fixes and new components - from the community are constantly being opened and also merged into the project's main branch.

Out of the compared libraries, shadcn/ui is the newest one, as the first commit to the project's GitHub repository was made on January 24th, 2023[36]. Though the latest tagged version of the project is from January 2024, a variety of fixes and new components have been added to the project since then. For example, pre-made "blocks" of components[37] and chart components[38].

Style-wise, the project's components come with a customizable default theme [83]. The project's website offers a tool[39] to preview and customize the theme by selecting either the "Default" or "New York" style, theme's main color, border radius, and mode. A copyable set of CSS variables is provided based on the user's choices.

[32] https://ui.mantine.dev/, visited 21.4.2024
[33] https://vercel.com/, visited 22.4.2024
[34] https://reflexjs.org/docs, visited 22.4.2024
[35] https://www.dito.so/, visited 22.4.2024
[36] https://github.com/shadcn-ui/ui/commit/0a241be17ffe07cf21cc72702feb1aea852ef571, visited 22.4.2024
[37] https://github.com/shadcn-ui/ui/pull/3094, visited 22.4.2024
[38] https://github.com/shadcn-ui/ui/pull/4181, visited 7.7.2024
[39] https://ui.shadcn.com/themes, visited 22.4.2024

Component code can be copied from the project's website or GitHub repository. Alternatively, the components can be generated via a command line tool, also called `shadcn-ui` [40]. Additionally, the tool can be used to initialize configuration and dependencies for a new project [20].

6.4 Reference application

The original plan was to limit implementation scope to the two types of question cards present in the election barometer: one with a yes/no answer and another with a radio button scale. However, the scope was expanded to include the questionnaire view and a simplified version of each candidate's page.

6.4.1 Questions page

The questionnaire, or questions, page consists of a list of cards containing two types of questions: yes/no questions and scale questions from strongly agree to strongly disagree. All questions offer an option to skip the questions, thus resetting a potential prior answer. Some questions also contain a button to show a modal with additional information about the topic.

Once the user starts answering questions, a bar on top of the page will show the top candidate matches, with a match percentage based on the user's answers. The user can hide the results, in which case just gray placeholder elements are shown in the top bar. Each of the top matches shown on top of the page is a button that will open a modal with more details about a candidate and a link to their full candidate page.

6.4.2 Candidate page

In the original application, the candidate page contained, for example, videos and article links with preview images. For the experiment, the information content of the page was simplified to contain just a section of essential details about the candidate and their election promises.

The page also contained a question list of question cards similar to the questionnaire page. However, these cards were not interactive but showed the candidate's answers and comments they had provided. If the user had answered some questions before opening a candidate page, the page would show the user's answers compared to the candidate's.

6.5 Base implementation

A "base implementation" with just React and CSS was created to serve as a reference for confirming the visual correctness of the library implementations, as will be described in Section 6.5.2. Additionally, it offered a shared starting point for library implementations. Each version could be converted from the base implementations

[40] https://www.npmjs.com/package/shadcn-ui, visited 22.4.2024

instead of having to copy or recreate the logic of the applications. This section briefly describes the process and design decisions behind this base implementation.

6.5.1 Project setup

The base implementation was created using Vite's default template for React as shown in Listing 4.

```
~thesis-base % npm create vite@latest
Need to install the following packages:
create-vite@5.2.3

Ok to proceed? (y) y
Project name: thesis-base
Select a framework: > React
Select a variant: > TypeScript
```

Listing 4: Setting up base implementation using Vite

To keep the implementation focused on styling, candidate data was hardcoded into the application as objects and used via simple functions instead of fetching the data from an API and storing it in a store. User answers were stored in a simple Zustand[41] store. As in the reference application, the answers persisted in browser session storage. Instead of Finnish presidential candidates, the demo application was set up to present the four component libraries compared in the thesis.

6.5.2 Matching reference application's styles

The base implementation was matched to the reference application Vaalikone as closely as possible by comparing the applications visually side by side. The reference's CSS was also inspected with Chrome's developer tools to verify paddings, colors, font sizes, and other style properties. While the goal was not a pixel-perfect match, the base implementation was meant to be as visually similar as possible to the reference application.

However, as I verified the match visually, some mistakes could have been made. Additionally, the election barometer utilizes a custom font called "Yle Next", which I do not have the right to use. Similarly, I did not deem it ethical to copy or recreate the graphics used in the website's header and footer and thus left them out.

6.5.3 Color palette

As the reference application's styles contained plenty of unused variables, the colors actually used on the site were extracted using a color picker extension ColorZilla[42].

[41] https://github.com/pmndrs/zustand, visited 4.7.2024
[42] https://www.colorzilla.com/chrome/, visited 11.4.2024

Elements using similar colors still had to be manually inspected with developer tools. But with a limited palette in hand, the process was quite a bit easier.

Altogether, the application's palette consisted of 25 colors (and four brand colors added for libraries presented as candidates in the application). As the color palette was quite black-and-white and telling some of the colors apart would have been difficult, I gave each color a descriptive name to ease working with them. Additionally, I anticipated that naming the colors and clearly structuring my color mode variables would make it easier to transfer the palette to the library-specific implementations. Frankly, I expected to be able to more or less copy and paste the palette to each library's implementation.

6.5.4 Testing

To verify that the library-specific implementations matched the base version, simple screenshot tests were created. The screenshots used for testing were generated from the base implementation and pulled to each library's project. Running them against each project would show if the library's components were styled correctly.

Cypress[43] and a plugin for it called `cypress-visual-regression`[44] were used to create and run screenshot tests. Beyond these tests, only some unit tests for candidate match calculations were written to support reverse-engineering the candidate match percentage logic. In the interest of time and due to the application's simplicity, tests for component functionality or integration were not written. Instead, plenty of manual testing and visual inspection was done to ensure the application worked as expected.

6.6 Setting up library versions

As described in Section 6.5, a base implementation with React and bare CSS was created first to serve as a reference point and shared foundation. Once the base was ready, its GitHub repository[45] was set as a pull-only remote for each project, as shown in Listing 5. Then each version could pull updates from the base implementation, but accidentally pushing to the wrong remote could be avoided.

```
git remote add base git@github.com:Rikkokiri/thesis-base.git
git remote set-url --push base no-pushing
```

Listing 5: Setting up base repository as pull-only remote

For libraries that did not include a starter template, the base repository was first pulled to serve as a starting point for the project. On top of that, the project was set up following the library's start guide for a Vite project, adapting it as needed. The following sections describe the setup process for each library version.

[43] https://www.cypress.io/, visited 2.6.2024
[44] https://www.npmjs.com/package/cypress-visual-regression, visited 2.6.2024
[45] GitHub repository for base implementation: https://github.com/rikkokiri/thesis-base

6.6.1 Ant Design

Ant Design did not offer a starter template, but there was a getting started guide for Vite[46]. However, the instructions were not very detailed and mostly showed how to set up a new, empty Vite project without making it clear that's all it was. So I started by pulling the main branch from the base repository to a repository created for Ant Design version[47]. Then, I installed Ant Design with the command:

```
npm install antd --save
```

Beyond this, the guide did not have any specific instructions. However, in order to later utilize and customize Ant Design's theme, I had to wrap my application in the library's <ConfigProvider>. I initially placed it around the entire application, above react-router's <RouterProvider> as can be seen in Listing 6. However, as I later discovered, I needed to nest multiple <ConfigProvider>s and use my own theme detection hook to switch between themes. Hence, I later moved the <ConfigProvider> to App.tsx, as shown in Listing 11.

```
import { ConfigProvider } from "antd";

createRoot(document.getElementById("root")!).render(
  <StrictMode>
    <ConfigProvider>
      <RouterProvider router={router} />
    </ConfigProvider>
  </StrictMode>,
);
```

Listing 6: Ant Design ConfigProvider placed around the entire application in main.tsx

6.6.2 Chakra UI

As with Ant Design, I pulled the base repository to a new repository for Chakra UI version [48]. Then I followed Chakra UI's guide for a Vite project[49]. Besides the library itself, Emotion[50] and framer-motion[51] needed to be installed.

```
npm i @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

Unlike Ant Design's starter guide, Chakra UI's instructions included how to set up the theme provider, shown in Listing 7, and how to get started with customizing the

[46] https://ant.design/docs/react/use-with-vite, visited 6.6.2024
[47] GitHub repository for Ant Design version: https://github.com/Rikkokiri/thesis-antd
[48] GitHub repository for Chakra UI version: https://github.com/Rikkokiri/thesis-chakra
[49] https://v2.chakra-ui.com/getting-started/vite-guide, visited 10.6.2024
[50] https://www.npmjs.com/package/emotion, visited 4.6.2024
[51] https://www.npmjs.com/package/framer-motion, visited 4.6.2024

theme. Chakra UI also offers its own component for detecting user's color scheme preference, which eliminated the need for `useDetectTheme()` created for the base implementation.

