ZURICH UNIVERSITY OF APPLIED SCIENCES

BACHELORS THESIS

A New Code Generation Tool for Rapid Application Development

CodeFlow: A developer-friendly code generator for rapid development of maintainable web applications

Patrick Egli, Karim Ibrahim

supervised by
Dr. Michael WAHLER

June 9, 2023

DECLARATION OF ORIGINALITY

Bachelors Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:	Name Student:
Gockhausen, June 9, 2023	Karim Ibrahim
Aadorf, June 9, 2023	Patrick Egli

Contents

1 Introduction **5**
- 1.1 Background and Motivation . . . 6
- 1.2 Objectives and research questions . . . 7
- 1.3 Scope . . . 8
- 1.4 Outline of the thesis . . . 9

2 Background **10**
- 2.1 Low Code Software Development . . . 11
  - 2.1.1 Low Code Software Development . . . 11
  - 2.1.2 Low Code Platforms . . . 11
  - 2.1.3 Low Code Platforms & Software Engineering . . . 11
- 2.2 Existing Comparisons . . . 13
- 2.3 Developer Experience . . . 14

3 Methodology **15**
- 3.1 Evaluation criteria . . . 16
  - 3.1.1 Maintainability . . . 16
  - 3.1.2 Performance . . . 16
  - 3.1.3 Scalability . . . 16
  - 3.1.4 Version Control . . . 17
  - 3.1.5 Reusability . . . 17
  - 3.1.6 Extendability . . . 18
  - 3.1.7 Documentation . . . 18
  - 3.1.8 Vendor Lock-in . . . 18
  - 3.1.9 Deployment . . . 18
  - 3.1.10 Frontend Integration . . . 19
  - 3.1.11 Time to Market . . . 19
- 3.2 Evaluation methods . . . 20
  - 3.2.1 Evaluation of tools . . . 20
  - 3.2.2 How we compare the tools . . . 20

4 Evaluation of existing tools and platforms **22**
- 4.1 Overview of existing platforms . . . 23
- 4.2 Evaluating Existing Platforms on Key Software Development Metrics . . . 24
  - 4.2.1 Evaluation of JHipster . . . 24
  - 4.2.2 Evaluation of OutSystems . . . 25
  - 4.2.3 Evaluation of Mendix . . . 25
  - 4.2.4 Conclusion . . . 26
- 4.3 Limitations of Existing Platforms and the Need for a New Tool . . . 27

5 Design and Implementation **28**
- 5.1 Advantages of Code Generation Tools in Software Development . . . 29
- 5.2 Design . . . 31
  - 5.2.1 Objectives . . . 31
  - 5.2.2 Overview . . . 31
- 5.3 Frameworks and libraries . . . 32
  - 5.3.1 Remix . . . 32

5.3.2 React . . . 32
5.3.3 Prisma . . . 32
5.3.4 Tailwind CSS . . . 32
5.3.5 Storybook . . . 32
5.3.6 TypeScript . . . 32
5.3.7 NX Workspace . . . 33
5.3.8 Docker . . . 33
5.3.9 GitHub Workflows . . . 33
5.4 Architecture and features . . . 34
5.4.1 Libraries . . . 35
5.4.2 Pre-built Applications . . . 37
5.4.3 Deployment Application . . . 38
5.4.4 CLI Application . . . 38
5.5 Usage of CodeFlow . . . 40
5.5.1 Cloning the CodeFlow Repository . . . 40
5.5.2 Setting up the Development Environment . . . 40
5.5.3 Root Workspace and Target Workspace . . . 41
5.5.4 Generating the Target Workspace . . . 41
5.5.5 UI Component Libraries . . . 43
5.5.6 Spinning up the Database . . . 45
5.5.7 Extending the Model . . . 46
5.5.8 Creating Schemas for Input Validation . . . 49
5.5.9 Creating Business Logic . . . 50
5.5.10 Creating a new UI component . . . 53
5.5.11 Combining the business logic with the frontend code . . . 55

6 Results **60**
6.1 Addressed Challenges . . . 62
6.1.1 Maintainability . . . 62
6.1.2 Performance . . . 62
6.1.3 Scalability . . . 63
6.1.4 Version Control . . . 63
6.1.5 Reusability . . . 63
6.1.6 Developer Experience . . . 64
6.1.7 Extendability . . . 64
6.1.8 Documentation . . . 65
6.1.9 Vendor Lock-in . . . 65
6.1.10 Deployment . . . 66
6.1.11 Frontend Integration . . . 66
6.1.12 Time to Market . . . 66
6.2 Practical Results . . . 68
6.2.1 JHipster . . . 68
6.2.2 OutSystems . . . 70
6.2.3 Mendix . . . 72
6.2.4 CodeFlow . . . 74
6.2.5 Comparison . . . 76
6.2.6 Sample Application . . . 76
6.2.7 Comparing the final Applications . . . 77
6.3 Interpretation of the results . . . 80
6.4 Suggestions . . . 81

7 Conclusion **82**
7.1 Summary of the research questions and objectives . . . 83
7.2 Contributions of the study . . . 84
7.3 Limitations of the study . . . 85
7.4 Future Work . . . 86

A New Code Generation Tool for Rapid Application Development

A Comparative Analysis of CodeFlow and other Tools and Platforms.

Karim Ibrahim, Patrick Egli

Abstract

In recent years, there has been a paradigm shift in the realm of software development, primarily induced by the advent of low code / no code platforms. These platforms have introduced an intuitive, user-friendly medium for both developers and non-technical users to build applications, eliminating the need to write code from scratch.

Despite the advantages presented by this approach, the current state of low code / no code platforms exhibits certain constraints. These limitations include vendor-lock in, limited maintainability, reusability, extendability, performance and scalability.
The objective of this research is to probe this void, through a comparative analysis of current low code / no code platforms, emphasizing their respective strengths and weaknesses and subsequently proposing a new tool. This proposed tool seeks to circumvent some of the prevalent limitations while capitalizing on their strengths. The goal is to develop a developer experience (DX)-friendly tool that generates code while allowing full access to the codebase. This simplifies application customization without the need to write code from scratch.
In order to achieve this, an in-depth examination of existing low code platforms and code generators, such as OutSystems, Mendix and JHipster, will be conducted to discern their strengths and weaknesses. This analysis will serve as a foundation for the development of our tool, targeting a more flexible and customizable development environment specifically catered towards DX-friendly low code projects.
The concluding portion of this research will delve into the implementation of our proposed tool, including a detailed description of its architecture, features and usability. Furthermore, the potential applications of this tool and its utility in creating DX-friendly low code projects that meet the modern software development requisites will also be discussed.

In summary, this research aims to augment the ongoing evolution of low code by introducing a new approach that acknowledges the advantages and limitations of existing platforms. Through the creation of a more flexible and customizable code generator tool, it is envisaged to simplify the process for developers to create applications that meet their unique needs and specifications.

Chapter 1

Introduction

This thesis explores the emerging field of code generators and low-code platforms, which are increasingly being used to develop software applications quickly and efficiently. The thesis compares and evaluates existing tools and frameworks and proposes a new tool that aims to address some of the weaknesses of these existing solutions. The research is motivated by the need for more developer-friendly code generator tools that allow users to define their own models and generate code automatically.

The introductory chapter provides an overview of the thesis and introduces the main themes and topics that will be covered in the subsequent chapters. This includes a discussion of the background and motivation for the research, the objectives and research questions, the scope and limitations of the study and an outline of the thesis structure. By the end of this chapter, the reader should have a clear understanding of the purpose and scope of the research and how the subsequent chapters are organized to address the research questions.

1.1 Background and Motivation

As experienced software developers, we have worked on a variety of projects over the years, both individually and in small teams. In our experience, project owners frequently seek rapid results, placing a significant amount of pressure on developers to deliver within tight deadlines, particularly when new requirements or technologies are involved. Backend development typically involves a notable amount of repetitive code, such as the creation of controllers that interact with services and repositories to retrieve or modify data. Considering front-end development which also involves repetitive code, such as the creation of basic components such as buttons, tables and other UI components. These tasks can include the creation of reusable components and the establishment of a cohesive code base.

In many cases, developers may not have a full understanding of the project requirements during the initial stages of development, leading to the creation of unnecessary or poorly-designed code. This can result in increased development time and costs. To mitigate these issues, the use of a code generator capable of creating boilerplate code for both backend and frontend development could significantly improve development efficiency and quality.

The need for high-quality, reusable code is particularly critical in modern software development, where rapid iteration and collaboration are essential components of the development process. A code generator capable of creating well-designed and tested code that is reusable across multiple projects could significantly enhance productivity and reduce the time required for development. Moreover, such a tool would likely facilitate collaboration between developers, as it would provide a standardized codebase and reduce the likelihood of errors or redundancies.

In light of the aforementioned challenges and opportunities, this thesis seeks to explore the current landscape of code generators and low-code platforms and evaluate their suitability for addressing the challenges outlined above. By identifying the strengths and weaknesses of these platforms, this thesis aims to develop a new tool that provides a more developer-friendly, efficient and effective approach to rapid application development.

1.2 Objectives and research questions

The objectives and research questions of this thesis are aimed at addressing the challenges associated with rapid software development and evaluating the suitability of current code generators and low-code platforms in addressing these challenges. By developing a new tool that generates high-quality, reusable code for both backend and frontend development, this thesis aims to enhance development efficiency and reduce the time and cost associated with creating software. The research questions that will be addressed in this thesis include the key challenges associated with rapid software development, the design and implementation of the new tool and the advantages and disadvantages of the tool in comparison to existing code generators and low-code platforms. The results of this research will contribute to the field of software development and provide insights into the potential of software development tools for addressing the challenges associated with rapid software development.

The objectives of this thesis are to:

1. Evaluate the current state of code generators and low-code development platforms and assess their suitability for addressing the challenges associated with rapid software development.
2. Develop a new tool that generates high-quality, reusable code for both backend and frontend development, with a particular focus on reducing repetitive code and enhancing developer experience.
3. Assess the usability and effectiveness of the new tool in comparison to existing code generators and low-code platforms.

To achieve these objectives, the following research questions will be addressed in this thesis:

1. What are the key challenges associated with rapid software development and how do current code generators and low-code platforms address these challenges?
2. How can a new code generator tool be designed and implemented to generate high-quality, reusable code for both backend and frontend development?
3. What are the advantages and disadvantages of the new tool in comparison to existing code generators and low-code platforms and how can these be addressed to optimize its usability and effectiveness?

By addressing these research questions, this thesis aims to make a contribution to the field of software development, particularly with regard to enhancing development efficiency and reducing the time and cost associated with creating high-quality, reusable code.

1.3 Scope

The scope of this thesis is to propose a new code generation tool specifically designed for developing full-stack web applications. The tool primarily focuses on the TypeScript programming language and generates an NX workspace that includes a comprehensive set of libraries and application templates. These resources enable developers to create modern and user-friendly web applications more efficiently.

In terms of data management, the proposed tool integrates the Prisma ORM, which not only handles the database schema but also manages database migrations. Prisma automatically generates a client library based on the provided schema, allowing developers to interact with the database in a type-safe manner. This approach enhances the developer experience and streamlines the database integration process.

For the frontend development, the tool generates a Remix application that utilizes React as the rendering library. Remix is a modern web framework that offers an excellent developer experience by providing robust data loaders and data mutations within a single framework. This integration simplifies the communication between the frontend and backend, resulting in faster and more intuitive web application development.

To enhance the styling capabilities, the proposed tool incorporates Tailwind CSS, a utility-first CSS framework. Tailwind CSS provides a comprehensive set of pre-built components and offers high customization options, allowing developers to create a unique design system for their application. The default configuration of Tailwind CSS is optimized for various use cases, enabling developers to leverage its capabilities out of the box.

To ensure a scalable and maintainable codebase, the entire project is organized within an NX workspace. NX provides a solid foundation for managing large-scale projects, offering efficient code structuring and maintenance.

By encompassing these features, the proposed tool aims to facilitate the development of full-stack web applications by offering a comprehensive set of tools, libraries and frameworks that enhance productivity, maintainability and overall developer experience.

1.4 Outline of the thesis

In Introduction, we provide an overview of the structure of this thesis and summarize the content of each subsequent chapter.

Delving into the topic, the Background chapter explores existing literature and related work in the field of low code and code generation, shedding light on various approaches and available tools.

Moving on to the Methodology chapter, we delve into the intricate details of our evaluation process for both existing tools and platforms as well as our proposed solution. We carefully outline the evaluation criteria used to assess the effectiveness and suitability of each tool and platform.

A crucial chapter in this thesis, the Evaluation of existing tools and platforms offers an in-depth analysis and assessment of different code generators and low-code platforms by applying the evaluation criteria outlined in the previous chapter.

The Design and Implementation chapter delves into the intricacies of designing and implementing the proposed tool, offering a thorough insight into its structure and functionality. Furthermore, we demonstrate the practical application of the tool by showcasing its ability to generate a sample project.

Subsequently, the Results chapter presents the outcomes of our evaluation of the proposed tool, comparing it with existing code generators and low-code platforms. Moreover, we analyze and elaborate on the significance of these findings, offering valuable insights and recommendations for developers.

Finally, the Conclusion consolidates our findings and contributions, summarizing the key takeaways of this thesis. Furthermore, we provide valuable insights into future work and potential enhancements for the proposed tool.

Chapter 2

Background

This chapter serves as a comprehensive review of the existing literature and related work, encompassing various sources and dimensions of software development. Its primary objective is to conduct a critical evaluation of the literature and related work, with a particular focus on fundamental principles in software engineering.

To begin, we aim to illuminate the concept of low-code and its significance in the realm of software engineering. By exploring the definition, characteristics and applications of low-code platforms, we establish a solid understanding of this emerging paradigm.

Furthermore, we address the existing limitations and gaps within the literature concerning low-code platforms. Through a thorough examination of research papers and industry publications, we identify areas where the current knowledge base may fall short, thereby highlighting opportunities for further research and improvement.

Additionally, we delve into the topic of developer experience (DX) and its paramount importance in the field of software engineering. By exploring the role of DX in enhancing productivity, code quality and overall user satisfaction, we underscore the significance of prioritizing the developer experience in software development endeavors.

By thoroughly examining and analyzing this body of knowledge, this chapter establishes a robust foundation for the subsequent developments and advancements in our tool. Readers will gain a comprehensive understanding of the low-code concept, recognize the existing gaps in the literature and appreciate the significance of developer experience in driving successful software engineering practices.

2.1 Low Code Software Development

This section delves into the concept of low-code software development and explores the landscape of low-code platforms. We provide an in-depth analysis of the key principles and characteristics of low-code development, highlighting its significance in the field of software engineering.

2.1.1 Low Code Software Development

Low Code Software Development (LCSD) is a software development paradigm that aims to reduce the amount of code that developers need to write in order to build software applications. The term low-code originated from a market research firm in 2014 and is used by the industry as a way to describe platforms and development environments that enable the creation of applications with minimal hand-coding. LCSD involves the use of tools and platforms that allow developers to build applications without writing a lot of code. The term gained traction in the industry and has since been used to describe the growing ecosystem of platforms that simplify and streamline the software development process. Recent results show that LCSD has a strong relation to model driven engineering (MDE) and can be defined as methods and tools that support the development of software applications by using models as a basis for the development process [1].

2.1.2 Low Code Platforms

Low code platforms or low code application platforms are tools which assist the process of LCSD. In contrast to traditional software development, low code platforms aim to reduce the amount of code that developers need to write in order to build software applications. Low code platforms provide visual design tools and pre-built components and claim to help developers and non-developers create applications more quickly and easily than traditional coding methods. Typically, a low code platform consist of a graphical user interface (GUI) that allows users to perform a variety of tasks, such as creating and editing models, generating code and deploying applications. They are often offered as a PaaS (Platform as a Service) or SaaS (Software as a Service) solution, where developers can use the platform to build applications without having to install or configure any software. While providing a high level of abstraction, enabling users to create applications without prior coding knowledge, low code platforms also provide a low level of control, since developers cannot modify the generated code. Low code platforms are often criticized for their lack of support for software engineering principles, such as maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment. These quality attributes are essential for the long-term operation of a software application and are often neglected by low-code platforms [2].

2.1.3 Low Code Platforms & Software Engineering

Low code platforms are designed to simplify the development process for individuals without programming knowledge. On the other hand, software development is a complex process that requires not only coding expertise but also knowledge of infrastructure, security, performance and many other factors. Both approaches have their advantages and limitations.

While low code platforms are designed to simplify the development process and reduce costs and time-to-market, software engineering is a complex process that requires a broader skill set and offers more flexibility and control over the development process. Interesting in this context is the work of [3] who propose a few principles on how low code platforms should evolve and adapt software engineering principles. While the term low code refers to the reduced amount of code that developers have to write when using low-code platforms, it is important to recognize

that the overall code generated by these platforms may still be substantial. Therefore, it is crucial to emphasize the significance of maintaining a well-designed and manageable codebase, even when leveraging low-code solutions. Ensuring the maintainability and quality of the generated code becomes an important consideration, as it directly impacts the long-term success and scalability of the application.

Effective documentation plays a crucial role in enabling developers to efficiently gain knowledge about the low-code platform being used. Additionally, adhering to the principle of separation of concerns facilitates the creation of reusable and maintainable code, which aligns with fundamental concerns in software engineering.

However, when it comes to open standards and interoperability, certain low-code platforms like OutSystems and Mendix exhibit limitations due to their reliance on proprietary technologies and the potential for vendor lock-in.

Furthermore, the importance of developer experience cannot be understated, as it directly impacts the productivity and motivation of developers. When developers can effectively utilize development tools and find satisfaction in their usage, it can significantly enhance the overall development process.

Interestingly, low-code platforms and software engineering are not mutually exclusive. In fact, they can be utilized in tandem to leverage the advantages of both approaches. While it remains unclear whether these principles alone can fully enable proper software engineering practices, it presents an interesting research topic that could expand the scope of this thesis.

2.2 Existing Comparisons

At the time of writing this thesis, the existing literature on low-code platforms and web application frameworks is limited and often focuses on outlining the advantages of specific tools rather than providing a comprehensive comparison. However, there are notable studies and efforts in this area that contribute to our understanding.

One significant challenge in evaluating low-code platforms is the lack of a universally accepted framework for comparison. While some attempts have been made to propose such frameworks, they are often limited in scope, focusing on a specific set of low-code platforms that share common functionality and services [4]. Additionally, market research firms such as Gartner and Forrester have published reports and comparisons on low-code platforms, but these reports tend to emphasize the business perspective rather than a comprehensive evaluation of technical aspects. Since these reports are not publicly available, they will not be discussed in this thesis.

To summarize the existing literature, we have identified two key points that are relevant to our thesis:

- The comparison of low-code platforms is often limited to a specific set of platforms that share common functionality and services.
- There is currently no widely adopted and democratized framework that defines how to compare low-code platforms comprehensively.

These points highlight the need for further research and investigation in the field of low-code platforms to establish a more comprehensive framework for evaluation and comparison.

2.3 Developer Experience

This section will explore the concept of Developer Experience (DX) and its influence on the development of applications, drawing on current research in the field.

The term Developer Experience has been introduced in 2012 by Munch & Falgerhom and has been identified by Lethbridge as a key subtopic of the user experience in software engineering today [5]. DX is a broad term that encompasses all aspects of a developer's experience, from the tools and technologies they use to the quality of the documentation and support available to them. For example considering integrated development environments (IDE) features such as syntax highlighting, command suggestion and inline documentation help developers to build new software. When considering the quality of the documentation and support, the availability of tutorials, forums and documentation is important.

DX has become an increasingly important consideration in software development as the demand for user-friendly and efficient tools continues to grow. When developers have a positive experience, they are more likely to be productive, engaged and innovative, which ultimately leads to better software products [6]. In the context of low code / no code development, DX is especially important because these platforms are designed to make app development accessible to a wider range of users. A thesis by Dahlberg has found out that more productivity and focus on the task leads to a positive DX while poor documentation, unsafe and suboptimal collaboration features and limitations in contrast to traditional development lead to a negative DX [7].

When considering the interests of a software developer, many low code platforms miss fundamental software engineering concerns. For example, automated testing, version control and separation of concerns are key concepts which help to extend and maintain software.

Literature research has shown that the term DX is ambiguous. Munch & Falgerhom describe DX on a meta level based on cognition, conation and effect. Lethbridge on the other hand describes DX as a combination of the user experience and the software engineering experience. To clarify the term DX, we will use the definition of Lethbridge and define DX as a combination of the user experience and the software engineering experience.

Therefore we will define that a positive DX is a combination of a positive user experience and a positive software engineering experience. A positive user experience is defined as a positive experience of the user interface and the user interaction with the software. Moreover, a positive software engineering experience is defined as a positive experience of the software engineering process and the tools used to build the software. The software engineering experience is positively influenced by the ability to use the tools to build software, the ability to extend and maintain the software as well as the tools being used.

On the other hand, a negative DX is a combination of a negative user experience and a negative software engineering experience. A negative user experience is defined as a negative experience of the user interface and the user interaction with the software. Similarly, a negative software engineering experience is defined as a negative experience of the software engineering process and the tools used to build the software.

Chapter 3

Methodology

In this chapter, we present the evaluation criteria and methodology used to assess the software development platforms and tools examined in this thesis. The chosen evaluation criteria are derived from the fundamental principles of software engineering, ensuring a systematic and well-grounded approach to our analysis.

The methodology section outlines the specific procedures adopted for evaluating both the prototype under investigation and existing low-code platforms or code generation tools. This step-by-step explanation provides clarity on the process by which our analysis is conducted, ensuring transparency and reproducibility in our scientific endeavor.

By adhering to rigorous evaluation criteria and employing a well-defined methodology, we aim to provide reliable and comprehensive insights into the strengths, weaknesses and capabilities of the evaluated platforms and tools.

3.1 Evaluation criteria

This section describes the evaluation criteria used to evaluate the considered tools. We identified the following evaluation criteria.

3.1.1 Maintainability

Maintainability refers to the ease with which a software tool can be updated or fixed. A tool with high maintainability exhibits clear and understandable code, as well as a well-structured design that enables developers to comprehend and modify its functionality as necessary. This encompasses the ability to update the tool to address bugs, enhance performance or introduce new features.

Maintainability is a crucial aspect of software engineering, as it contributes to the long-term sustainability and evolution of the tool. By prioritizing maintainability, developers can streamline the process of making necessary modifications and ensure the tool remains adaptable to changing requirements over time.

3.1.2 Performance

Performance evaluation assesses the runtime performance of the tool, encompassing factors such as speed, responsiveness and resource consumption. A high-performance tool operates rapidly and efficiently, minimizing lag time and optimizing resource utilization.

The runtime performance of a tool is particularly critical for applications that require real-time or mission-critical functionality. In such scenarios, even minor delays or inefficiencies can result in significant consequences. By evaluating the performance of a tool, developers can ensure its suitability for time-sensitive or high-demand environments.

A high-performance tool not only delivers faster results but also enhances the overall user experience. With quick response times and efficient resource management, users can interact with the tool seamlessly, leading to increased productivity and satisfaction.

Considering the runtime performance of a tool during evaluation enables developers to make informed decisions, selecting tools that meet the performance requirements of their specific use cases. By prioritizing performance, developers can create robust and responsive software applications that deliver optimal results.

3.1.3 Scalability

Scalability evaluation focuses on the tool's ability to handle growing workloads and its potential for expansion to accommodate increased demands. A scalable tool maintains effective performance even as the software system it is building grows in size or complexity.

Scalability is a crucial consideration in software development, as applications often experience varying levels of usage and data volume. A tool with good scalability can seamlessly adapt to increased workloads without compromising performance or stability. It allows for the efficient allocation of resources, such as processing power and memory, to ensure optimal execution even under high-demand conditions.

Considering the scalability of a tool during evaluation helps developers select tools that align with their long-term objectives and can support the growth and evolution

of their software systems. By choosing scalable tools, developers can build applications that can effectively handle expanding workloads, ensuring performance and reliability as the system grows.

3.1.4 Version Control

Version control is an important aspect that assesses how effectively a tool manages different versions or iterations of the software. Robust version control functionality enables developers to track changes, facilitate collaboration among team members and ensure the integrity and stability of the codebase.

A tool with reliable version control allows developers to easily monitor and document modifications made to the software over time. This includes the ability to track individual changes, view revision history and compare different versions of the code. Effective version control enables developers to understand the evolution of the software, identify the contributors to specific changes and revert to previous versions if needed.

Furthermore, version control facilitates seamless collaboration among team members. It enables multiple developers to work on the same codebase simultaneously, managing concurrent changes and merging them efficiently. This ensures that everyone is working on the latest version of the software and minimizes conflicts or inconsistencies.

Additionally, version control plays a crucial role in maintaining code quality and stability. It provides a safety net by allowing developers to roll back to a known working state in case of unforeseen issues or regressions. This ability to revert to a previous version helps mitigate risks and ensures that the software remains reliable and functional throughout its lifecycle.

In summary, strong version control capabilities are essential for effective software development. They enable developers to track changes, collaborate seamlessly and maintain code integrity. By utilizing tools with robust version control, teams can streamline their development processes, improve productivity and ensure the successful management of software iterations and releases.

3.1.5 Reusability

The reusability of a tool refers to its capability to be employed multiple times across different projects or purposes. A reusable tool possesses components or modules that can be leveraged in various contexts, promoting efficiency and cost-effectiveness in software development.

A tool with high reusability allows developers to extract and repurpose specific functionalities or components, eliminating the need to reinvent the wheel for each new project. By utilizing reusable components, developers can save time and effort by building upon established, tested and reliable building blocks. This not only accelerates development timelines but also enhances the overall quality and consistency of the software.

In conclusion, a tool with high reusability provides significant benefits to the software development process. It empowers developers to leverage existing components, promote consistency and improve efficiency. By embracing reusability, teams can optimize their development efforts, reduce duplication of work and foster collaboration and knowledge sharing within the organization.

3.1.6 Extendability

The extensibility of a tool relates to its ability to accommodate the addition of new features or capabilities in a seamless and non-disruptive manner. An extendable tool is designed with modularity in mind, allowing for the incorporation of new functionalities without compromising the existing system.

An extendable tool adopts a modular architecture, where components are designed to be loosely coupled and independent. This design approach enables developers to introduce new features or extend the tool's functionality by simply adding or modifying specific modules, without the need for extensive rework or impacting the existing system. The modularity of the tool ensures that changes or enhancements can be made in isolation, minimizing the risk of unintended side effects or system-wide disruptions.