```
import { ChakraProvider, ColorModeScript } from "@chakra-ui/react";
import theme from "./theme";

ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <ChakraProvider theme={theme} resetCSS={true}>
      <ColorModeScript initialColorMode={theme.config.initialColorMode} />
      <RouterProvider router={router} />
    </ChakraProvider>
  </React.StrictMode>
);
```

Listing 7: Abbreviated version of main.tsx with Chakra UI's theme provider placed around the application

6.6.3 Mantine

Mantine project was created using the library's own Vite template[52], which included configuration for Vitest[53], Prettier[54], Storybook[55], ESLint[56], and TypeScript[57]. A simpler template with just default Vite configuration[58] was also available, but the more feature-rich template was chosen to save time on setting up tooling that was already going to be used for in the project, namely Prettier, ESLint and TypeScript.

GitHub offers an option to create a new repository from a template repository, so the template was used to create a new repository [59] for Mantine version. As the template was not at the time using the latest version of Mantine (7.8.1 instead of 7.9.0), the library's version was updated manually:

```
npm install @mantine/core@latest @mantine/hooks@latest
```

Later, during the implementation, as new versions of Mantine were released, the dependencies were updated again (7.10.1 being the latest version used).

While having all of the tooling included in the template initially saved some time, it also required merging the configuration with the base implementation's setup. As the template used different file types to configure Prettier and Vite, I mistakenly included

[52] https://github.com/mantinedev/vite-template, visited 11.4.2024
[53] https://vitest.dev/, visited 4.6.2024
[54] https://prettier.io/, visited 4.6.2024
[55] https://storybook.js.org/, 4.6.2024
[56] https://eslint.org/, visited 4.6.2024
[57] https://www.typescriptlang.org/, visited 4.6.2024
[58] https://github.com/mantinedev/vite-min-template, visited 11.4.2024
[59] GitHub repository for Mantine version: https://github.com/Rikkokiri/thesis-mantine

some duplicate files when the initially pulling code from the base repository. This did not end up causing any actual problems, just slight confusion and extra work of removing duplicate configuration files.

Since I did not end up utilizing Storybook or Vitest in the project, I could have started with the simpler template or followed a similar process to other libraries. But for a normal use case of starting from scratch, the template provided an excellent starting point.

6.6.4 Shadcn/ui

As with other versions, a new repository was created for shadcn/ui version[60]. The base version was again taken as the starting point, as described in Section 6.6. Then project tooling could be setup following Vite specific instructions[61]. I just skipped the initialization of a new project and some configurations steps that I had already taken, such as resolving paths in Vite. Hence the process was as follows:

- Added Tailwind CSS and its peer dependencies:
  `npm install -D tailwindcss postcss autoprefixer`
- Generated Tailwind and PostCSS configuration files:
  `npx tailwindcss init -p -ts`
  (option `-ts` added due to using TypeScript)
- Ran shadcn-ui (command line tool) init command to setup the project:
  `npx shadcn-ui@latest init`

The `shadcn-ui` command line tool provided a series of questions to configure the project, as shown in Listing 8.

```
~thesis-shadcn % npx shadcn-ui@latest init
Would you like to use TypeScript (recommended)? ... no / yes
Which style would you like to use? > Default
Which color would you like to use as base color? > Slate
Where is your global CSS file? ... src/styles/global.css
Would you like to use CSS variables for colors? ... no / yes
Are you using a custom tailwind prefix eg. tw-? (Leave blank if not) ...
Where is your tailwind.config.js located? ... tailwind.config.ts
Configure the import alias for components: ... @/components
Configure the import alias for utils: ... @/lib/utils
Are you using React Server Components? ... no / yes
Write configuration to components.json. Proceed? ... yes
```

Listing 8: Setting up shadcn/ui project using its command line tool

[60] GitHub repository for shadcn/ui version: https://github.com/Rikkokiri/thesis-shadcn
[61] https://ui.shadcn.com/docs/installation/vite, visited 4.6.2024

Each component's code was then generated using the library's CLI tool. For example a button component's code was added with the command:

```
npx shadcn-ui@latest add button
```

This generated a new file in the components directory with the button component's code. As the default styles were applies as Tailwind classes, each component consisted of a single file.

6.7 Conversion from base implementation

Before diving into the results, I will provide a brief overview of how I approached converting from the base implementation to versions based on each component library. As described in the project setup (Section 6.6), each library-specific version started as a copy of the base implementation or, in the case of Mantine, the base implementation was merged with the starter template.

If a component library offered layout components, such as `<Flex>`, or more specific ones, such as `<Header>`, I started by replacing some of the layout-focused elements and my own layout utility classes with said components. The extent to which layout components, cards, and text elements were converted to the library's components depended on the library. Namely, shadcn/ui did not offer layout-specific components (such as `<Flex>`) like Chakra UI and Mantine did, and thus `<div>` elements or relevant semantic elements were kept instead. At the same time, I'd also familiarize myself with the library's approach to colors and try, for example, to apply footer and header colors using the library's approach.

Regarding components, the first ones I'd usually tackle were the buttons. First, I'd replace the use of a regular stateless button and then move on to the toggleable buttons that I'd either build based on the library's button component or, if available, use a toggle component.

Alongside or before customizing the button, I'd usually replace typography classes and elements (such as `<h1>` and `<p>`) with the library's components. This would often involve defining some typography styles in the library's theme object.

After the buttons and typography, I'd move on to, for example, modals and a card component (if the library offered suitable ones). As I would progress, I'd also search for and replace the base implementation's utility classes (such as `.m-0` and `.mb-16` for margins) with the library's style props, inline styles, or other styling methods, thus eliminating the need for base implementation's util.css style sheet.

The last components I'd usually convert were the radio button ranges, which I expected to be the most complex component to convert. Especially as the version of the radio range that was used to display candidate and user answers together required various conditionally rendered elements and styles.

7 Results

This chapter presents the findings from implementing the application with each of the four component libraries. Each library is discussed in relation to the following aspects. Firstly, ease - or difficulty - of implementation and reusability of created styles is discussed. Secondly, the extent and ways the libraries could accommodate the design requirements are evaluated. Thirdly, how the library's documentation supported development is analyzed. After the main findings, the time spent on each implementation is discussed briefly. While the time spent is not the main focus of this study, it is included for general interest and to provide context. For the same reason, the size of generated CSS is also discussed.

In terms of customization, the most prominent needs were related to the color palette - both in light and dark mode - button styles, typography, and radio buttons. The reference application contained a custom "radio range" for answering questions and to show the user's answers compared to the candidate's. Hence, these are the primary areas of customization discussed in this chapter.

7.1 Ant Design

Ant Design was the most challenging library to bend to meet the design requirements. As the library is intended to follow its own design language, customization options are limited.

7.1.1 Limited component customizability

Each component has design tokens specific to it, and its different sizes and variants (e.g., primary, dashed, and text buttons) and are also set via the theme object, as shown in Listing 10. What tokens are available varies from component to component and even between component variants. For example, for the variant "default" for the `<Button>` component, there are ten design tokens, but for variant "primary" only two design tokens, `primaryColor` for text color and `primaryShadow` for box shadow [17].

As the set of tokens was so limited, achieving the planned design required overwriting the component's styles with custom CSS. This was done by overriding existing style rules with more specific ones, which required targeting class names currently used by Ant Design, as seen in Listing 9. Such a solution is brittle because the styles will break if those classes or component structures change.

One potential alternative to this specificity-based approach would have been to create a custom component that extends the Ant Design component and adds the desired styles inline. However, this would have been more work and would have required creating a custom component for each component that needed to be customized.

```css
.ant-radio-wrapper::after,
.eb-radio.ant-radio-wrapper .ant-radio-checked::after {
  display: none;
}

.eb-radio.disagree .ant-radio-inner::after {
  background-color: var(--disagree);
}

.eb-radio.agree .ant-radio-inner::after {
  background-color: var(--agree);
}

.ant-radio-wrapper .ant-radio-checked .ant-radio-inner::after {
  animation-name: scale-option;
  animation-duration: 0.3s;
  animation-iteration-count: 1;
  transition: background-color 0.3s ease 0s;
  outline: 0.25rem solid var(--radio-background);
}
```

Listing 9: Sample of CSS to override radio button styles in Ant Design

7.1.2 Global design tokens hindering customizability and ease of implementation

Though there are component-specific design tokens, much of the styling is meant to be done using a limited set of global and component-specific design tokens via the library's theme object. As global tokens often impact the appearance of multiple components, changing them can potentially cause undesired side effects when trying to customize a single component. Namely, a specific style might only be shared by a subset of affected components. Additionally, the list of impacted components was stated ambiguously for some of the tokens. For example, `colorBgElevated` was stated to impact "e.g. modal, pop-up, menu, etc." [25], thus leaving it unclear which components would be affected.

The experience of using global theme tokens felt similar to working with poorly maintained global CSS where changing one style might break another. In the context of this experiment, I could have caught such issues with the screenshot tests, but in a real-world application, this could have required a lot of manual inspecting to ensure that the changes did not break other parts of the application.

7.1.3 Customizability of color palette and dark mode

Dark mode could be implemented using Ant Design automatic algorithm. While the generated theme looked good, it did not match the specific color palette used in the

```ts
export const baseTheme: ThemeConfig = {
  components: {
    Layout: {
      headerHeight: 72,
      headerPadding: "0 24px",
    },
    Button: {
      borderRadius: 100,
      borderRadiusLG: 100,
      contentFontSizeLG: 16,
      fontWeight: 700,
      paddingBlockLG: 4, // Vertical padding of button
      paddingBlock: 6, // Vertical padding of small button
    },
    Card: {
      headerFontSize: 20,
    },
    Radio: {
      radioSize: 32,
    },
    Typography: {
      titleMarginBottom: 0,
      titleMarginTop: 0,
    },
  },
};
```

Listing 10: Defining component-specific theme tokens in Ant Design

reference application. Customizing the particular colors or even setting the primary color for dark mode was not possible - or at least not intended to be done. Instead, the algorithm derived the primary color for the dark theme from the light theme's primary color.

Ultimately, I defined the two color modes as nested themes under a base theme as can be seen in Listing 11. Using the hook I had created for the base implementation, I switched which theme would be active. Additionally, for some of the layout components that I did not replace with Ant Design alternatives, I kept the original CSS variable-based approach. So, only part of the dark and light modes were defined via the theme object. In conclusion, while Ant Design offered a built-in dark mode, it was unsuitable for cases where one wanted to use a specific color scheme instead of relying on automation.

As described above in Section 7.1.2, a large part of theme customization, including colors, was supposed to be done via the theme object. However, due to color differences between the components each token would impact, the global tokens were not used much. Instead, many of the colors were defined inline, in the remaining original style

```
function App() {
  const { prefersDarkMode, setActiveTheme } = useDetectTheme();
  useEffect(() => {
    setActiveTheme(prefersDarkMode);
  }, [prefersDarkMode, setActiveTheme]);

  return (
    <ConfigProvider
      theme={{
        cssVar: true,
        hashed: false,
        ...baseTheme,
      }}
    >
      <ConfigProvider
        theme={{
          ...(prefersDarkMode ? darkTheme : lightTheme),
        }}
      >
      ...
```

Listing 11: Nesting themes in Ant Design to support light and dark mode

sheets, and in the added component override style sheets mentioned above in Section 7.1.1.

7.1.4 Documentation

From a developer's perspective, the main sections of Ant Design's documentation are divided slightly oddly. Besides sections for design, blog, and resources, there are two code-related sections: development and components. A button with the text "Get started" on the Ant Design website's front page takes the user to the component overview. However, instructions for getting started and using the library with different React frameworks are under the development section. Beyond instructions for getting started, the development section includes guides on customizing the theme. As the name implies, the components section contains pages for each library's components. This top-level separation and not linking to actual starting instructions is a minor issue but could cause unnecessary confusion.

The large number of **component usage examples** is one of the strengths of Ant Design's documentation. All code examples utilize CodeSandbox[62], which allows the user to quickly try out the code in a live editor and allows interacting with the components.

A table of **component-specific tokens** is displayed as a table at the end of each component's documentation page, containing the name, description, type, and default

[62] https://codesandbox.io/, visited 1.7.2024

value. Tokens are listed in alphabetical order, which, depending on the component and naming scheme of its tokens, might mean that either tokens related to the same part of or style property of the component are grouped together. Alternatively, if a component has multiple variants, the token for each variant might be grouped together. Or the order might be a mix of both.

For example, for the <Button> component[63], many of the color-related tokens for button variants are named starting with the variant name and thus grouped together, e.g., `defaultBg`, `defaultBorderColor`, `defaultColor`, and so on for the default variant. However, the size of a component is often a postfix in a token name (such as `paddingBlockSM` and `contentFontSizeSM` for <Button>) resulting in the size tokens being scattered throughout the table. Breaking component token tables down into smaller tables or sections based on the type of token or the part of the component it affects could make the tables more readable and easier to navigate. Still, the tables are pretty legible, given the current structure and seemingly consistent naming scheme.

GitHub issues submitted by other users of the library are often great resources for finding answers to common issues. In April 2024, Ant Design's GitHub repository had over 29,000 GitHub issues, of which a bit over 1,000 were open[64]. However, the language barrier may limit the usefulness of the issues for non-Chinese speakers as some of the submitted issues and the following discussions are in Chinese. For example, on April 21st, 2024, ten of the 17 most recent open issues were entirely or primarily in Chinese.

7.2 Chakra UI

Chakra UI was the only one of the libraries that I had prior experience with. Having quite recently worked with it for a year and a half on several work projects, I was familiar with its theming capabilities, which added a layer of comfort to working with it. But, towards the end of 2023, I had switched from React projects with Chakra UI to working exclusively with Vue. Thus Chakra UI was not as fresh in my memory as it had been.

Likely partially due to my previous experience, I found customizing components via Chakra UI's theme object smooth and easy for the most part. Though I had not previously implemented dark mode with it, doing it via theme was straightforward. However, for elements not styled via theme, I struggled to find clear best practices for applying mode-dependent styles. Otherwise, the documentation was clear and easy to follow, and the component pages were especially well-structured and informative.

7.2.1 The ease of implementation: Dark mode

As mentioned, the ease of implementing dark mode in Chakra UI depended on whether the styles were applied to a component via theme. For components styled via theme, applying mode-dependent styles is straightforward.

[63] https://ant.design/components/button#design-token, visited 1.7.2024

[64] Ant Design GitHub issues: https://github.com/ant-design/ant-design/issues, visited 21.4.2024

Pseudo selectors `_dark` and `_light` can be used in theme object to apply styles only when the dark or light mode is active, respectively [79]. Or, as shown in Listing 12, when for example light mode is treated as default, it is sufficient to use `_dark` selector to apply styles that are different from the default.

```
import { defineStyle, defineStyleConfig } from "@chakra-ui/react";

const solid = defineStyle({
  color: "brand.white",
  bg: "brand.blueBlack",
  border: "none",
  _dark: {
    color: "brand.blueBlack",
    bg: "brand.gray5",
  },
  _hover: {
    bg: "brand.black70",
    _dark: {
      bg: "brand.white71",
    },
  },
});

export const buttonTheme = defineStyleConfig({
  defaultProps: {
    variant: "solid",
  },
  variants: { solid },
});
```

Listing 12: Abbreviated version of styling Button variant based on color theme

Chakra also offers a hook `useColorModeValue` to assign a value based on the active theme[65]. The hook takes two arguments: one for the light mode and one for the dark mode. The value returned by the hook can then be passed to the component's inline styles or style props, as shown in Listing 13. The hook accepts both colors defined via theme and hex color codes as values.

I found this approach cumbersome and used it only four times. As most components (such as buttons) were styled via theme, the need for the hook would mainly arise with layout components such as <Box> or <Flex>. As such components would, at least in the demo application, follow a limited set of text and background colors, using a clearly named CSS variable would be more maintainable and readable than referring to theme colors on the component level and using them inline. But since the hook was the solution offered in the documentation, I utilized it. However, based on migration

[65] https://v2.chakra-ui.com/docs/styled-system/color-mode#usecolormodevalue, visited 26.6.2024

guide for upcoming Chakra UI version 3 [19], the hook will deprecated in favor of **next-themes**[66] or another similar library.

```
export const Toolbar = () => {
  const bg = useColorModeValue("brand.white", "brand.blueBlack");

  return (
    <chakra.header
      bg={bg}
      ...
```

Listing 13: Using `useColorModeValue` in Chakra UI

7.2.2 Ease of implementation: Customizing components

As mentioned in the introduction for this section, customizing components via Chakra UI's theme object was for the most part straightforward. The only issues I encountered were misunderstandings regarding where a specific style should be applied.

For example, when sizing <Modal>, I was trying to add max and min widths and heights via `baseStyle` - section of component styles shared among all variants - but some of those values weren't being applied all. They were not overwritten in the inspector but simply a different value. Looking through GitHub issues, I found that `maxWidth` and some other size-related properties on Modal need to be set via `sizes` key in the component's theme[67]. Coming from a CSS background, this was not obvious to me. Though Modal's documentation contained a section about adding a custom style [68], it did not specify that changing size properties should be done via `sizes` key.

7.2.3 Documentation

Chakra UI's component documentation has a consistent and easy-to-understand structure. Each components page is divided into two or three tabs: Usage, Props, and potentially Theming. Theming (via theme object) is not available for all components, for example, <Grid>, and for such components, the tab is not available. Each of the tabs is broken down into further subsections that are shown as an outline.

Usage sections demonstrate how to bring the component into one's code. Alongside code examples, there are renditions of the actual components. The props section provides a detailed table of component's props, default values, and types. The theming tab includes code examples of how a component can be styled via Chakra UI's theme object. For multipart components, such as <Modal>[69], a list of component parts and

[66] https://www.npmjs.com/package/next-themes, visited 4.7.2024

[67] https://github.com/chakra-ui/chakra-ui/discussions/2676, visited 18.6.2024

[68] https://v2.chakra-ui.com/docs/components/modal/theming#adding-a-custom-size, visited 18.6.2024

[69] https://v2.chakra-ui.com/docs/components/modal/theming, visited 3.5.2024

an "anatomy" picture identifying each part is provided, as seen in Figure 2. Though the names of the component parts are generally descriptive, the anatomy picture is handy for understanding which part of the component to target in theme for the desired styling effect.

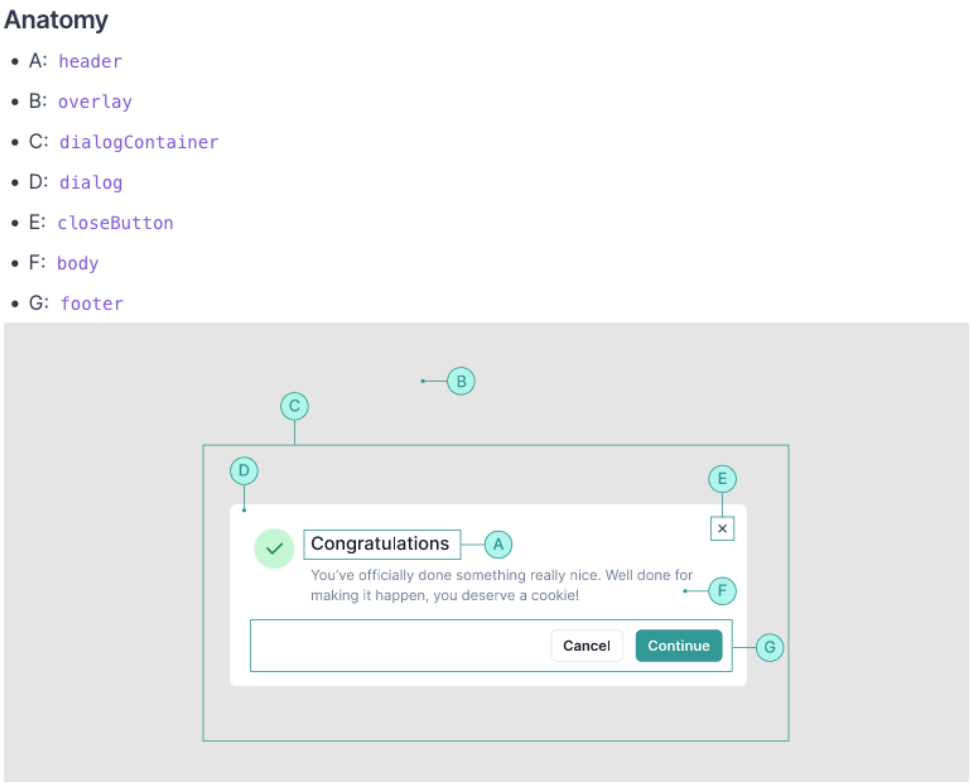


Figure 2: Visualization of Chakra UI's <Modal> component's "Anatomy" as shown in its theming tab (May 2024)

As mentioned, some components do not have a theming tab. Confusingly, at the time of writing, there were also components for which the tab existed but was empty, such as `RangeSlider`[70] and `PinInput`[71]. While these are minor issues, empty documentation sections can cause unnecessary confusion. Does an empty section mean that documentation has been written? Or does the component not support theming? The only way to find out is to try it out - which I did not do as I did not need these components in my project.

7.3 Mantine

The experience with Mantine was initially confusing and frustrating due to the color system and a plethora of styling methods. Due to the many styling methods, it was not immediately apparent to me which methods were alternatives to each other and

[70] https://v2.chakra-ui.com/docs/components/range-slider/theming, visited 3.5.2024
[71] https://v2.chakra-ui.com/docs/components/pin-input/theming, visited 3.5.2024

which were meant to be used together. However, once I had opted out of using Mantine's color system and figured out that CSS Modules were the preferred way to customize components, achieving the design was relatively straightforward. The design requirements were met, and the code was relatively clean and maintainable.

7.3.1 Customizability: Color palette

Mantine's default color palette is far more extensive than the reference application's. Mantine uses **open-color**[72], with some additions [21]. Each of the 14 default colors in the palette have ten shades. The content of the default palette itself did not matter to me as I was coming in with my own. However, defining my palette was more challenging than I had expected.

When defining colors via Mantine's theme object, each color had to be an array of at least ten shades. Since the application had only 25 colors in its palette, this practice would not fit it well. Mantine offers a function `colorsTuple`[73], which allows one to use a single value for all shades of a color. While it solves the issue of having to come up with extra shades, it generates many more CSS variables than I would have expected. For example, defining a color

```
liveRed: colorsTuple("#e90e43")
```

in Mantine's theme would generate the following eight CSS variables shown in Listing 14.