In summary, an extendable tool provides the flexibility and agility required to accommodate the addition of new features and capabilities. By embracing modularity and providing well-defined extension points, the tool enables developers to extend its functionality without disrupting the existing system. This promotes adaptability, encourages collaboration and ensures the tool can evolve and meet the changing needs of its users.

3.1.7 Documentation

The quality of documentation for a tool encompasses its comprehensiveness, clarity and usefulness. Well-crafted documentation provides detailed explanations of the tool's functions, offers practical examples on how to utilize it effectively and provides solutions to common challenges or issues.

Comprehensive documentation encompasses a wide range of aspects, covering the tool's features, functionalities and usage scenarios. It offers in-depth descriptions of each component, module or function, allowing users to gain a thorough understanding of the tool's capabilities. Furthermore, comprehensive documentation may include tutorials, guides or step-by-step instructions that demonstrate how to utilize the tool in various contexts. This enables users, especially newcomers, to quickly grasp the tool's functionalities and utilize them efficiently.

3.1.8 Vendor Lock-in

This relates to the degree of dependency users have on a specific vendor for services and products. A tool with high vendor lock-in indicates that users will encounter challenges when attempting to switch to a different product or vendor, often incurring significant transition costs.

This aspect also considers the potential limitations and constraints imposed by vendor lock-in. When users heavily rely on a specific vendor, they may face challenges such as limited customization options, lack of flexibility and potential restrictions on integrating with external systems or technologies. Additionally, vendor lock-in can impact the long-term viability and sustainability of the tool, as users may become reliant on the vendor's continued support and availability. Therefore, it is crucial to carefully evaluate and assess the level of vendor lock-in associated with a tool, taking into account the potential implications and risks it may pose to future scalability, adaptability and vendor independence.

3.1.9 Deployment

This aspect evaluates the deployment capabilities of the software created with the tool in a live environment. It encompasses considerations such as the tool's compatibility with different systems and platforms, the ease of setting up and configuring

the software and the efficiency of the update process once the software has been deployed. A tool with good deployment features will facilitate smooth and seamless transitions from development to production, ensuring that the software can be readily installed and used in a live environment. This includes providing robust mechanisms for system compatibility, straightforward setup procedures and streamlined update mechanisms to ensure that the deployed software remains up-to-date and maintainable throughout its lifecycle.

3.1.10 Frontend Integration

The frontend integration capabilities of a tool encompass its ability to seamlessly connect backend services with frontend applications. The ease and simplicity of the integration process directly impact the efficiency and effectiveness of the tool.

A tool with robust frontend integration capabilities streamlines the process of integrating backend functionality into the frontend, allowing for smooth communication and data exchange between the two layers. This facilitates the development of dynamic and interactive user interfaces that seamlessly interact with the backend services. An efficient frontend integration process ensures that developers can easily consume backend APIs, access data and implement business logic within the frontend application. It minimizes the complexity and effort required to establish connections, handle data formats and manage communication protocols.

By providing seamless and straightforward frontend integration, a tool empowers developers to build sophisticated frontend applications that leverage the full potential of backend services, resulting in enhanced user experiences and efficient software development processes.

3.1.11 Time to Market

Time to market is a critical metric that measures the duration it takes to develop and deploy a software product that fulfills user requirements. It plays a pivotal role in determining the success of a product and is a significant consideration for organizations when assessing various tools and technologies.

The faster a software product can be developed and brought to market, the greater the competitive advantage for the organization. Shorter time to market enables companies to seize business opportunities, respond rapidly to changing market demands and gain an edge over competitors. It allows organizations to deliver value to customers promptly, satisfy their needs and establish a strong market presence.

3.2 Evaluation methods

This section describes the evaluation methods used to evaluate the considered platforms, tools and our proposed solution. The evaluation methods are based on the evaluation criteria defined in section 3.1.

3.2.1 Evaluation of tools

The evaluation is based on the evaluation criteria defined in section 3.1. For this thesis we have chosen to evaluate existing frameworks and tools.

Present work such as [2] and [4] have already evaluated existing low code platforms based on business evaluation criteria. In contrast, this thesis will focus on the technical evaluation criteria. The technical evaluation criteria are based on the fundamental principles in software engineering, as stated in 3.1 We do not evaluate the business centered evaluation criteria, such as the ability to create a business application without programming knowledge. The reason for this is that we are not evaluating low code platforms and frameworks for business users, but for software engineers.

3.2.2 How we compare the tools

The evaluation of our prototype and the selected existing tools follows a systematic approach, ensuring a comprehensive and objective analysis of their capabilities. To facilitate this evaluation, we have developed a test application called TeamUp, which serves as a dedicated platform for assessing the functionalities and performance of the tools. TeamUp is a collaborative platform designed to facilitate the sharing of scholarly projects and foster interaction among users. It provides a valuable avenue for connecting students with similar interests, making it particularly advantageous for cross-curricular collaborations.

In addition to evaluating the features and capabilities of the tools, we also consider the time-to-market aspect. To do so, we set a fixed time frame of 4 hours for developing the TeamUp application using each tool. This enables us to assess how efficiently and effectively each tool enables us to deliver a functional product within a specific time frame, reflecting the real-world pressures and demands of software development projects.

Throughout the evaluation process, we have carefully built the TeamUp application using our prototype and the selected existing tools. This step-by-step evaluation approach allows us to thoroughly assess various aspects of the tools such as their performance, scalability, extensibility and maintainability. By employing this structured methodology, we aim to provide an objective and comprehensive evaluation of the tools suitability and effectiveness in meeting the requirements of modern software development projects.

Implementation of TeamUp

TeamUp is a collaborative platform designed to facilitate the sharing of scholarly projects and foster interaction among users. It provides a valuable avenue for connecting students with similar interests, making it particularly advantageous for cross-curricular collaborations.

Creating TeamUp on each platform allows us to evaluate the tools based on the evaluation criteria defined in section 3.1. For each tool we set a fixed time frame of 4 hours.

Evaluation of Criteria

The evaluation process assesses each tool based on the predetermined criteria: Maintainability, Performance, Scalability, Version Control, Reusability, Extendability, Documentation, Vendor Lock-in, Deployment, Frontend Integration and Time-to-Market. These criteria serve as benchmarks for evaluating the strengths and weaknesses of each tool. Throughout the development and testing stages, observations and measurements are collected to assign scores to each tool, providing a quantitative assessment of their quality.

Interpretation and Suggestions

The evaluation process concludes with an interpretation of the results and the formulation of practical recommendations for developers. This systematic and objective evaluation methodology ensures a fair and comprehensive comparison between the proposed prototype, OutSystems, Mendix and JHipster, enabling us to draw meaningful conclusions about the performance and capabilities of each tool. The insights gained from this evaluation provide valuable guidance for developers in selecting the most suitable tool for their specific project requirements.

Chapter 4

Evaluation of existing tools and platforms

This chapter provides a comprehensive analysis of some existing platforms and tools that are currently available for code generation and software development automation with low-code platforms. The purpose of this chapter is to critically evaluate these tools and platforms in terms of fundamental principles in software engineering, such as maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment. By leveraging the insights and findings from this review, we aim to develop a robust and effective tool that addresses the current challenges and opportunities in software development.

4.1 Overview of existing platforms

The field of low-code and no-code platforms has been rapidly growing in recent years and it can be overwhelming to keep track of all the available options. In this chapter, we provide an overview of some of the most popular platforms in this space. As we cannot cover all existing platforms, we have selected a few that we believe are representative of the current state of the field. Our focus will be on OutSystems, Mendix and JHipster. Our selection of these platforms and frameworks was informed by the need to contrast their differences effectively.

JHipster

JHipster is a development platform to quickly generate, develop and deploy modern web applications and microservice structures [8]. The generated application code can then be used as a starting point for further development. The code base generated by JHipster is based on the Spring Boot framework, which is a popular Java framework for creating web applications. Beside the backend code, JHipster also generates a frontend application in either Angular, React or Vue. Furthermore, the developer can choose to integrate many third-party services, such as Elasticsearch, caching and authentication with external providers like Okta or Auth0.

OutSystems

Outsystems is a low-code application development platform that allows organizations to create and deploy web and mobile applications quickly and efficiently. The platform provides a visual development environment where developers can create custom applications by dragging and dropping pre-built components and configuring them through a simple visual interface. OutSystems also provides a range of advanced features such as integration with external systems, real-time analytics and automated testing to ensure high-quality applications that meet the needs of users. The platform's low-code approach reduces the amount of hand-coding required, thereby enabling organizations to accelerate their application development processes and reduce their time-to-market [9].

Mendix

Finally, Mendix is a low-code development platform that enables businesses to create, deploy and manage custom web and mobile applications quickly and easily. The platform offers a visual development environment that allows developers to create applications using drag-and-drop components, reusable templates and a simple visual interface. Mendix also provides a range of advanced features such as integration with external systems, real-time collaboration and automated testing to ensure high-quality applications that meet the needs of users. The platform's low-code approach reduces the amount of manual coding required, thereby enabling organizations to accelerate their application development processes and reduce their time-to-market [10].

4.2 Evaluating Existing Platforms on Key Software Development Metrics

Code generation tools and low-code platforms have emerged as powerful resources for software development, streamlining the process with minimal coding effort. While low-code platforms offer an intuitive visual interface for building custom logic, code generation tools provide developers with a more flexible solution, enabling the construction of complex applications with enhanced control and adaptability.

When comparing software development tools, establishing a common ground for comparison is crucial. To achieve this, we have conducted a comprehensive evaluation of both low-code platforms and code generation tools based on the criteria outlined in section 3.1. Moving forward, we will refer to low-code platforms as LCDPs, representing Low-Code Development Platforms.

By constructing the same application using each platform, we have gained valuable insights into the strengths and weaknesses of each approach. Throughout the development process, we have placed particular emphasis on key software development metrics, including maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment. In this section, we will delve into each metric in detail and compare LCDPs with code generation tools such as JHipster.

This knowledge empowers us to comprehend the strengths and weaknesses inherent in each platform, enabling informed decisions to enhance our proposed solution. By leveraging this understanding, we can optimize our development process and deliver higher-quality software applications.

4.2.1 Evaluation of JHipster

JHipster stands out in terms of providing developers with full access to the codebase, enabling them to customize the entire codebase according to their specific requirements. This traditional approach to software development aligns with the key metrics of software engineering, including maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment.

However, a limitation of JHipster lies in its frontend capabilities, which are primarily geared towards administrative purposes rather than building customer-facing frontend applications for end users. Consequently, developers often need to develop an additional frontend application to cater to the needs of end users. This dual development approach impacts the time-to-market, as it requires building and maintaining two separate applications, resulting in increased development efforts.

Furthermore, while JHipster provides a generic REST API for backend interaction, this API may not be specifically tailored to the requirements of the frontend application. As a result, the frontend integration may not be optimal, potentially impacting the overall performance of the application. To enhance frontend integration, developers would need to customize the backend codebase to align with frontend needs and create a separate communication layer that ensures type safety and facilitates intuitive interaction with the backend. These additional steps can be time-consuming and increase the overall development effort.

Considering these factors, while JHipster offers significant advantages in terms of codebase accessibility and adherence to software engineering metrics, developers should be aware of the limitations regarding frontend capabilities and the need for additional frontend development efforts.

4.2.2 Evaluation of OutSystems

OutSystems is a low-code platform that enables developers to build web and mobile applications with minimal coding effort. The platform offers a visual interface for building custom logic, which allows developers to create complex applications with ease. As we developed the sample application using OutSystems, we focused on key metrics of software engineering, including maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment.

We observed that most of these metrics can be applied to the visual builder, as it abstracts the underlying codebase from the developer. However, we had concerns regarding the extendability, vendor lock-in and version control capabilities of the platform.

One limitation of OutSystems is its proprietary nature, which results in vendor lock-in. Developers are tied to the platform and face challenges if they want to migrate to another platform. This can be problematic if the platform becomes obsolete or if the developer desires greater flexibility.

In terms of version control, OutSystems provides a simplistic system where a new version is created upon publishing the application. However, advanced version control features like branching, merging or cherry-picking are not supported. This limitation can hinder collaboration and make it difficult to manage code changes effectively.

Furthermore, we found that the platform's extendability is limited. When developers encounter specific requirements that are not supported by the platform, they often need to modify the requirements or employ workarounds to achieve the desired functionality. This lack of flexibility can be a hindrance when building complex applications that require customized solutions.

In summary, while OutSystems simplifies application development through its low-code approach, considerations should be given to its limitations in terms of extendability, vendor lock-in and version control. Developers should carefully evaluate their project requirements and assess whether the platform aligns with their long-term goals and needs.

4.2.3 Evaluation of Mendix

After completing the application with OutSystems, we proceeded to develop the same application using Mendix, another low-code platform that offers developers the ability to create web and mobile applications with minimal coding effort. Similar to OutSystems, Mendix features a visual interface for building custom logic. During the development of the sample application using Mendix, we evaluated the platform based on key software engineering metrics, including maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment.

Our evaluation revealed similar limitations in terms of extendability, vendor lock-in and version control compared to OutSystems.

Like OutSystems, Mendix is a proprietary platform, which results in vendor lock-in. Developers are bound to the platform and may face challenges if they decide to migrate to another platform. This limitation becomes particularly problematic if the platform becomes obsolete or if developers seek greater flexibility.

Regarding version control, Mendix offers a basic system where a new version is created upon publishing the application. However, advanced version control features

such as branching, merging or cherry-picking are not supported. This limitation can impede effective collaboration and make managing code changes more challenging.