```
--mantine-color-liveRed-text: var(--mantine-color-liveRed-filled);
--mantine-color-liveRed-filled: var(--mantine-color-liveRed-6);
--mantine-color-liveRed-filled-hover: var(--mantine-color-liveRed-7);
--mantine-color-liveRed-light: rgba(233, 14, 67, 0.1);
--mantine-color-liveRed-light-hover: rgba(233, 14, 67, 0.12);
--mantine-color-liveRed-light-color: var(--mantine-color-liveRed-6);
--mantine-color-liveRed-outline: var(--mantine-color-liveRed-6);
--mantine-color-liveRed-outline-hover: rgba(233, 14, 67, 0.05);
```

Listing 14: CSS variables generated by Mantine from colorsTuple

This was unintuitive considering that `colorsTuple` was marketed as a way to "use a single color as the same color for all shades" [21]. Using Mantine's color definitions and functions in the theme generally produced far more CSS variables than I had expected or needed. Defining a primary color via theme would generate 17 CSS variables. While maybe necessary for some use cases, it felt excessive for mine. Thus, I opted to use mainly CSS color variables that I had already defined for the base implementation.

I was not alone in my confusion, as I found a GitHub issue where another user had written [38]: "A hundred different ways to set theme colors, and very few specifics on

[72] https://yeun.github.io/open-color/, visited 26.6.2024
[73] https://mantine.dev/theming/colors/#colorstuple, visited 1.7.2024

how to consume them. - - How do I get the colors? There's no info about using a hook to do it, or anything. Please and thank you."

7.3.2 Customizability: Component customization

The initial challenge in component customization was understanding which method one could and should use. As Mantine offers a multitude of component styling options, such as customizing components directly in the theme, utilizing `variantResolver` to change colors of some components, CSS Module based customization, and more, the choice was not immediately clear.

However, once I encountered documentation recommending CSS Modules as the preferred method, customization turned out to be relatively straightforward. With CSS Modules, each component's styles could be defined in a separate file, and there were no specificity issues or and existing styles could be modified easily. As CSS Modules generate unique class names, there is no risk of style conflicts between components. Customizing component variants was also easy, as one could target a specific variant in CSS using `data-variant` selectors, such as `[data-variant="filled"]` to customize <Button> variant "filled".

7.3.3 Discoverability of documentation

Mantine's documentation is extensive and includes plenty of examples. The structure and discoverability of information could be improved. For instance, on the "Getting started" -page of the documentation, there is a list of the most important documentation pages to read before starting development[74]. But that section, titled "Learn" is the seventh on that page, after, for example, official templates, community templates, framework guides, and a guide for getting PostCSS highlighting to work in the code editor Visual Studio Code. It is next to the section after "Get Help", which might help you find it, but it is still a bit hidden.

Similarly, an info box titled "Consider using CSS Modules first" that recommends and explains the use of CSS Modules, is placed at the bottom of the CSS Modules documentation page[75]. Though, to be fair, the statement that CSS Modules are the preferred way to customize components is stated multiple times elsewhere in the documentation, including on top of the CSS Modules documentation page.

Besides highlighting the most important sections, the documentation could benefit from better connecting related concepts. In "Styles Overview"[76], the first subpage of "Styles" section of the documentation, CSS Modules are presented as a way to customize Mantine's components. However, neither that page nor the linked CSS Modules documentation shows that said CSS modules can be applied to components via theme to have all said components in the app use those styles. Nor did the theme object page. Instead, the instructions for applying CSS Modules to components are

[74] https://mantine.dev/getting-started/#learn, visited 1.7.2024
[75] https://mantine.dev/styles/css-modules/, visited 1.7.2024
[76] https://mantine.dev/styles/styles-overview/#css-modules, visited 1.7.2024

found on "Styles API" -page[77]. Granted, the short components section[78] on the theme object page links to it. Still, a mention of this option on other pages talking about CSS Modules would help users discover the full potential of using CSS Modules with Mantine.

Until I discovered the code examples on the Styles API page, I was wondering whether I was expected to wrap Mantine components in my own components to apply CSS Modules. Or was I supposed to apply the styles without having to import them each time I used a component? This confusion might have been compounded by the fact that I had not, to my memory, ever used CSS Modules with a component library.

Even if one diligently reads through the recommended sections listed in the list in the "Learn" section, all relevant information related to single concepts (such as CSS Modules) is neither in the same place nor clearly linked to each other. Additionally, as there are multiple methods for styling components, it would be beneficial to have a clear comparison of the methods and when to use each.

The difference between pages such as "style prop" and "Style props" might be apparent to someone who has worked with React component libraries a lot. However, for a developer new to not only Mantine but also React or component libraries in general, such titles might not be as clear. The "style prop" page explains how to use prop `style` to apply inline styles to any Mantine components - similar to how `style` prop is used in React components, as described in Section 5.6.1.

These discoverability issues could also be mitigated by efficient search functionality. Unfortunately, the search function of Mantine's documentation usually yielded no results as it seemed to target only the pages' titles, as can be seen in Figure 3.

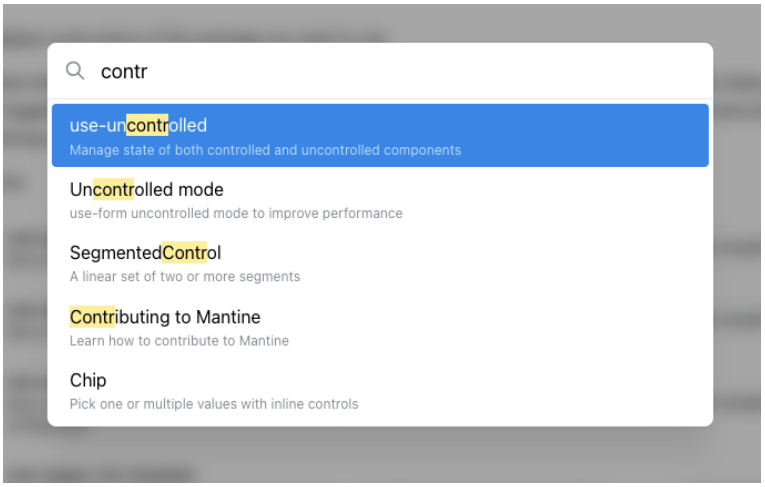


Figure 3: Search results for "contr" in Mantine's documentation (June 2024)

As an alternative, I tried using search engines to find the correct documentation pages. However, as Mantine is on its seventh version and each version's documentation is available on a separate site, the results would often link to older versions of

[77] https://mantine.dev/styles/styles-api/#styles-api-selectors, visited 1.7.2024
[78] https://mantine.dev/theming/theme-object/#components, visited 1.7.2024

documentation. The same issue applied to using Stack Overflow and other online forums to find answers. Thus, when searching for a solution, I alternated between browsing documentation, referring to my previous notes, and searching for answers in the library's GitHub issues and discussions.

7.4 shadcn/ui

While I had never heard of **shadcn/ui** before, I admittedly had doubts and apprehensions about its recommended styling solution, Tailwind CSS. My impression was that Tailwind was a bit too verbose, would hamper separation of concerns and limit styling options, and was mainly a tool for people who did not like writing CSS themselves. But as working with shadcn/ui revealed, Tailwind CSS was not limiting but rather a powerful tool for styling components. I found it intuitive to use as class names were very similar to CSS properties they changed. Though I had to refer to the Tailwind CSS documentation regularly, it was so easy to search and find the information I needed that it did not hinder my workflow.

7.4.1 Customizability via code ownership

As shadcn/ui gives the user ownership of the code, customizations can be applied to actual source code instead of "on top of it". This gives the user more control. There's a sense of security created by the knowledge that changes one makes will not be overwritten by updates - as could be the case when overwriting a traditional component library's styles by overriding CSS classes.

7.4.2 Ease of implementation

Taking shadcn/ui components into use was simple as each component could be generated using the provided command line tool. As one then has the source code, inspecting the components and understanding how they work was easy. Components came with some predefined styles, but with complete control over the style code, it was easy to customize them to fit the design requirements.

When styling component variants that shared some styles - but not in a way that they could be included in the base styles of the components - it seemed initially necessary to repeat the styles for each variant. However, I could reduce the repetition and make the code more readable by using template strings and grouping shared styles in variables (see Listing 15). Whether such a method is the best practice is, of course, debatable, but it suited my workflow and made, in my opinion, the code easier to read.

What was slightly surprising and not apparent from the project's website is that some of the components are imported from Radix UI[79], thus requiring additional dependencies. Specifically, the `<RadioGroup>` component was from Radix UI, and thus, figuring out the typings for its `value` and `onChange` handlers accept, required referring to Radix UI's documentation. The API reference link on the radio group component's documentation page links to Radix UI, but otherwise, the page did not

[79] https://www.radix-ui.com/, visited 1.7.2024