Additionally, we observed that the extendability of the Mendix platform is limited. When developers encounter specific requirements that are not supported by the platform's built-in features, they often need to modify the requirements or find workarounds to achieve the desired functionality. This lack of flexibility can be a constraint when developing complex applications that require tailored solutions.

In summary, while Mendix simplifies application development through its low-code approach, it is important to consider the limitations related to extendability, vendor lock-in and version control. Developers should assess these factors alongside their project requirements to determine whether Mendix aligns with their long-term goals and needs.

4.2.4 Conclusion

In conclusion, the evaluation of JHipster, OutSystems and Mendix revealed unique strengths and limitations associated with each platform. JHipster stood out with its focus on providing developers with full access to the codebase, allowing for customization and adherence to key software engineering metrics. This approach is particularly advantageous for complex projects that may require flexibility and the ability to handle unforeseen features and dependencies. However, JHipster faced limitations in its frontend capabilities, which required the development of additional frontend applications and increased development efforts and time-to-market.

OutSystems and Mendix demonstrated their low-code capabilities, simplifying the development process and offering visual interfaces for building custom logic. While these platforms provided convenience, concerns arose regarding extendability, vendor lock-in and version control limitations. The proprietary nature of both platforms restricted flexibility and posed challenges for migrating to other platforms. Additionally, advanced version control features were lacking, hindering effective collaboration and code management.

In summary, JHipster's emphasis on codebase accessibility and adherence to software engineering metrics make it a favorable choice for projects with complex requirements. OutSystems and Mendix offer low-code convenience but require careful consideration of limitations related to frontend capabilities, extendability, vendor lock-in and version control. Developers should thoroughly evaluate project requirements and long-term goals to select the platform that best aligns with their specific needs.

4.3 Limitations of Existing Platforms and the Need for a New Tool

In this chapter, we contrasted LCDPs and code generators in general on key metrics that are critical to software development projects. We came to the conclusion that code generator tools are more suitable for more complex software projects, as they provide more flexibility and customization capabilities. Therefore, we will shift our focus to code generator approaches, which also reflects the focus of our proposed tool.

In the field of code generators we analyzed mainly JHipster, as it is a proven and well-established tool that has a huge community of users. JHipster is a powerful tool that can be used to generate a complete application stack, including the frontend, backend and database. However, JHipster's main focus is on the backend side of the application, and it does not provide a lot of functionality for the frontend. The generated frontend mainly consists of administrative pages for managing the application's data, which is a good starting point for an administrative interface. However, it is not suitable for building a customer-facing frontend application that can be used by actual end users. While JHipster provides a lot of different technology options for different types of concerns, it also comes with a strong dependency on the Spring framework. Every dependency that is added to the project, increases the complexity of the project as well as the effort required to maintain it. Generating an application stack with JHipster can save initial setup efforts. On the other hand the amount of different technologies which can be selected during initialization can be overwhelming, typically a project evolves incrementally over time in terms of dependencies which contrasts to JHipsters philosophy to select all technologies involved at initialization time.

To address these limitations, we decided to build our own code generator tool that can be used to generate a complete application stack, similar to JHipster, but our solution includes components that helps the developer to build a more robust frontend that can be used to build a customer-facing application. Furthermore, we believe a close communication between the backend and frontend of the application, helps to improve the developer experience, as the developer has generally a simpler and more intuitive way to interact with the applications data.
Therefore, we decided to use a more traditional approach by leveraging the power of server side rendering, which allows us to build a frontend that is tightly coupled to the backend.

Beside that, our solution should be flexible and customizable to support a wide range of use cases and project requirements. Therefore, our solution embraces the key metrics that were presented in chapter 3.1, by providing a tool that is located in the field of code generators.

Chapter 5

Design and Implementation

In this chapter, we present a comprehensive overview of the design and implementation process of our proposed tool. This development was motivated by the identification of gaps and shortcomings within the existing landscape of tools, as discussed in chapter 4. We begin by outlining the objectives that guided the creation of this new tool, establishing a clear understanding of the problem space we aimed to address. Subsequently, we delve into the technical aspects, providing insights into the frameworks and technologies that were used to develop a new tool. Furthermore, we provide a detailed explanation of the architectural design, shedding light on the structural components and their interconnections, which play a critical role in the overall functionality of the tool.

5.1 Advantages of Code Generation Tools in Software Development

The previous chapter discussed the evaluation of existing code generation tools and low-code platforms. This section outlines the advantages of code generation tools, which can be used to justify the use of code generation tools in software development projects.

Low-code and no-code platforms are designed to simplify the development process for individuals without programming knowledge. However, software development is a complex process that requires not only coding expertise but also knowledge of infrastructure, security, performance, user experience, caching and many other factors. Non-programmers may find it challenging to create high-quality and secure software using these tools. Creating a robust and reliable software solution is not solely about writing good code, it requires a broader skill set.

The use of code generation tools in software development can be advantageous when used by skilled software engineers who can leverage the tool's capabilities to generate boilerplate code and improve the overall quality of the end product. This can result in reduced development costs and a faster development process. A skilled software engineer can use a high-quality code generation tool to generate code that contains best practices and a good starting point for the development of a new software product.

As we discussed in the previous chapter, code generation tools shine in many areas of software development, such as maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor lock-in and deployment. This is due to the fact that the software engineer has full control over the generated code and can customize it to meet the requirements of the project.

As experienced software engineers, we know that setting up a new software project can be a tedious and time-consuming process. The use of code generation tools can help to reduce the time and effort required to set up a new project. A code generation tool could generate a complete application stack, including the frontend, backend, database communication, authentication and many other features, while remaining the biggest advantage of code generation tools, which is the ability to customize the generated code to meet the requirements of the project. Moreover, code generation tools can reduce time-to-market by leveraging the generated code as a starting point for the development of a new software product.

In the previous chapter we outlined the features of JHipster, a well-known code generation tool in the Java ecosystem. JHipster can generate a complete application stack, including the frontend, backend, database communication, authentication and many other features. The backend side of the generated application is production ready for simple CRUD operations. However, the frontend side of the generated application is mainly an admin facing application, which is not suitable for client facing applications. Therefore, a developer still has to write a lot of code to create a frontend application. Overall, JHipster is a good code generation tool for a Full Stack Java developer, but the limitations of the frontend side of the generated application makes it less powerful for complete web application development.

These facts encourage us to develop a new code generation tool that shifts the focus not only to the backend side of the application but also to the frontend side. As we have a lot of experience in many different stacks, including Java, C#, PHP and TypeScript, we have a good understanding of the advantages and disadvantages of these stacks. In our experience, within the TypeScript ecosystem we could do more in less time. This is mainly due to the fact that TypeScript is a modern language with a great

type system that drastically reduces time to create or infer types. Another advantage of TypeScript, respectively JavaScript, is async / await, which makes asynchronous programming much easier. Moreover, the TypeScript ecosystem is growing rapidly and there are many high-quality libraries available. This brings us to the conclusion that the TypeScript ecosystem is a good choice for the development of a new code generation tool.

5.2 Design

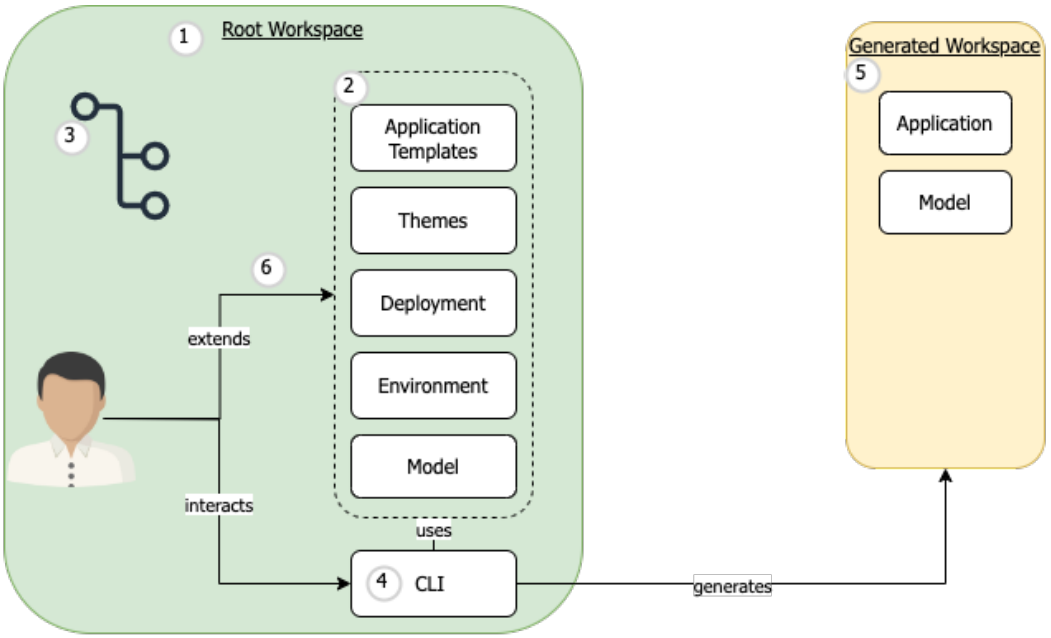
This section describes the high level design of the new tool. We begin by describing the objectives that guided the creation of this new tool, establishing the problem space we aimed to address.

5.2.1 Objectives

The primary objective of this thesis is to develop a tool that enables software engineers to generate a web application with a user interface from a data model. The tool should be able to generate a web application with a user interface that is maintainable, performant, scalable, version controlled, reusable, extendable, well documented and deployable. The tool itself should be extendable by the developer, so that the developer can add new features to the tool. The tool itself should be well documented, so that the developer can easily extend the tool.

5.2.2 Overview

The following figure 5.1 shows the high level design of the tool and the interaction with the developer.



1 **Root Workspace**: entry point for the developer, contains core components

2 **Core Components**: libraries providing reusable code in isolated modules, also used by the CLI 4

3 **Version Control**: manage and maintain changes

4 **CLI**: main interface for the developer to generate new workspaces. Uses the core components 2

5 **Generated Workspace**: generated by the CLI , contains libraries, environment, application

6 **Extendable Code Base**: all core components can be extended by the developer to add new features to the tool

Figure 5.1: Overview

5.3 Frameworks and libraries

The proposed implementation of the tool builds on the following frameworks and libraries. We briefly describe the frameworks and libraries outlined in the following subsections.

5.3.1 Remix

Remix is a full stack web framework that focuses on web standards and modern web application UX [11]. We have chosen Remix because of its intuitive approach to web development. Remix is a framework that is built on top of React and provides a lot of features out of the box. For example, it provides routing, scoped error handling and building forms. The advantage Remix has is that it allows the developer to write code that is easy to read and understand. This is achieved through the use of lifecycle methods and hooks which for example allow the developer to fetch data before the page is rendered.

5.3.2 React

React is a well known library for web application user interfaces [12]. It is used by many companies and is well documented. React allows us to write components that are reusable and easy to understand. React is also used by Remix and therefore we can use the same components in Remix as we would in React.

5.3.3 Prisma

Prisma is a database toolkit that provides type-safe database access and declarative data modeling [13]. It allows us to define the database schema in a declarative way via its Prisma DSL. The convenience having a library that provides type-safe database access and declarative data modeling is that it allows the developer to focus on the business logic. Prisma also provides the automatic generation of the model, the database client and schema migrations.

5.3.4 Tailwind CSS

Tailwind CSS is a utility-first CSS framework for rapidly building custom user interfaces [14]. It allows the developer to write CSS in a declarative way directly in the HTML. With its built in utility classes, it allows the developer to write CSS without the need to write custom CSS.

5.3.5 Storybook

Storybook is an open source tool for developing UI components in isolation for React [15]. It allows the developer to write components in isolation and document them. This is especially useful for a design system. Furthermore, it allows the developer to preview the components in different states without the requirement of deploying the application. This is especially useful for prototyping, testing and previewing the components.

5.3.6 TypeScript

TypeScript is a strongly typed superset of JavaScript that compiles to plain JavaScript [16]. It allows the developer to write understandable, maintainable and typesafe code. Typescript can catch errors at compile time and therefore reduces the number of bugs in the code. This is especially useful for a large codebase with many moving parts.

5.3.7 NX Workspace

NX is a build system with monorepo support and integrations for many technologies like Remix or Docker [17]. For example, it allows the developer to generate code, run tests and lint the code. Project dependencies are defined declarative and NX will only build the projects that are affected by the changes by using the concept of incremental build. This increases the speed of the development process and allows the developer to focus on the code. To visualize the dependencies between the projects, NX provides a dependency graph which can be generated. Nx allows us to separate applications and infrastructure into different projects. This allows us to reuse the infrastructure for different applications.

5.3.8 Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers [18]. It allows us to build the application and its dependencies into a single image which can be deployed to a server or to a local machine. This is especially useful for deployment because it allows the developer to run the application in the same environment as it will be deployed. This reduces the number of bugs that are caused by different environments. Being portable also allows the developer to run the application on different machines without the need to install the dependencies.

5.3.9 GitHub Workflows

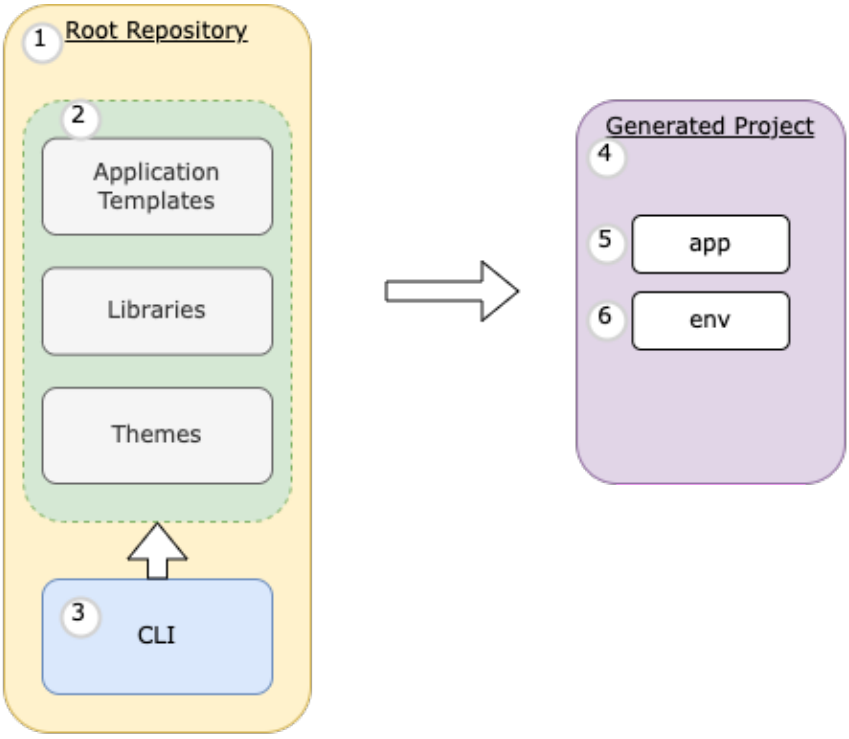
GitHub Workflows is a GitHub feature that allows developers to automate tasks based on events [19]. For example, it allows the developer to run tests on every push to the repository. This is especially useful for CI/CD because it allows the developer to automate the deployment process. GitHub Workflows is also used by NX to run tests and lint the code. By providing a CI/CD pipeline, GitHub Workflows allows the developer to focus on the code and not on the deployment process. Workflows are defined in a YAML file which makes them easy to read and understand.

5.4 Architecture and features

This section provides an overview of the architecture and features of our proposed tool named CodeFlow.

The web app generator is specifically designed to streamline the development process by rapidly generating web applications using modern web technologies. It is built upon a codebase (root workspace) that serves as the foundation for the generated project. The generator incorporates various essential tools and frameworks, including Remix, Prisma, TypeScript, Tailwind CSS, Storybook, NX workspace, Docker, GitHub workflows and a CLI tool. Additionally, the root workspace is fully functional, allowing developers to easily extend it by incorporating additional features. Changes made to the codebase can be viewed in real-time using the integrated development server, facilitating rapid codebase expansion. The availability of a fully functional root workspace significantly enhances the developer experience, fostering an improved and streamlined development process.

The following figure 5.2 shows the conceptual diagram of the application.



1. Root Project: Repository containing all components
2. Application Templates, Libraries & Themes: Extendable codebase used for generating projects
3. CLI: Tool to generate a new project using the codebase
4. Generated Project: The project generated by the CLI
5. app: The generated application
6. env: The environment defined for the generated application

Figure 5.2: Project Diagram

5.4.1 Libraries

The generated project includes a set of reusable libraries and pre-built applications. Each of these libraries is responsible for a specific task, such as managing the database schema, handling HTTP requests or generating forms. Extending these libraries allows developers to customize the generated project to suit their needs. The following figure 5.3 shows the separation of the provided libraries. Each library has its own purpose and encapsulates functionality.

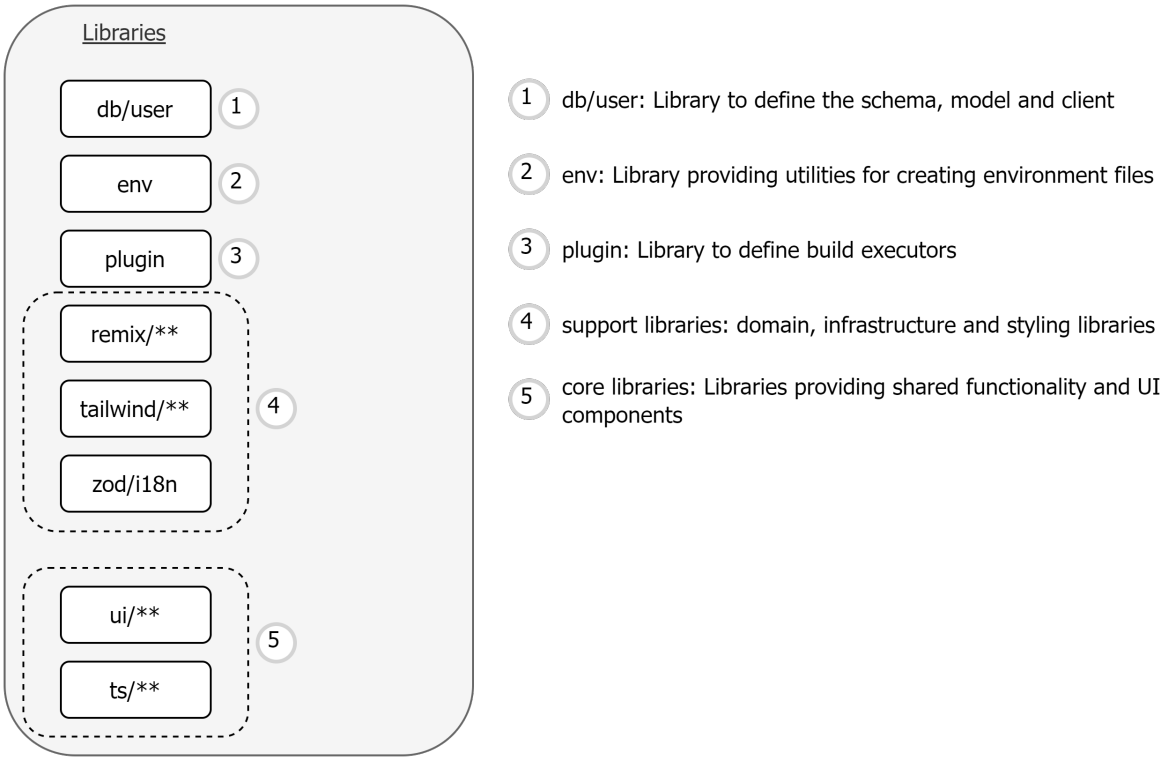


Figure 5.3: Project Diagram

The libraries include:

- **db/user:** This library contains the basic Prisma schema that can be extended using the Prisma DSL. It provides a foundation for defining database schemas and models, which can be used to build database-backed applications.
- **env:** The env library contains utilities for generating specific environment files used within the applications. It simplifies the process of managing environment variables by providing a set of tools for creating, updating and validating environment files.
- **plugin:** The plugin library contains custom NX executors that help in building applications. It provides a set of reusable tools that can be used to streamline the development process, reduce boilerplate code and enhance the developer experience.
- **remix/domain:** The remix/domain library contains utilities that help in separating business logic from the rest of the application. This library is designed for remix applications only and it provides a set of tools for managing business logic, such as managing state, handling events and performing API requests.
- **remix/forms:** The remix/forms library helps in automatically generating forms based on a Zod schema. Zod is a TypeScript-first schema declaration and validation library, which helps in defining data types and validating data. This library

simplifies the process of creating forms by automatically generating form components based on the given Zod schema.

- **remix/server:** The remix/server library contains common server-side logic that can be used in all remix applications. It provides a set of tools for handling HTTP requests, managing sessions and performing database queries, among other functionalities.
- **tailwind/animation:** The tailwind/animation library is a Tailwind plugin that adds additional utilities for animations to the Tailwind ecosystem. It provides a set of pre-built animations that can be easily integrated into web applications, simplifying the process of adding animations to UI components.
- **tailwind/pattern:** The tailwind/pattern library is a Tailwind plugin that adds additional utilities for background patterns to the Tailwind ecosystem. It provides a set of pre-built patterns that can be used to enhance the visual design of web applications, simplifying the process of creating visually appealing UI components.
- **ts/core:** The ts/core library contains general types as well as constants that are frequently used over the whole workspace. It provides a set of reusable tools for managing data types, constants and other general-purpose functionalities, reducing boilerplate code and enhancing the developer experience.
- **ts/utils:** The ts/utils library contains general utilities that are frequently used over the whole workspace. It provides a set of reusable tools for managing common functionalities such as string manipulation, array manipulation and other general-purpose functionalities, reducing boilerplate code and enhancing the developer experience.
- **ui/core:** The ui/core library contains the atoms of the UI library. Atoms are the smallest elements in a design system, such as buttons, badges and similar elements. It provides a set of reusable UI components that can be used to build custom UI components, reducing the amount of code needed to create custom UI elements.
- **ui/app:** The ui/app library contains UI components that are used in a typical admin application. These components are referred to as molecules or organisms, which are composed of atoms. It provides a set of reusable UI components that can be used to build custom admin applications, reducing the amount of code needed to create custom admin UI elements.
- **ui/common:** The ui/common library contains UI components that are used in many different kinds of applications, such as e-commerce, admin applications and so on. It provides a set of reusable UI components that can be used across different applications, reducing the amount of code needed to create custom UI elements.
- **ui/context:** The ui/context library contains common React context APIs such as LoadingContext or ThemeContext. These contexts are used to provide specific functionality to the entire application in a way that reduces the coupling of components by leveraging the power of hooks. For example, the LoadingContext can be used to display a loading indicator across the application, while the ThemeContext can be used to provide a consistent theme across all components. By using contexts, components can consume functionality without needing to know the implementation details, making the code more modular and easier to maintain.
- **ui/utils:** The ui/utils library contains commonly used UI utilities. It provides a set of reusable tools for managing common functionalities such as styling, layout and event handling. These utilities can be used to enhance the developer experience by reducing the amount of code needed to achieve common UI tasks.

- **zod/i18n:** The zod/i18n library adds internationalization support to the Zod library. It provides a set of tools for defining translations and applying them to Zod schemas, ensuring that all validation messages and error messages are correctly translated. This library simplifies the process of adding internationalization support to applications by providing a set of reusable tools that can be easily integrated into the Zod schema validation process. By using this library, developers can create applications that are accessible to a wider audience, enhancing the overall user experience.

5.4.2 Pre-built Applications

The generator also includes pre-built applications that can be customized based on specific project requirements. Each of this application contains two separate projects: one for the web application itself and another for setting up the environment variables required by the application. The pre-built applications are designed to be used as starting points for developers who want to build web applications quickly and efficiently. Furthermore, they can be extended to add new features or customize existing ones. Finally, these applications can then be used as templates for creating new applications. The following figure 5.4 shows the pre-built applications included in the generator. It shows that each application contains two separate projects: one for the web application itself and another for setting up the environment variables required by the application.

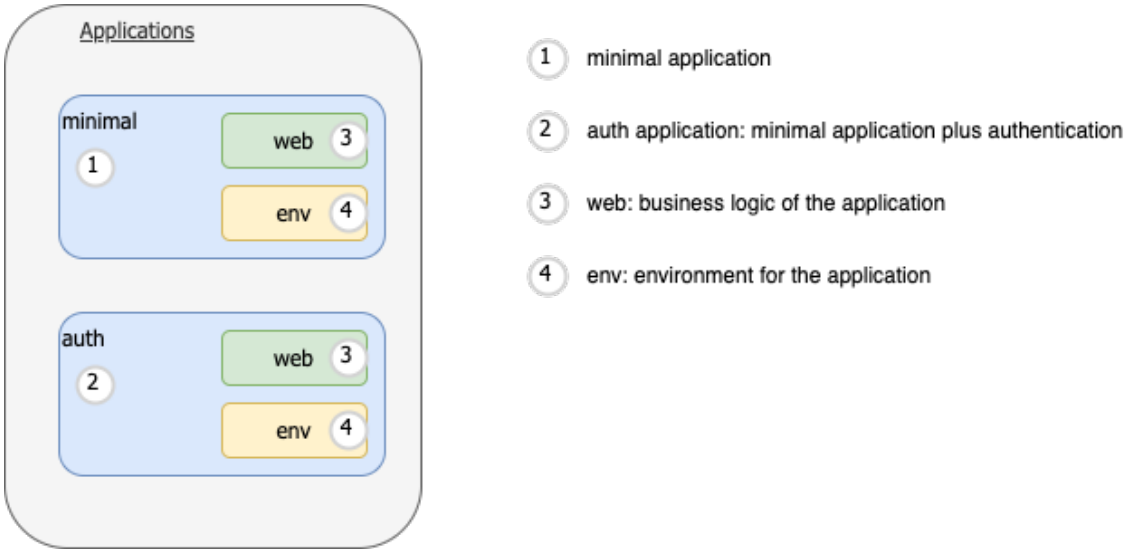


Figure 5.4: Applications

The following applications are included in the generator:

- **apps/minimal/web:** The apps/minimal/web project is a minimal Remix application that provides all the fundamental components needed to create a web application. It serves as a great starting point for developers who want to create web applications quickly and efficiently. By providing a basic structure and set of UI components, developers can focus on building their application's unique features without spending time on boilerplate code.
- **apps/minimal/env:** The apps/minimal/env project is responsible for setting up the environment variables for the minimal application. It provides a starting point for developers to configure the environment variables required by their applications. The developer can extend the environment file whenever needed, making it easy to add new environment variables as the application grows.
- **apps/auth/web:** The apps/auth/web project is a Remix application that comes with built-in authentication functionality. It provides a great starting point for

developers who want to build secure web applications that require user authentication. By providing authentication functionality out-of-the-box, developers can focus on building their application's unique features, while ensuring that the application is secure and user-friendly.