```
const toggleBg = "data-[state=on]:bg-black-9 dark:data-[state=on]:bg-white-17";
const hoverBg = "hover:bg-black-9 dark:hover:bg-white-17";

const toggleVariants = cva(
  `inline-flex gap-2 items-center justify-center text-base
  ...
  `,
  {
    variants: {
      variant: {
        default: `bg-transparent ${toggleBg}`,
        outline: `border-2 border-primary bg-transparent ${hoverBg} ${toggleBg}
          disabled:border-black-15 disabled:dark:border-white-21
          `,
        ghost:
          "focus-visible:ring-offset-0 focus:ring-2 focus:ring-offset-0",
        noHoverGhost: `${toggleBg}`,
        outlineGhost: `border-2 border-primary bg-transparent ${hoverBg}`,
      },
    ...
```

Listing 15: Abbreviated version of shadcn/ui Toggle component's styles

indicate that the component was relying on another library. Though this did not cause any issues, it was a minor inconvenience and would be good to communicate clearly in the documentation.

Despite depending on Radix UI, the radio group component was easy to use and customize. The component consists of several separate elements, making assembling two different radio groups needed for the reference application easy. The only slight struggle was figuring out how to style the indicators based on their type (agree or disagree) without hardcoding the color style into the component. The solution I came up with was to use the `group-` modifier to style the indicators. That allows setting a style based on parent state (such as class). Thus, each option could pass a class (agree or disagree) to the group, which would then style the indicator based on the class. Clearer or more flexible solutions probably exist, but this was sufficient for the use case.

7.4.3 Documentation

The recommended styling solution for shadcn/ui is Tailwind CSS, which has its own documentation. Hence, the Tailwind CSS document was the primary source of information for styling shadcn/ui components and where I spent most of my time. Shadcn/ui's documentation focused on demonstrating, mainly through code snippets, the use of the component code snippets. However, as the components utilized in the application were relatively straightforward, the need for shadcn/ui's documentation was minimal.

Unlike the other libraries, shadcn/ui does not categorize its components by type; it just lists them alphabetically. At the time of writing, the collection consists of 49

components, a manageable amount to browse. Thus, categorization does not seem as necessary.

Where the documentation could be expanded is the dark mode instructions[80]. The section for dark mode focuses on the detection and changing of the mode but does not dwell on how to style the components differently in dark mode. While this depends on the styling solution, it would be beneficial to provide at least some examples of how to style the components differently in dark mode depending on whether the user is using Tailwind CSS or relies solely on CSS variables - which are both suggested methods of styling the shadcn/ui components.

7.5 Use of style sheets

The application's base implementation used bare CSS, defined mainly via style sheets. The application's styles consisted of 33 style sheets containing 762 individual style rules or variable definitions. While the goal of the library implementations was not to get rid of these style sheets, the expectation was that most of the styles would be moved to the theme objects, component styles, or other styling methods used by the libraries, thus reducing the number of original style sheets or even eliminating them. The number of style sheets and individual style rules or variable definitions in each library implementation are summarized in Table 5.

Library	**Style Sheets**			**Rule count**
	Total	**Original**	**New**	
Base implementation	33	-	-	756
Ant Design	26	20	6	477
Chakra UI	2	2	0	27
Mantine	24	17	7	314
shadcn/ui	3	2	1	85

Table 5: Number of style sheets and rules in each version

In the case of shadcn/ui and Chakra UI, the style sheets were indeed almost entirely replaced by library-specific solutions. Only some rules in `index.css` and `reset.css` were kept. **Shadcn/ui** introduced one new style sheet, `global.css`, that was used to define CSS variables, e.g., component colors. Otherwise, all styling could be done via Tailwind CSS utility classes. Of course, some customizations for Tailwind CSS were done via `tailwind.config.ts`.

As the preferred method of styling in **Mantine** was via CSS modules, 16 out of 17 remaining style sheets were converted to CSS modules. Thus those are considered original style sheets in Table 5. Additionally, new style sheets were introduced for Mantine components that needed customization. In the end the project contained 23 CSS Modules and one global style sheet, a simplified version of the original global `index.css`.

[80] https://ui.shadcn.com/docs/dark-mode/vite, visited 1.7.2024

Ant Design version retained the largest number of original style sheets, 20 out of 33. Additionally, six new style sheets were introduced to override individual components' styles. Keeping each component's override in a separate file was for better organization and easier maintenance.

7.6 Screenshot test results

As mentioned in Section 6.6, screenshot tests were created to verify that the library-specific implementations matched the base version. While the tests did not cover all application states (such as hover or focus), they provided a baseline check of component placement, sizing, and color. Each of the three pages, the front page, the questionnaire page, and the candidate page, were tested in both light and dark mode.

Some of the test failures were due to differences so small that they weren't noticeable to the human eye. While it would have been possible to add some tolerance to the tests and allow them to pass with small differences, it was decided to keep the tests strict and instead evaluate the causes by inspecting difference images provided by the `cypress-visual-regression` plugin.

Line-height issues in Ant Design caused some hard-to-debug failures. Though line height had been defined via the theme object, it was being overridden by Ant Design's default styles. The issue was resolved by setting the line height directly in the component styles.

7.7 Summary of results

The key findings in terms of customizability, ease of implementation, and support from documentation of each library are summarized in Table 6. Especially findings about customizability and ease of implementation overlap as the level of customizability affected how easy it is to implement the desired styles.

Additionally, the compiled sizes of the CSS and JS files and the time spent on each implementation are summarized in Table 7. Though neither the sizes nor the time spent are the main focus of this study, they provide some context for how each library handles styles and how much effort was needed to implement the design requirements.

7.7.1 Amount of CSS

The Table 7 shows the amount of minified CSS and JS generated when creating a production build of each version of the application with the command:

```
npm run build
```

It runs TypeScript and Vite build processes:

```
tsc && vite build
```

	Customizability	Ease of implementation	Documentation
Chakra UI	Customizing components via the theme object is flexible and smooth	Dark mode support on component level convenient, otherwise a bit odd but doable	Well-structured, especially component pages. Could use more complex examples, e.g. dark mode.
Mantine	Flexible, CSS modules and component structure allow easy and extensive customization.	Prescriptive and confusing color system. Effective and easy component customization via CSS Modules.	Comprehensive but discoverability and structure could be enhanced.
shadcn/UI	Code ownership and Tailwind offer extensive customizability	Straightforward component usage, full access to source code makes work easy.	Minimal but covers most components. User needs to refer to Tailwind and Radix UI documentation.
Ant Design	Limited, tied to a design system. Not suitable for custom design	Limited token options, token impact sometimes unclear	Good executable examples. Token documentation could be more detailed and structured.

Table 6: Summary of main findings

What stands out from the build sizes is comparatively how much CSS Mantine generates and how little CSS Chakra UI produces. Reasons for the varying build sizes were not investigated as they fall outside the scope of the work. However, I suspect Chakra UI's small amount is due to the library's use of a CSS-in-JS library.

When it comes to Mantine, I think the large amount of CSS variables - mentioned in relation to colors in Section 7.3.1 - might be a contributing factor - or at least an indication of the library's approach to generating plenty of CSS. Still, I doubt just variables would explain the multiple amount of CSS compared to the other libraries.

7.7.2 Time spent on each library

Out of personal interest and the need to manage my time, I had been tracking the time I spent on thesis work using a time-tracking tool Toggl[81]. As I started working on the implementations, I made sure to track which library I was working on and what I was doing. While the tracking certainly is not perfect, it provides a rough idea of how the work effort was distributed between the libraries.

[81] https://toggl.com/, visited 19.6.2024

After base implementation was close to finished, I first started working on the page Chakra UI version, as I expected the familiar library to be the easiest to get started with. However, as I did have issues with the dark mode support (see Section 7.2.1), I also started working on shadcn/ui version alongside Chakra UI and shadcn/ui ended up being the first version to be completed.

Ant Design and Mantine versions were started after Chakra UI and shadcn/ui were mostly ready. Time spent on the Ant Design version is the lowest of the libraries. This was not because the Ant Design version was easier to implement but because of more limited conversion to library-specific components. Due to limitations in Ant Design customization, several components were left as they were in the base implementation.

The Mantine version took the most time to implement, partially due to the extensive documentation and issues I had with the color palette. Due to those issues, I also spent more time writing notes about the library than I did, for example, with shadcn/ui and Chakra UI. Some of this note taking also factored into the development time. Though most of the notes were written outside of the list hours.

Version	Compiled CSS	Compiled JS	Time spent (h)
Base implementation	20.40 kB	295.31 kB	52 hours
Chakra UI	0.75 kB	629.47 kB	24 hours
Mantine	197.68 kB	402.15 kB	26 hours
shadcn/ui	21.86 kB	364.35 kB	23 hours
Ant Design	13.66 kB	810.51 kB	17 hours

Table 7: CSS and JS build sizes and hours spent on each version

8 Discussion

In addition to the aspects discussed in Section 7, I noticed several other common themes arising from many of the library versions. These issues, differences, and considerations will be discussed in this section. Finally, I will address the limitations of the study.

8.1 Naming style and component properties

Both Chakra UI and Mantine generally follow the same naming convention for style props. Props are camelCase versions of CSS properties, such as `bgColor`.

For some properties, Chakra UI uses a shorthand version of the property name, such as `bg` for `backgroundColor`, `maxW` for `maxWidth`. Convenient are the shorthand versions of the margin and padding properties, such as `m` for `margin` and `p` for `padding`. Unlike in CSS, there are shorthand versions for styling only vertical or horizontal margins and paddings, such as `mx` for horizontal margin and `my` for vertical margin. These not only shorten the code slightly but also improve readability.

Whereas Chakra UI shortens style props sometimes, or offers both short and long version of a property name, Mantine tends to shorten all style props [80]. While this produces shorter code, it does at times hinder finding the correct property. Especially with IntelliSense, one may start to type a couple of first letters of a CSS property, such as "max" to get `maxWidth` and `maxHeight`. But Mantine's style props are shortened even further thus yielding no results.

In general I would advocate for using full or almost full property names when matching existing CSS properties. Then developers don't need to learn new abbreviations and can rely on their existing knowledge of CSS properties and thus reduce cognitive load. Of course there is nuance to this issue. Ant Design's decision to use boolean prop `vertical` instead of `flexDirection` with values "row" or "column", can be seen as more intuitive and easier to understand. I would personally prefer to matching CSS properties but can see the merit in Ant Design's approach.

8.2 Automating colors

A common theme in both Ant Design and Mantine seemed to be the attempt to automate some of the color logic. For example, Mantine offers the option to switch button text colors automatically based on a luminance threshold[82] and Ant Design automatically generated a dark theme, as discussed above in Section 7.1.3.

Partial color automation can be convenient and time-saving, especially if the design specifications align with the library's color philosophy or when a developer is working without a designer. When color changes such as different component states are automated, a developer can quickly assemble a visually appealing and accessible - in terms of contrast - application without the need to specify every detail.

[82] https://mantine.dev/theming/theme-object/#luminancethreshold, visited 4.7.2024

However, while such automation is a good starting point, it assumes designs to follow a specific color logic. When adhering to a specific design specification, partial automation might also prove restrictive, as it may not align perfectly with the desired customization, as was the case with the ten shades approach in Mantine. Nevertheless, ensuring that the automated color logic meets accessibility requirements can significantly enhance the overall quality and usability of the services developed.

8.3 Differences in dark mode approaches

All of the component libraries used supported dark mode, some even automatically. But as discussed in Section 7, with a specific design in mind, the automatic dark mode might not always be the best fit.

When manually implementing dark mode, I found it easier to use the simple CSS variable-based approach I utilized in the base implementation. When CSS variables are used to handle theme differences, duplication happens on the variable level and not in component styles, as is the case with Chakra UI's component theming demonstrated in Listing 12. This keeps the component styles shorter and easier to read. Though applications commonly contain just two themes, light and dark, the CSS variables approach can be extended to support more themes.

Of course, the downside is the maintenance overhead of the CSS variables. Depending on the design, if one theme has more color variation, the other will contain more variables than needed. For example, if the border color changes on hover on the light theme but not on the dark theme, both must have a variable defined. Additionally, missing or unused variables are sometimes hard to spot, especially if the application grows and the number of variables increases. Thankfully, for example linting tools, such as Stylelint[83] can help catch such issues, or they can be spotted when inspecting the styles via the browser's developer tools. And even without the help of such tooling, I personally preferred CSS variable-based approach over checking for theme in every component style.

8.4 Semantic HTML

Semantics refers to the meaning of the code in programming. Semantic HTML elements are elements that assign a specific meaning to content in them [70]. For example, the `<nav>` element defines a set of navigation links and `<footer>` defines a footer for a document or section. In contrast, elements such as `<div>` and `<span>` have no specific meaning.

Using HTML elements correctly is especially important for accessibility. Screen readers rely on the HTML structure to provide context and navigate the content [70]. However, using semantic elements - and using them correctly - offers also other benefits. It can improve SEO (Search Engine Optimization) as search engines can utilize the structure of the page to better understand the content [70]. Additionally, it improves readability and maintainability of the code, as the code is easier to understand

[83] https://stylelint.io/, visited 4.7.2024

for other developers - both in an editor and when inspecting the page in the browser's developer tools.

8.4.1 Headings

Using HTML headings correctly is important for accessibility, as headings provide structure to the page, communicate content hierarchy, and allow skimming the page content - both visually and with a screen reader [89]. Heading tags are available in six importance levels from <h1> to <h6>, where <h1> is the most important and <h6> the least important. Headings should be used in order, without skipping levels, and always paired with content relevant to the heading [32].

It is a common mistake that developers will apply headings based on style instead of content structure [32]. Hence, it would be beneficial for a component library to provide ways to separate style from heading rank or, at the very least, not tie the two together.

This is the case for Mantine's heading component <Title>. Its prop `order` (values 1-6, default value 1) sets heading level rendering a specific element <h1> to <h6> [84]. Though setting a heading level applies a visual style, the size of the heading can be changed independent of order with the prop `size`. The `size` can be set to another heading (h1-h6), in which case the component will follow the corresponding font size and line height from the theme. Or if set to any other value, only the font size will change, and line height will be calculated based on the `order`.

8.4.2 Changing rendered HTML element

Some HTML elements have semantic meaning that is important, for example, for accessibility. Beyond headings, lists and other semantic elements that have predefined styles, there are plenty of unstyled semantic elements that can be used to provide additional context and structure to the content, such as <main>, <header>, <footer>, <section>, and <article>. Layout-wise these elements are often comparable to but provide more meaning.

Libraries often provide layout components such as <Flex> and <Box> that are by default rendered as elements. However, some libraries provide a way to change the rendered element to a more semantic one, such as the aforementioned <section>. The benefit of using a library-specific component or the bare semantic element is that the library component might have additional features or properties that are not available in the bare element, such as inline style properties. Hence, the option to change the rendered element is a valuable feature for a component library as it allows developers to use semantic elements without sacrificing the features provided by the library.

Both Chakra UI and Mantine provide a way to change the rendered element of their layout components. Chakra UI offers a prop '`as`' that can be used to change the rendered element of all Chakra components [82]. In turn, all Mantine component accept prop `component` for changing the rendered root element [63].

8.5 Choosing between theming and component-level styling