- **apps/auth/env:** The apps/auth/env project is responsible for setting up the environment variables for the auth application. It provides a starting point for developers to configure the environment variables required by their authentication application. The developer can extend the environment file whenever needed, making it easy to add new environment variables as the authentication application grows.

5.4.3 Deployment Application

Besides that, the generator contains an additional application that allows to easily integrate docker into the development process. This application is located in the apps/docker directory and contains a docker-compose file as well as all the required environment files that are generated through the specific environment projects. The existing docker-compose file is meant to be used for development purposes only, but it can be easily extended to support production deployments as well.

5.4.4 CLI Application

The CLI project located in apps/cli is responsible for providing a command-line interface that allows to easily generate a new workspace or update an existing one with new features.
The CLI is the main interface for the code generation tool.

It allows the developer to create a new NX workspace including many libraries and applications as well as a CI/CD pipeline integrated with GitHub. After the initial setup of the NX workspace, the developer can add additional applications with the CLI.

The proof of concept of the code generation tool includes basic applications and many useful libraries that can be used to further customize the generated applications. Besides that, the developer can add additional libraries and applications to the NX workspace as full access to the workspace is provided.

This architecture allows to easily extend the generator with new features, such as additional libraries or applications. Additionally, the developer can easily customize the generated workspace by adding new libraries or applications to the workspace. As the developer has full access and control over the entire workspace, it is possible to customize every aspect of the entire project, which reflects the philosophy of traditional application development. Earlier, we discussed the key metrics of software development, including maintainability, performance, scalability, version control, reusability, extendability, documentation, vendor-lock-in and deployment. Our generator addresses all of these metrics by leveraging the power of traditional application development and the developer experience of generators, which mainly affects the performance in terms of development speed and the reusability of the generated code.

The following figure 5.5 shows the module dependencies:

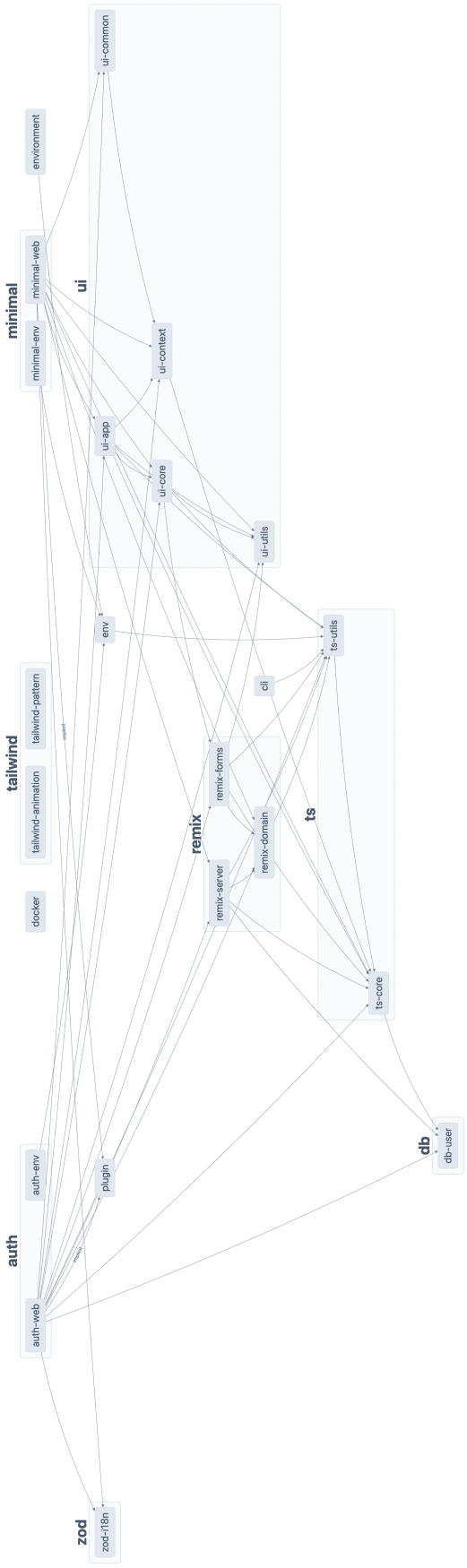


Figure 5.5: Dependency Graph

5.5 Usage of CodeFlow

In this section, we explore the practical usage of CodeFlow and delve into its various features and functionalities. Whether you are a seasoned developer or new to the world of software development, this section will provide valuable insights on how to effectively leverage CodeFlow for rapid application development.

To enhance your understanding and facilitate the learning process, we have created a comprehensive video series that walks you through the usage of CodeFlow step by step. Each video focuses on a specific aspect of CodeFlow, ranging from the setup and configuration of the development environment to the generation of high-quality code and the integration of custom code. We encourage you to follow along with the video series, which can be accessed at `https://drive.google.com/drive/folders/1dVv8U_OnwGzdbIJXRLTwEJvTJ5avYXLR?usp=sharing`.

In addition to the video series, this section provides a high level overview of the individual parts that are presented in a practical fashion throughout the video course. We will demonstrate how CodeFlow enables efficient prototyping, code generation and the development of scalable and maintainable applications. Furthermore, we will highlight the benefits and considerations associated with using CodeFlow, allowing you to make informed decisions and optimize your development workflow.

The complete source code of the resulting application from the video series can be found at `https://github.zhaw.ch/eglipat3/todo-app`. Note, this repository is only accessible to limited number of users. If you are interested in gaining access to the repository, please reach out to us at `mailto:eglipat3@students.zhaw.ch`. Whether you are looking to streamline your development process, increase productivity or build high-quality applications in a shorter time frame, CodeFlow offers a powerful solution that combines the benefits of low-code development with the flexibility and extensibility of traditional coding practices.

5.5.1 Cloning the CodeFlow Repository

To get started with CodeFlow, the initial step is to clone the repository. It's important to note that CodeFlow is currently a closed-source project, available to a limited number of users. If you are interested in gaining access to the repository, please reach out to us at `mailto:eglipat3@students.zhaw.ch` and we will gladly provide you with the necessary access.

Once you have been granted access to the repository, you can proceed with cloning it using the following command:

```
git clone <repository-url>
```

Replace <**repository-url**> with the actual URL of the CodeFlow repository. By executing this command, you will download the entire repository to your local machine, enabling you to explore the CodeFlow tool and its associated resources.

With the repository successfully cloned, you are now ready to embark on your journey with CodeFlow and discover its capabilities for rapid application development.

5.5.2 Setting up the Development Environment

To begin using CodeFlow, we first need to set up the development environment. Emphasizing a seamless developer experience, the setup process is straightforward and requires executing a single command.

To install the necessary dependencies and prepare the development environment, run the following command:

```
npm install
```

This command will handle the installation of all required dependencies, ensuring that your environment is properly configured. Once the installation process is complete, we can proceed with creating the target workspace. But before we do that, let's take a moment to delve into the concept of the root workspace and the target workspace.

5.5.3 Root Workspace and Target Workspace

The root workspace serves as the foundation of CodeFlow and encompasses the CLI application responsible for generating the target workspace. It also houses the libraries and application templates necessary for generating the target workspace.

The CLI application plays a crucial role in this process by prompting the user for input and using that information to generate the target workspace. It adjusts the provided source code from the libraries and application templates to align with the user's specific requirements.

The target workspace, generated by the CLI application, serves as the working environment for developers. If the developer chooses to generate an application within the target workspace, it will contain a fully functional application. Developers have complete access to the codebase and can freely modify it to meet their needs.

Furthermore, the CLI application supports extending previously generated target workspaces. Developers can generate additional projects within the target workspace, which seamlessly integrate with the existing codebase. Additionally, developers have the flexibility to extend the codebase of the CodeFlow repository itself, enhancing the capabilities of the root workspace. These modifications are applied to the target workspace upon generation.

Now that we have gained a better understanding of the root workspace and the target workspace, we are ready to proceed with creating the target workspace.

5.5.4 Generating the Target Workspace

To generate the target workspace, we need to execute the following command:

```
npm run cli:generate:project
```

This command will prompt the user for input and use that information to generate the target workspace. The following questions will be asked and we provide the answers that we used for the video series:

- **What is the name of the project?**
  todo-app
- **Please specify the parent directory from the new parent**
  <path-to-your-target-workspace-directory>
- **Please select the type of GitHub workflow you want to use**
  ci-only
- **Please select the theme**
  sky

- **Do you want to create a new app?**
  Yes
- **Which template do you want to use for the new app?**
  Application with Authentication included
- **What is the directory of the app? Please make sure that the directory is unique and does not exist yet.**
  main
- **What is the display name of the app? This name will be used in the PWA manifest and in some other places.**
  Todo Application
- **What is the description of the app? This name will be used in the PWA manifest and in some other places.**
  This is an amazing todo application
- **What is the theme color of the app? It is recommended to use the primary color of your app. By default we use the primary-[500] color of your theme.**
  #0ea5e9
- **What is the background color of the app? By default we use the gray-[900] color of your theme.**
  #0f172a
- **What is the name of the database client library? Most probably you will extend this library with your custom schema, so it is recommended to use a name that fits your DB schema purpose. Make sure that no other directory with the same name exists in libs/db.**
  user
- **Do you want to create a new app?**
  No

Once the CLI application has finished generating the target workspace, we can open the target workspace in our IDE of choice. All the required dependencies are already installed and the target workspace is ready for development.

The following figure 5.6 shows the CLI application in action.

```
> code-flow@0.0.0 cli:generate:project
> npx nx generate:project cli



> nx run cli:"generate:project"

Debugger listening on ws://localhost:9229/1497f873-003d-4cda-8c0d-068e88ce4a16
Debugger listening on ws://localhost:9229/1497f873-003d-4cda-8c0d-068e88ce4a16
For help, see: https://nodejs.org/en/docs/inspector
? What is the name of the project? todo-app
? Please specify the parent directory from the new project:
```

Figure 5.6: CodeFlow CLI

To start the development server of the main application, we need to execute the following command:

```
npm run main:dev
```

This command will start the development server of the main application, which we can access by navigating to `http://localhost:3000` in our browser.

The generated application is fully functional and provides authentication functionality out of the box. Besides the authentication, the application also provides internationalization support, PWA integration, error handling, full responsiveness ensuring the application works on mobile devices as well as on desktop devices and a dashboard with a navigation that can be easily extended by developers.

The following figure 5.7 shows the dashboard of the generated application.

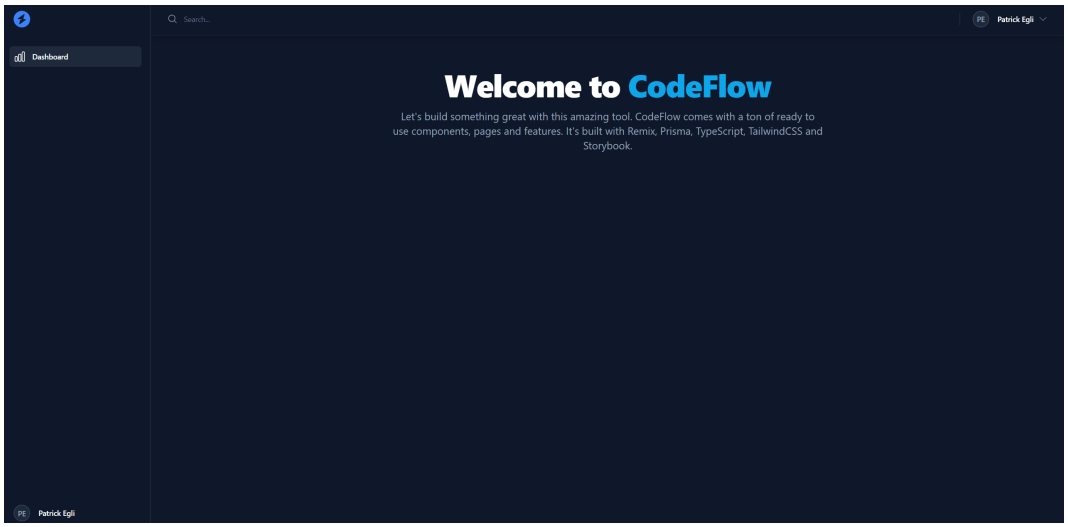


Figure 5.7: CodeFlow Dashboard

Now that we have successfully generated the target workspace, we could start developing our application. But before we do that, let's take a moment to explore the UI component libraries that CodeFlow provides.

5.5.5 UI Component Libraries

CodeFlow provides a set of UI component libraries that developers can use to build their applications. These libraries are located in the **libs/ui** directory of the workspace and are structured as follows:

- **ui/core:** The ui/core library contains the atoms of the UI library. Atoms are the smallest elements in a design system, such as buttons, badges and similar elements. It provides a set of reusable UI components that can be used to build custom UI components, reducing the amount of code needed to create custom UI elements.
- **ui/app:** The ui/app library contains UI components that are used in a typical admin application. These components are referred to as molecules or organisms, which are composed of atoms. It provides a set of reusable UI components that can be used to build custom admin applications, reducing the amount of code needed to create custom admin UI elements.
- **ui/common:** The ui/common library contains UI components that are used in many different kinds of applications, such as e-commerce, admin applications

and so on. It provides a set of reusable UI components that can be used across different applications, reducing the amount of code needed to create custom UI elements.