A recurring theme in the experiment was choosing where to apply styles: was a given component style a new variant, a one-off use case, or even a new component? For example, when styling a custom-styled ordered list used to display candidates' election promises, it was unclear whether it was more efficient to create a separate component with its own component-level styles, apply inline styles, or use the theming capabilities of the component library.

In a larger application, such decisions might be easier to make based on reoccurrence of the style or designer's input. But, as the experiment was a small, one-time project, it meant that there were more one-off use cases, no designer or defined design system, and no genuine concern for scalability or long term maintainability.

Defining a new variant offered easier reuse of the styles but also meant component styles were longer and, thus, potentially more challenging to read. As most of the libraries' variants were defined via the theme, they were further away from where a specific style was used. Though such separation can offer a cleaner structure, it can also make it harder to understand the purpose of a particular style. However, that can be mitigated by clearly naming and documenting the styles.

8.6 Limitations

The study had several limitations that should be taken into account when interpreting the results.

8.6.1 One developer's perspective

The primary limiting factor of the study was that relying on a single developer's experience offers, by nature, a limited perspective. Additionally, as mentioned in Section 6.1, the the author has a particular interest in CSS and design, thus not representing most developers' relationship to CSS - as mentioned in Chapter 2.

8.6.2 Simple application

The implemented website was relatively simple, thus limiting the issues that could arise. The set of components that would and could be utilized from component libraries was limited to buttons, radios, cards, modals, and layout and typography components. Though modal and radios do have some complexity, wider variety of components would have tested the limits of customizing libraries' components more thoroughly.

Additionally, due to the small scope of the application and limited period of use, the maintainability of the styles - which is crucial to an actual production application - was not evaluated. The style overwriting needed, for example, to customize Ant Design components, might not stand the test of time. Similarly, had the application been more extensive, more of the styles might have been refactored into reusable components, utility classes, or other reusable solutions. In a small application, there was not as much repetition that would have justified and benefited from such solutions.

Setting the styles directly in the components or their style sheets was more reasonable and straightforward.

8.6.3 Author's time management

Finally, my time management should also be mentioned as a limiting factor. I could have provided deeper insights if I had reserved more time for the library implementations and their analysis. The implementations that were done first likely got more time and effort than the later ones. Additionally, I would have wanted to expand the scope of the application and or at least the screenshot tests further, but due to time constraints, I could not.

When starting the thesis, I had recently switched from working mainly on React projects to working full-time with Vue. This switch had made me a bit rusty with React. In just a few months, I had already grown fond of Vue's scoped styles in component files, which were discussed briefly in Section 5.7. This rustiness and the recent switch to Vue might have affected the time spent on the base implementation and the first library versions.

8.6.4 Conclusion

Due to the limitations mentioned above, the results of the study should be interpreted with caution. The findings can serve as starting points for further research and component library development, but they should not be taken as definitive truths. As mentioned throughout the thesis, best practices and solutions were not always clear from documentation and also depend on the context and the project at hand.

9 Conclusion

Styling web applications is a nontrivial task, and the choice of styling method can significantly impact the developer experience and maintainability of styles. Using third-party component libraries is a common practice and can save time and effort and improve accessibility and overall application quality. While headless libraries that include minimal styling are gaining popularity, pre-styled libraries are still widely used. Assuming that a product has design requirements that differ from the library's default styles, the developer must customize the library's components. Depending on the library, customizability and methods to achieve the desired design can vary greatly. This affects not only the resulting UI and maintainability of the styles but also the developer experience.

9.1 Main findings

This study compared the developer experience of customizing pre-styled component libraries, namely Ant Design, Chakra UI, and Mantine, and a component collection shadcn/ui. By porting the same React application, a simplified clone[84] of Yle's election barometer, to each library, the study aimed to compare how each library's approach to styling impacted developer experience and whether the styling approach was flexible enough to meet the custom design requirements.

In this thesis, I evaluated the developer experience while implementing the application with each library. I documented my impression and challenges encountered throughout the process focusing on three questions:

1. How does the theme implementation and structure impact theming and customization in terms of ease of implementation and reusability of styles?
2. How flexible is the theme to accommodate various design requirements?
3. How do the provided theme documentation, examples, and type definitions (TypeScript) support development?

To verify that each version met the design requirements - originating from the reference application - I used screenshot tests to compare the library versions to the base implementation. The study found that meeting the design requirements was possible with all the studied libraries, but the ease of customization varied considerably between the libraries. Namely, forcing Ant Design to meet the design requirements required using CSS specificity to override the library's styles.

I found that the developer's experience of customizing pre-styled component libraries that are meant to be customized is generally positive. Out of the compared libraries, Ant Design, intended to follow its own design language, was the most challenging to customize.

Beyond Ant Design's limited customization options, the main challenges were understanding the possible and especially recommended ways to customize the libraries.

[84] https://github.com/Rikkokiri/thesis-base

Additionally, while all libraries supported dark mode, the breath of documentation for it and ease of customizing the dark mode's color palette varied between the libraries.

Ultimately, customizability and styling methods of component libraries are just some of the many factors one needs to consider when choosing a library. This study did not evaluate, for example, the accessibility, performance, or usability of the libraries. The findings of this study can help developers and maintainers of component libraries understand how their design decisions and documentation structure can impact developer experience. Improving documentation structure, providing more code examples, and explaining best practices for styling could meaningfully improve developers' experience. As practices between libraries vary, offering guidance on a given library's philosophy can ease the learning curve.

9.2 Future work

This work provides an initial comparison of the developer experience of customizing pre-styled component libraries. However, due to the study's limited scope, there are several directions for future work. A similar study could be configured with a larger group of developers to include a variety of perspectives and experience levels. Ideally, the study would also include a longer time frame to evaluate the maintainability of the styling solutions in a team setting.

This study intentionally focused on the customization of pre-styled libraries. However, headless libraries seem to be gaining popularity, and comparing them to each other or pre-styled libraries could provide insights to inform the design of future component libraries. While headless libraries do not include any styles, they could still include design decisions that affect styling solutions or prefer a particular method. For example, the component library Headless UI by Tailwind Labs strongly recommends using Tailwind CSS for styling. Additionally, some libraries, such as Mantine, can contain styles but can also be used as headless libraries. Such libraries could be compared to themselves by using them with and without the provided styles.

References

[1] *[JSConfUS 2013] Tom Occhino and Jordan Walke: JS Apps at Facebook*. Aug. 2013. URL: https://www.youtube.com/watch?v=GW0rj4sNH2w (Accessed on 3/7/2024).

[2] *10 Best React UI Frameworks to Use in 2023*. URL: https://clouddevs.com/react/10-best-react-ui-frameworks-to-use-in-2023/ (Accessed on 6/4/2024).

[3] *7 Reasons To Use UI Component Libraries To Style Web Apps*. June 2023. URL: https://www.sencha.com/blog/7-reasons-to-use-ui-component-libraries-to-style-web-apps/ (Accessed on 3/4/2024).

[4] *A brief history of CSS until 2016*. URL: https://www.w3.org/Style/CSS20/history.html (Accessed on 25/4/2024).

[5] *About Mantine | Mantine*. URL: https://mantine.dev/about/ (Accessed on 26/3/2024).

[6] Durga Prasad Acharya. *React UI Components Libraries: Our Top Picks*. Mar. 2024. URL: https://kinsta.com/blog/react-components-library/ (Accessed on 18/3/2024).

[7] Vytenis Alaburda. *Why Is React a Library and Next.js a Framework? | Built In*. July 2023. URL: https://builtin.com/software-engineering-perspectives/react-framework (Accessed on 5/3/2024).

[8] Christopher. Alexander. *A pattern language : towns, buildings, construction*. Ed. by Sara. Ishikawa and Murray. Silverstein. Oxford University Press, 1977. ISBN: 0-19-501919-9.

[9] Peter Alston, David Walsh, and Gary Westhead. "Uncovering "Threshold Concepts" in Web Development: An Instructor Perspective". In: *ACM Transactions on Computing Education (TOCE)* 15 (1 Mar. 2015). ISSN: 19466226. DOI: 10.1145/2700513. URL: https://dl-acm-org.libproxy.aalto.fi/doi/10.1145/2700513 (Accessed on 25/4/2024).

[10] Amarachi Amaechi. *Top utility-first CSS frameworks - LogRocket Blog*. URL: https://blog.logrocket.com/top-utility-first-css-frameworks/ (Accessed on 3/4/2024).

[11] *Ant Design - The world's second most popular React UI framework*. URL: https://ant.design/ (Accessed on 21/4/2024).

[12] Sarah Chima Atuonwu. *Native CSS nesting: What you need to know - LogRocket Blog*. Dec. 2023. URL: https://blog.logrocket.com/native-css-nesting/ (Accessed on 19/3/2024).

[13] Kent Beck and Ward Cunningham. "Using Pattern Languages for Object Oriented Programs". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1987. URL: http://c2.com/doc/oopsla87.html.

[14] Tim Berners-Lee. *The original proposal of the WWW, HTMLized*. Mar. 1989. URL: https://www.w3.org/History/1989/proposal.html (Accessed on 25/4/2024).

[15] *Best open source components library for ReactJS? : r/reactjs*. URL: https://www.reddit.com/r/reactjs/comments/107etkc/best_open_source_components_library_for_reactjs/ (Accessed on 18/3/2024).

[16] Tobias Paul Bleisch. "An Empirical Study of CSS Code Smells in Web Frameworks". In: *Master's Theses* (Mar. 2018). DOI: 10.15368/theses.2018.105. URL: https://digitalcommons.calpoly.edu/theses/1842 (Accessed on 24/2/2024).

[17] *Button - Ant Design*. URL: https://ant.design/components/button (Accessed on 1/7/2024).

[18] *Chakra UI*. URL: https://v2.chakra-ui.com/ (Accessed on 5/5/2024).

[19] *chakra-ui/MIGRATION.md at f6517fbdddcecac1a21910ac7259603d93ff96ce - chakra-ui/chakra-ui*. URL: https://github.com/chakra-ui/chakra-ui/blob/f6517fbdddcecac1a21910ac7259603d93ff96ce/MIGRATION.md#color-mode (Accessed on 4/7/2024).

[20] *CLI - shadcn/ui*. URL: https://ui.shadcn.com/docs/cli (Accessed on 22/4/2024).