- **ui/context:** The ui/context library contains common React context APIs such as LoadingContext or ThemeContext. These contexts are used to provide specific functionality to the entire application in a way that reduces the coupling of components by leveraging the power of hooks. For example, the LoadingContext can be used to display a loading indicator across the application, while the ThemeContext can be used to provide a consistent theme across all components. By using contexts, components can consume functionality without needing to know the implementation details, making the code more modular and easier to maintain.
- **ui/utils:** The ui/utils library contains commonly used UI utilities. It provides a set of reusable tools for managing common functionalities such as styling, layout and event handling. These utilities can be used to enhance the developer experience by reducing the amount of code needed to achieve common UI tasks. For example, the library can include utilities for handling responsive design, creating animations and handling input validation.

To provide a great developer experience, CodeFlow integrates Storybook. Storybook is a tool for developing UI components in isolation, which allows developers to build UI components without needing to worry about application-specific dependencies. Besides that, Storybook is suitable for exploring the UI component libraries that CodeFlow provides. To start Storybook for the ui/core library, we need to execute the following command:

```
npm run ui:core:storybook
```

This command will start Storybook for the ui/core library, which we can access by navigating to `http://localhost:4400` in our browser.

The following figure 5.8 shows a screenshot of Storybook in action. This is the Storybook instance of the ui/core library, which contains the atoms of the UI library.

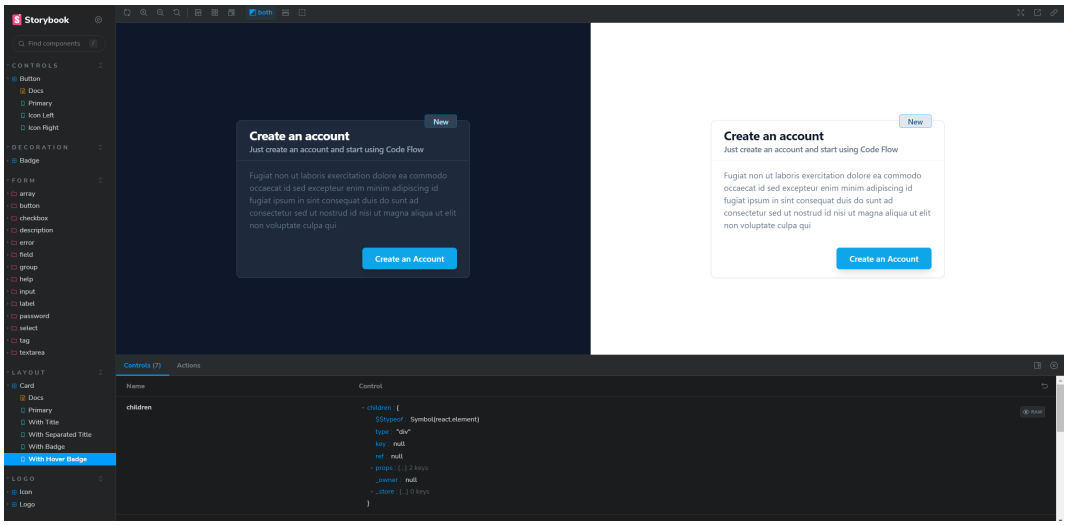


Figure 5.8: CodeFlow Storybook instance of the ui/core library

The same can be done for the ui/app component library by executing the following command:

```
npm run ui:app:storybook
```

This command will start Storybook for the ui/app library, which we can access by navigating to `http://localhost:4401` in our browser.

The following figure 5.9 shows a screenshot of Storybook in action. This is the Storybook instance of the ui/app library, which contains the atoms of the UI library.

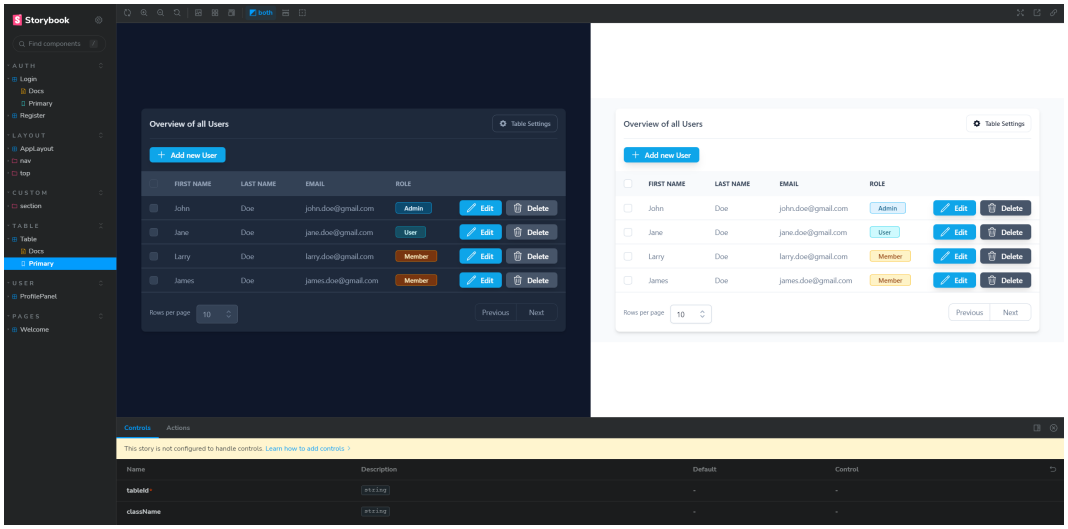


Figure 5.9: CodeFlow Storybook instance of the ui/app library

Feel free to explore the UI component libraries that CodeFlow provides. Once you are done exploring, we can move on to the next step, spinning up the database.

5.5.6 Spinning up the Database

CodeFlow provides a PostgreSQL database container that we can use for development purposes as well as for production deployments if needed. To spin up the PostgreSQL database container, we execute the following command:

```
docker-compose -f apps/docker/docker-compose.yaml up -d
```

This command spins up the PostgreSQL database container in the background. To stop the PostgreSQL database container, we execute the following command:

```
docker-compose -f apps/docker/docker-compose.yaml down -v
```

This command stops the PostgreSQL database container and removes the data volume. If we do not want to remove the data volume, we can omit the **-v** flag.

The PostgreSQL database container reads by default the environment variables from the **apps/docker/env/dev/db.env** file. To change the values of these environment variables, we can adjust the **.env.local** file in the root directory of the workspace. The following **.env.local** file is an example of how we can adjust the environment variables of the PostgreSQL database container:

```
POSTGRES_HOST=db
POSTGRES_DB="todo-application"
POSTGRES_USER="todo-user"
POSTGRES_PASSWORD="top-secure"
```

These environment variables are picked up by the **build:env** target. The generated workspace contains many applications that use the **build:env** target to generate en-

vironment variables for the applications. To run the build:env target for all applications that use it, we execute the following command:

```
npm run build:env
```

After running the **build:env** target, we should see the configured environment variables in the **apps/docker/env/dev/db.env** file. If we want to apply the environment variables to the PostgreSQL database container, we need to restart the container by executing the following command:

```
docker-compose -f apps/docker/docker-compose.yaml restart db
```

Next, we want to extend our model by adding a new entity to our application.

5.5.7 Extending the Model

CodeFlow incorporates Prisma [13], a cutting-edge ORM for Node.js and TypeScript, which offers a range of tools for data modeling in applications. To extend the model of our application, we navigate to the **libs/db/user/prisma** directory within the workspace. Within this directory, we locate the **schema.prisma** file, which defines the model of our application.

The default model of our application is represented as follows:

```
generator client {
  provider = "prisma-client-js"
  output   = "./generated/prisma"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String    @id @default(uuid())
  email     String    @unique
  password  String
  firstName String?
  lastName  String?
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
}
```

The provided model showcases the default structure for a User entity in our application. It includes attributes such as id, email, password, firstName, lastName, createdAt and updatedAt, each with specific data types and optional modifiers. Prisma's generator configuration specifies the output directory for the Prisma client, while the datasource configuration establishes the connection to a PostgreSQL database using the provided URL.

By modifying the schema.prisma file, developers can tailor the data model to suit their application's specific requirements. This flexibility enables the seamless integration of custom entities, relationships and additional attributes into the application.

In our case, we want to extend the model of our application to include a new entity called **Todo**. Therefore, we add the following code to the schema.prisma file:

```
model Todo {
  id            String    @id @default(uuid())
  title         String
  description   String
  completed     Boolean   @default(false)
  createdAt     DateTime  @default(now())
  updatedAt     DateTime  @updatedAt
  user          User      @relation(fields: [userId], references:
     [id])
  userId        String
  }
```

Additionally, we need to add the following code to the User entity:

```
todos     Todo[]
```

This code establishes a one-to-many relationship between the User and Todo entities. The User entity can have many Todo entities, while the Todo entity can only have one User entity. This relationship is represented by the todos attribute in the User entity and the user attribute in the Todo entity. The todos attribute in the User entity is an array of Todo entities, while the user attribute in the Todo entity is a single User entity.

The final **schema.prisma** file should look as follows:

```
generator client {
  provider = "prisma-client-js"
  output   = "./generated/prisma"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String    @id @default(uuid())
  email     String    @unique
  password  String
  firstName String?
  lastName  String?
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
  todos     Todo[]
}

model Todo {
  id            String    @id @default(uuid())
  title         String
  description   String
  completed     Boolean   @default(false)
  createdAt     DateTime  @default(now())
  updatedAt     DateTime  @updatedAt
  user          User      @relation(fields: [userId], references:
     [id])
  userId        String
}
```

Once we are done modifying the schema.prisma file, we can move on to the next step, which is to generate the Prisma client. So far, we only modified the schema.prisma file, which is a declarative file that defines the model of our application. To generate the actual Prisma client, we need to execute the following command:

```
npm run prisma:generate
```

This command will generate the Prisma client, which we can use to interact with the database in a type-safe manner. The generated Prisma client is located in the **libs/db/user/generated/prisma** directory within the workspace. The generated Prisma client is exported from the **libs/db/user** library and can be imported into other libraries and applications through the **@todo-app/db/user** package. In case of you decided to use a different name for your workspace, the package name will be different, e.g., **@my-workspace/db/user**.

Now that we have generated the Prisma client, we can move on to the next step, which is to create a migration. Add the following script to the **package.json** file in the root of the workspace:

```
"main:prisma:migrate": "cd libs/db/user && prisma migrate dev --
   schema ./prisma/schema.prisma",
```

This command will create a migration based on the changes we made to the schema.prisma file. The resulting **package.json** file should look as follows:

```
{
  "name": "todo-app",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "build:affected": "nx affected:build",
    "build:all": "nx run-many --all --target=build",
    "build:all:no:cache": "nx run-many --all --target=build --
       skip-nx-cache",
    "lint": "nx run-many --all --target=lint",
    "lint:fix": "nx affected:lint --fix",
    "prettier": "nx format:write",
    "build:env": "npx nx run-many --all --target=build:env",
    "ui:app:storybook": "cross-env NODE_OPTIONS=--
       openssl-legacy-provider npx nx storybook ui-app",
    "ui:core:storybook": "cross-env NODE_OPTIONS=--
       openssl-legacy-provider npx nx storybook ui-core",
    "prisma:generate": "npx nx run-many --all --target=prisma:
       generate",
    "main:prisma:migrate": "cd libs/db/user && prisma migrate dev
       --schema ./prisma/schema.prisma",
    "test": "npx nx run-many --all --target=test --skip-nx-cache",
    "postinstall": "remix setup node && npm run prisma:generate &&
        npm run build:env",
    "prepare": "husky install",
    "main:dev": "npx nx dev main-web",
    "main:build": "npx nx build main-web"
  },
  "private": true,
  "dependencies": {
    ...
  },
  "devDependencies": {
```

```
    ...
  },
  "workspaces": [
    "libs/*"
  ],
  "engines": {
    "node": ">=18.13.0",
    "npm": ">=8.19.3"
  }
}
```

To create the migration, we need to execute the following command:

```
npm run main:prisma:migrate
```

This command will create a new migration in the **libs/db/user/prisma/migrations** directory within the workspace. The migration is a declarative file that defines the changes that need to be applied to the database. These migrations are automatically applied to the running database instance during development. However, in production, migrations need to be applied with the following command:

```
prisma migrate deploy
```

This command will apply all pending migrations to the database instance. After the migration has been applied, we can move on to the next step, which is to create schemas that will perform input validation.

5.5.8 Creating Schemas for Input Validation

In this section, we will create schemas that will perform input validation. Besides input validation, these schemas can also be used to generate forms automatically as CodeFlow has built-in support for generating forms from zod-schemas.

To perform the actual input validation, CodeFlow uses the **zod** library [20], which is a TypeScript-first schema declaration and validation library. We recommend to add all the schema files to the **libs/ts/core/src/lib/schema** directory and export them from index.ts file, located in the same directory. Let's create a schema file called **todo.create.schema.ts** in the **libs/ts/core/src/lib/schema** directory and add the following code to it:

```
import { z } from 'zod';
import { SESSION_PROPERTIES } from '../constants';

export const TODO_CREATE_SCHEMA = z.object({
  title: z.string().min(1),
  description: z.string().min(1),
  action: z.enum(['create']),
  [SESSION_PROPERTIES.csrf]: z.string().min(1),
});
```

This schema defines the following properties:

- **title:** The title of the todo. This property is required and must be a string with a minimum length of 1.
- **description:** The description of the todo. This property is required and must be a string with a minimum length of 1.