[21] *Colors | Mantine*. URL: https://mantine.dev/theming/colors/ (Accessed on 26/6/2024).

[22] Chris Coyier. *CSS Is, In Fact, Awesome*. Apr. 2021. URL: https://css-tricks.com/css-is-in-fact-awesome/ (Accessed on 1/6/2024).

[23] *Creating and nesting components | Quick Start – React*. URL: https://react.dev/learn#components (Accessed on 5/3/2024).

[24] *CSS: Cascading Style Sheets | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (Accessed on 19/3/2024).

[25] *Customize Theme - Ant Design*. URL: https://ant.design/docs/react/customize-theme (Accessed on 21/4/2024).

[26] Fabian Fagerholm and Jürgen Münch. "Developer Experience: Concept and Definition". In: IEEE Press, 2012, pp. 73–77. ISBN: 9781467323529. DOI: 10.5555/2664360.2664372. URL: https://dl-acm-org.libproxy.aalto.fi/doi/10.5555/2664360.2664372 (Accessed on 25/4/2024).

[27] Therese Fessenden. *Design Systems 101*. Apr. 2021. URL: https://www.nngroup.com/articles/design-systems-101/ (Accessed on 19/4/2024).

[28] Brad Frost. *Atomic Design*. June 2013. URL: https://bradfrost.com/blog/post/atomic-web-design/ (Accessed on 21/4/2024).

[29] Brad Frost. *Atomic design*. Brad Frost, 2016. URL: https://atomicdesign.bradfrost.com/ (Accessed on 25/4/2024).

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612. URL: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

[31] Golnaz Gharachorlu. "Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model". In: (2014). DOI: 10.14288/1.0167067. URL: https://open.library.ubc.ca/soa/cIRcle/collections/ubctheses/24/items/1.0167067 (Accessed on 25/2/2024).

[32] Colleen Gratzer. *Mastering Accessible Headings*. Nov. 2023. URL: https://medium.com/design-domination/mastering-accessible-headings-39e3b402a260 (Accessed on 20/6/2024).

[33] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. "What happens when software developers are (un)happy". In: *Journal of Systems and Software* 140 (June 2018), pp. 32–47. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2018.02.041.

[34] Matthew Hague, Anthony W Lin, and C.-H Luke Ong. "Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach". In: *ACM SIGPLAN Notices* 50 (10 Oct. 2015), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/2858965.2814288. URL: https://dl-acm-org.libproxy.aalto.fi/doi/10.1145/2858965.2814288 (Accessed on 31/1/2024).

[35] Ferenc Hámori. *The History of React.js on a Timeline*. May 2024. URL: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ (Accessed on 4/4/2024).

[36] *History of JavaScript - GeeksforGeeks*. URL: https://www.geeksforgeeks.org/history-of-javascript/ (Accessed on 25/4/2024).

[37] *How A Small Team of Developers Created React at Facebook | React.js: The Documentary*. Feb. 2023. URL: https://www.youtube.com/watch?v=8pDqJVdNa44&t=528s (Accessed on 4/4/2024).

[38] *How to get theme colors from within a component - mantinedev - Discussion #4629*. URL: https://github.com/orgs/mantinedev/discussions/4629 (Accessed on 26/6/2024).

[39] Yun-Hsuan Huang. "Developing a design system for an e-commerce website". Aalto University, 2019, pp. 1–56. URL: https://aaltodoc.aalto.fi/handle/123456789/38465 (Accessed on 25/4/2024).

[40] Pete Hunt. *Why did we build React?* June 2013. URL: https://legacy.reactjs.org/blog/2013/06/05/why-react.html (Accessed on 27/2/2024).

[41] Pete Hunt, Paul O'Shannessy, Dave Smith, and Terry Coatta. "React: Facebook's functional turn on writing Javascript". In: *Communications of the ACM* 59 (12 Dec. 2016), pp. 56–62. ISSN: 15577317. DOI: 10.1145/2980991. URL: https://dl-acm-org.libproxy.aalto.fi/doi/10.1145/2980991 (Accessed on 25/2/2024).

[42] *Introduction - Ant Design*. URL: https://ant.design/docs/spec/introduce (Accessed on 21/4/2024).

[43] *Introduction - shadcn/ui*. URL: https://ui.shadcn.com/docs (Accessed on 17/3/2024).

[44] Hanna Järveläinen. "Creating a React Native UI component Library". In: (2024). URL: http://www.theseus.fi/handle/10024/844290 (Accessed on 30/3/2024).

[45] Laura Kalbag. *Design Systems*. Dec. 2012. URL: https://24ways.org/2012/design-systems/ (Accessed on 21/4/2024).

[46] Antti Kallonen. "Responsiiviset web-käyttöliittymäkirjastot". Tampereen teknillinen yliopisto, Nov. 2017. URL: https://trepo.tuni.fi/handle/123456789/25211 (Accessed on 16/3/2024).

[47] Karthik Karki. *What is Atomic Design?* Aug. 2023. URL: https://medium.com/@karkiajith/what-is-atomic-design-2f157d258934 (Accessed on 6/5/2024).

[48] Lea Katalina Kivinen. "Working As A Full Stack Developer In A Large Project : Working Diary". In: (2016). URL: http://www.theseus.fi/handle/10024/118337,%20https://urn.fi/URN:NBN:fi:amk-2016112817977 (Accessed on 25/4/2024).

[49] Ruskin Kollin. "Lived Experience of a Junior Software Developer". In: (Feb. 2022). URL: http://www.theseus.fi/handle/10024/711292 (Accessed on 23/4/2024).

[50] Antti Kopperi. "CSS-ohjelmistokehysten vertailu". Häme University of Applied Sciences, Oct. 2018. URL: http://www.theseus.fi/handle/10024/153074 (Accessed on 5/4/2024).

[51] Majida Laaziri, Khaoula Benmoussa, Samira Khoulji, Kerkeb Mohamed Larbi, and Abir El Yamami. "Analyzing bootsrap and foundation font-end frameworks : a comparative study". In: *International journal of electrical and computer engineering (Malacca, Malacca)* 9 (1 2019), p. 713. ISSN: 2088-8708. DOI: 10.11591/ijece.v9i1.pp713-722.

[52] Danilo Leal, Michał Dudak, Colm Tuite, and Olivier Tassinari. *An exciting year ahead for Base UI*. Feb. 2024. URL: https://mui.com/blog/base-ui-2024-plans/ (Accessed on 7/4/2024).

[53] Sam Magura. *Why We're Breaking Up with CSS-in-JS*. Oct. 2022. URL: https://dev.to/srmagura/why-were-breaking-up-wiht-css-in-js-4g9b (Accessed on 19/3/2024).

[54] Philip M. Marden and Ethan V. Munson. “Today’s style sheet standards: The great vision blinded”. In: *Computer* 32 (11 Nov. 1999), pp. 123–125. ISSN: 00189162. DOI: 10.1109/2.803645. URL: http://ieeexplore.ieee.org/document/803645/ (Accessed on 25/2/2024).

[55] Davood Mazinanian. “Eliminating Code Duplication in Cascading Style Sheets”. In: (Nov. 2017). URL: https://spectrum.library.concordia.ca/id/eprint/982965/ (Accessed on 31/1/2024).

[56] Davood Mazinanian and Nikolaos Tsantalis. “CSSDev: Refactoring duplication in cascading style sheets”. In: *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017* (2017), pp. 63–66. DOI: 10.1109/ICSE-C.2017.7.

[57] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. “Discovering refactoring opportunities in cascading style sheets”. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* 16-21-November-2014 (Nov. 2014), pp. 496–506. DOI: 10.1145/2635868.2635879. URL: https://dl-acm-org.libproxy.aalto.fi/doi/10.1145/2635868.2635879 (Accessed on 31/1/2024).

[58] Chidume Nnamdi. *The top React UI libraries and kits in 2023 - LogRocket Blog*. Oct. 2023. URL: https://blog.logrocket.com/top-react-ui-libraries-kits/ (Accessed on 6/4/2024).

[59] *Our Work Here is Done - The Web Standards Project*. Mar. 2013. URL: https://www.webstandards.org/2013/03/01/our-work-here-is-done/index.html (Accessed on 25/4/2024).

[60] *Overview - Base UI*. URL: https://mui.com/base-ui/getting-started/ (Accessed on 7/4/2024).

[61] Williams Peter. *Top 10 React UI Component Libraries for 2023*. Jan. 2023. URL: https://medium.com/@ghostsmaw/top-10-react-ui-component-libraries-for-2023-b42a3eeb4ae7 (Accessed on 6/4/2024).

[62] *PodRocket - A web development podcast from LogRocket: CSS nesting with Adam Argyle*. URL: https://podrocket.logrocket.com/css-nesting-adam-argyle (Accessed on 21/5/2024).

[63] *Polymorphic components | Mantine*. URL: https://mantine.dev/guides/polymorphic/ (Accessed on 26/6/2024).

[64] Leonard Punt, Sjoerd Visscher, and Vadim Zaytsev. “The A?BA pattern: Undoing style in css and refactoring opportunities it presents”. In: *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (Jan. 2017), pp. 67–77. DOI: 10.1109/ICSME.2016.73.

[65] Vincent Quint and Irne Vatton. “Editing with style”. In: ACM Press, 2007, p. 151. ISBN: 9781595937766. DOI: 10.1145/1284420.1284460. URL: http://dl.acm.org/citation.cfm?doid=1284420.1284460 (Accessed on 3/4/2024).

[66] *React CSS*. URL: https://www.w3schools.com/react/react_css.asp (Accessed on 19/3/2024).

[67] *React UI Frameworks and Component Libraries in 2023*. June 2023. URL: https://dev.to/fireartd/react-ui-frameworks-and-component-libraries-in-2023-236b (Accessed on 6/4/2024).

[68] *Releases · mantinedev/mantine*. URL: https://github.com/mantinedev/mantine/releases (Accessed on 21/4/2024).

[69] *Resources - Ant Design*. URL: https://ant.design/docs/resources (Accessed on 21/4/2024).

[70] *Semantics - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. Aug. 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Semantics (Accessed on 1/7/2024).

[71] *SFC CSS Features | Vue.js*. URL: https://vuejs.org/api/sfc-css-features.html (Accessed on 19/4/2024).

[72] *SFC Syntax Specification | Vue.js*. URL: https://vuejs.org/api/sfc-spec.html (Accessed on 19/4/2024).

[73] *shadcn*. URL: https://shadcn.com/ (Accessed on 8/4/2024).

[74] *shadcn (shadcn)*. URL: https://github.com/shadcn (Accessed on 22/4/2024).

[75] *shadcn-ui - npm*. URL: https://www.npmjs.com/package/shadcn-ui (Accessed on 17/3/2024).

[76] Hamza Shahab. *9 React component libraries for efficient development in 2024*. Dec. 2023. URL: https://ably.com/blog/best-react-component-libraries (Accessed on 18/3/2024).

[77] James Sinkala. *Understanding Vue.js Single-File Components (SFCs)*. Mar. 2023. URL: https://vuenoob.com/posts/understanding-vue-js-single-file-components-sfcs/ (Accessed on 19/4/2024).

[78] *Stack Overflow Developer Survey 2021*. URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe (Accessed on 27/2/2024).

[79] *Style Props - Chakra UI*. URL: https://v2.chakra-ui.com/docs/styled-system/style-props#pseudo (Accessed on 4/7/2024).

[80] *Style props | Mantine*. URL: https://mantine.dev/styles/style-props/ (Accessed on 26/6/2024).

[81] *Styling – Radix Themes*. URL: https://www.radix-ui.com/themes/docs/overview/styling (Accessed on 7/4/2024).

[82] *The as prop and Custom component - Chakra UI*. URL: https://v2.chakra-ui.com/community/recipes/as-prop (Accessed on 4/7/2024).

[83] *Themes - shadcn/ui*. URL: https://ui.shadcn.com/themes (Accessed on 22/4/2024).

[84] *Title | Mantine*. URL: https://mantine.dev/core/title/ (Accessed on 26/6/2024).

[85] *Usage Statistics of CSS for Websites, July 2024*. URL: https://w3techs.com/technologies/details/ce-css (Accessed on 1/7/2024).

[86] *Using CSS nesting - CSS: Cascading Style Sheets | MDN*. May 2024. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_nesting/Using_CSS_nesting (Accessed on 21/5/2024).

[87] Natalia Vetter-Neo. *When To Use A Ui Component Library In A React Project? | Sunscrapers*. Feb. 2020. URL: https://sunscrapers.com/blog/ui-component-library-in-a-react-project/ (Accessed on 3/2/2024).

[88] *Virtual DOM and Internals – React*. URL: https://legacy.reactjs.org/docs/faq-internals.html (Accessed on 4/3/2024).

[89] *W3 Schools - Accessibility Heading Levels*. URL: https://www.w3schools.com/accessibility/accessibility_heading_levels.php (Accessed on 20/6/2024).

[90] Christine E Wania. "Investigating an author's influence using citation analyses: Christopher Alexander (1964–2014)". In: *Proceedings of the Association for Information Science and Technology* 52 (1 Jan. 2015), pp. 1–10. ISSN: 2373-9231. DOI: 10.1002/PRA2.2015.145052010029. URL: https://onlinelibrary.wiley.com/doi/full/10.1002/pra2.2015.145052010029 (Accessed on 18/4/2024).

[91] *What is a Component Library?* Nov. 2023. URL: https://www.uxpin.com/studio/blog/ui-component-library/ (Accessed on 3/2/2024).

[92] *Why is React So Popular These Days?* Mar. 2023. URL: https://a-team.global/blog/why-is-react-so-popular/ (Accessed on 27/6/2024).

[93] *Writing Markup with JSX*. URL: https://react.dev/learn/writing-markup-with-jsx (Accessed on 5/6/2024).

[94] *Yle vaalikone - Presidentinvaali 2024 - Yle*. URL: https://vaalit.yle.fi/vaalikone/presidentinvaali2024 (Accessed on 5/3/2024).

[95] Andriani Yunita. "Challenges in front-end JavaScript development for web applications — Developers' perspective". In: (Dec. 2023). URL: https://aaltodoc.aalto.fi/handle/123456789/124998 (Accessed on 15/4/2024).

[96] Jeffrey Zeldman. *Designing with Web standards*. 2nd edition. New Riders, 2007. ISBN: 0-321-47771-5.

[97] Jeffrey Zeldman. *Taking Your Talent to the Web: A Guide for the Transitioning Designer*. USA: New Riders Publishing, 2001. ISBN: 0735710732.