- **action:** The action that needs to be performed. This property is required and must be equal to the string **create**.
- **csrf:** The CSRF token. This property is required and must be a string with a minimum length of 1.

Let's create another schema file that is responsible for updating the state of a todo. We call this schema file **todo.complete.schema.ts** and add the following code to it:

```
import { z } from 'zod';
import { SESSION_PROPERTIES } from '../constants';

export const TODO_COMPLETE_SCHEMA = z.object({
  id: z.string().min(1),
  completed: z.boolean(),
  action: z.enum(['complete']),
  [SESSION_PROPERTIES.csrf]: z.string().min(1),
});
```

Then we simply export the schema files from the **libs/ts/core/src/lib/schema/index.ts** file:

```
export * from './login.schema';
export * from './register.schema';
export * from './todo.complete.schema';
export * from './todo.create.schema';
```

That's it! By creating these schemas, we have already defined the input validation rules for our application. Feel free to add more schemas to the **libs/ts/core/src/lib/schema** directory if you need to perform input validation for other use cases.

In the next section, we will create the actual business logic that we will use in the data loaders and action handlers of Remix.

5.5.9 Creating Business Logic

In this section, we will create the business logic. The input coming from the data loaders and action handlers is validated against the schemas that we created in the previous section. We recommend to add all the business logic files to the **apps/main/web/app/domain** directory assuming that we are working within the **main** application.

Let's create a new directory called **todo** in the **apps/main/web/app/domain** directory. This directory houses all the business logic related to todos. In this directory, we create three files:

- **todo.create.server.ts:** This file contains the business logic for creating a todo.
- **todo.complete.server.ts:** This file contains the business logic for changing the state of a todo.
- **todo.list.server.ts:** This file contains the business logic for listing todos that belong to the authenticated user.

Let's start with the **todo.create.server.ts** file.

```
import { prisma } from '@todo-app/db/user';
import { makeDomainFunction } from '@todo-app/remix/domain';
import { handleMutationError, requireUser } from '@todo-app/remix/
    server';
```

```
import { TODO_CREATE_SCHEMA } from '@todo-app/ts/core';
import { z } from 'zod';
import i18nextServer from '../../i18n/i18next.server';

const environmentSchema = z.object({
  request: z.any(),
});

export const todoCreateMutation = makeDomainFunction(
  TODO_CREATE_SCHEMA,
  environmentSchema
)(async (values, { request }) => {
  const { title, description } = values;
  const user = await requireUser(request);

  try {
    return await prisma.todo.create({
      data: {
        title: title as string,
        description: description as string,
        user: {
          connect: {
            id: user.id,
          },
        },
      },
    });
  } catch (e) {
  await handleMutationError(e, request, i18nextServer);
  }
});
```

In only 34 lines of code, we have created a function that is responsible for creating a todo, including the input validation, authentication checks as well as error handling.

- **Line 1 - 6:** Importing the required dependencies.
- **Line 8 - 10:** Defining the environment schema. This schema is used to validate the environment that is passed to the function. In this case, we only need the request object from the environment.
- **Line 15 - 34:** Defining the actual business logic. This code is only executed in case of the input validation has passed.
- **Line 17:** Authentication check. If the user is not authenticated, an automatic redirect to the login page is performed. This ensures that only authenticated users can create todos.

Let's create the **todo.complete.server.ts** file with the following code:

```
import { prisma } from '@todo-app/db/user';
import { makeDomainFunction } from '@todo-app/remix/domain';
import { handleMutationError, requireUser } from '@todo-app/remix/
   server';
import { TODO_COMPLETE_SCHEMA } from '@todo-app/ts/core';
import { z } from 'zod';
import i18nextServer from '../../i18n/i18next.server';

const environmentSchema = z.object({
  request: z.any(),
```

```
});

export const todoCompleteMutation = makeDomainFunction(
  TODO_COMPLETE_SCHEMA,
  environmentSchema
)(async (values, { request }) => {
  const { id, completed } = values;
  const user = await requireUser(request);

  try {
    const todo = await prisma.todo.findFirst({
      where: {
        id: id as string,
        userId: user.id,
      },
    });

    if (!todo) {
      throw new Error('Todo not found');
    }

    return await prisma.todo.update({
      data: {
        completed: completed as boolean,
      },
      where: {
        id: todo.id,
      },
    });
  } catch (e) {
  await handleMutationError(e, request, i18nextServer);
  }
});
```

This file is very similar to the **todo.create.server.ts** file. The only difference is that we are updating an existing todo instead of creating a new one. Therefore, we need to perform an additional database query to check if the todo exists and belongs to the authenticated user.

Finally, let's create the **todo.list.server.ts** file with the following code:

```
import { json } from '@remix-run/node';
import { Prisma, prisma } from '@todo-app/db/user';
import { requireUser } from '@todo-app/remix/server';

export type TodoListLoaderData = {
  todos: TodoListType;
};

export const todoListLoader = async (request: Request) => {
  const user = await requireUser(request);
  const todos = await prisma.todo.findMany({
    where: {
      userId: user.id,
    },
  });

  return json<TodoListLoaderData>({
```

```
    todos,
  });
};

const todoListInternal = async (userId: string) => {
  return prisma.todo.findMany({
  where: {
  userId,
  },
  });
};

export type TodoListType = Prisma.PromiseReturnType<typeof
    todoListInternal>;
```

This file is a bit different from the previous two files, as the function is used as a data loader. The data loader is responsible for loading the data that is required for the page. In this case, we are loading all todos that belong to the authenticated user. To ensure type safety, we are using the **Prisma.PromiseReturnType** type to infer the return type of the **todoListInternal** function. This allows us to use the same type for the frontend and backend code and helps to avoid type errors. This is especially useful when the database schema changes, as the type is automatically updated or when we query different fields from the database in a later stage of the project.

Most likely, developers will extend the default UI components with additional ones. The following section describes how to create a new UI component that we will use in the final section of this chapter.

5.5.10 Creating a new UI component

In this section we create a simple UI component that is responsible for displaying a title and a subtitle. This is a highly reusable component that can be used in many different places.

Let's start by creating a new file called **SectionTitle.tsx** in the **libs/ui/app/src/lib/-section** directory with the following code:

```
import { cx } from '@todo-app/ui/utils';

type SectionTitleProps = {
  title: string;
  subTitle: string;
  className?: string;
};

export function SectionTitle({ title, subTitle, className }:
    SectionTitleProps) {
  return (
    <div className={cx('flex flex-col border-b border-gray-200
       pb-2 dark:border-gray-700', className)}>
      <h2 className="text-lg font-bold text-gray-900 dark:
         text-white">{title}</h2>
      <span className="text-sm font-normal text-gray-600 dark:
         text-gray-500">{subTitle}</span>
    </div>
  );
}
```

This component has a wrapper div with a border at the bottom and within this wrapper, we have a title and a subtitle. Previously, we mentioned the Storybook integration that allows us to develop UI components in isolation. Now, we want to leverage the power of Storybook to develop this component in isolation. Let's create a new file called **SectionTitle.stories.tsx** in the **libs/ui/app/src/lib/section** directory with the following code:

```
import { SectionTitle } from './SectionTitle';

export default {
  title: 'custom/section/SectionTitle',
  component: SectionTitle,
  argTypes: {
    title: {
      control: 'text',
    },
    subTitle: {
      control: 'text',
    },
  },
};

export const Primary = {
  args: {
    title: 'Create a new project',
    subTitle: 'Create a new project and invite your team members',
  },
  parameters: {},
};
```

Luckily, Storybook provides an intuitive API that allows us to create stories for our components in a very short time. In this case, we are creating a story for the **SectionTitle** component. The **title** property is used to define the title of the story. Additionally, the **component** property is used to define the component that we want to use for the story and the **argTypes** property is used to define the arguments of the component. In this case, we have two arguments, the **title** and the **subTitle**. Both arguments are of type **text**.
Furthermore, we can create multiple stories for the same component. In this case, we are only creating one story called **Primary** and we provide the default values for the arguments **title** and **subTitle**.

Now, we need to make sure, that the **SectionTitle** component is exported from the **libs/ui/app/src/lib/section/index.ts** file. So, let's create the **index.ts** file and add the following code:

```
export * from './SectionTitle';
```

Furthermore, we have to make sure that all the components within the **libs/ui/app/src/lib/section** directory are exported from the **libs/ui/app** library. Therefore, we have to export the **section** directory from the **libs/ui/app/src/lib/index.ts** file. The resulting **index.ts** file should look like this:

```
export * from './auth';
export * from './layout';
export * from './section'; // Add this line to the file
export * from './table';
export * from './user';
export * from './welcome';
```

Let's spin up Storybook to see the custom UI component in action. To do so, we have to run the following command:

```
npm run ui:app:storybook
```

The following figure 5.10 shows a screenshot of the custom UI component within the Storybook instance of the **ui/app** library.

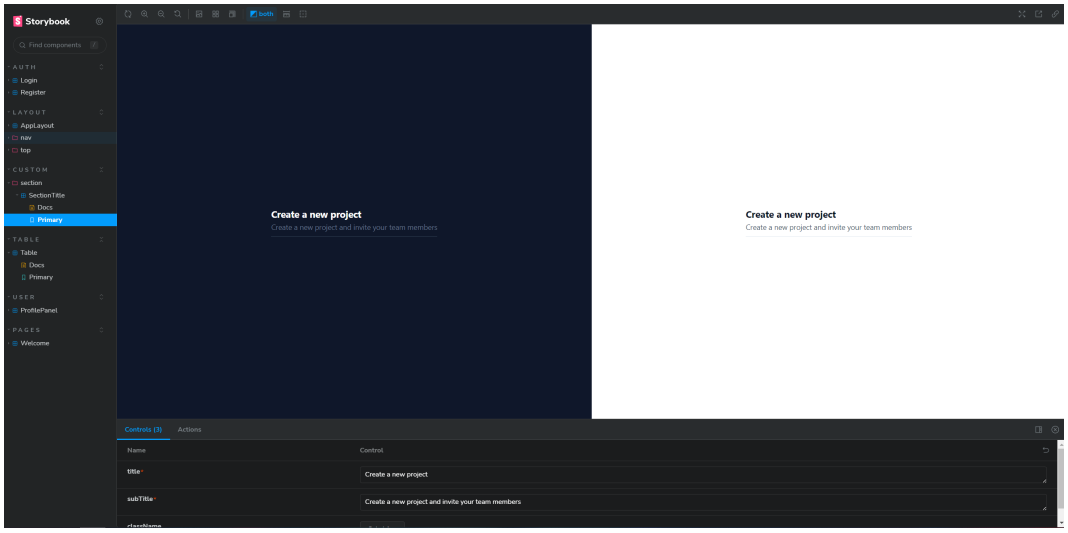


Figure 5.10: Storybook custom UI component

As the last step, we need to combine the business logic with the frontend code.

5.5.11 Combining the business logic with the frontend code

In this section, we are going to combine the business logic with the frontend code. Within Remix, all routes are defined in the **app/routes** directory. In our case, in the **apps/main/web/app/routes** directory.

Let's modify **apps/main/web/app/routes/__protected/__layout/index.tsx** with the following code:

```
import { CheckIcon, XMarkIcon } from '@heroicons/react/24/outline
    ';
import { ActionFunction, LoaderFunction } from '@remix-run/node';
import { useLoaderData } from '@remix-run/react';
import { formAction } from '@todo-app/remix/forms';
import { SESSION_PROPERTIES, TODO_COMPLETE_SCHEMA,
    TODO_CREATE_SCHEMA } from '@todo-app/ts/core';
import { SectionTitle } from '@todo-app/ui/app';
import { BaseButton, Card, Form } from '@todo-app/ui/core';
import { cx } from '@todo-app/ui/utils';
import { useAuthenticityToken } from 'remix-utils';
import { todoCompleteMutation } from '../../../domain/todo/todo.
    complete.server';
import { todoCreateMutation } from '../../../domain/todo/todo.
    create.server';
import { todoListLoader, TodoListLoaderData } from '../../../
    domain/todo/todo.list.server';

export const loader: LoaderFunction = async ({ request }) => {
  return await todoListLoader(request);
```

```
};

export const action: ActionFunction = async ({ request }) => {
  const action = (await request.clone().formData()).get('action');

  if (action === 'create') {
    return await formAction({
      request,
      schema: TODO_CREATE_SCHEMA,
      mutation: todoCreateMutation,
      environment: { request },
      successPath: '/',
    });
  } else if (action === 'complete') {
    return await formAction({
      request,
      schema: TODO_COMPLETE_SCHEMA,
      mutation: todoCompleteMutation,
      environment: { request },
      successPath: '/',
    });
  }

  return new Response(null, { status: 400 });
};

export default function Index() {
  const { todos } = useLoaderData<TodoListLoaderData>();
  const csrf = useAuthenticityToken();

  return (
    <div className="flex flex-col space-y-8">
      <div className="flex flex-col space-y-8">
        <SectionTitle title="Create Todo" subTitle="Create a new
           todo, just provide the title and a description" />
        <Form
          useFieldsWrapper={true}
          fieldsWrapperCssClasses="grid grid-cols-1 gap-6 sm:
             grid-cols-2"
          schema={TODO_CREATE_SCHEMA}
          hiddenFields={[SESSION_PROPERTIES.csrf, 'action']}
          values={{ [SESSION_PROPERTIES.csrf]: csrf, action: '
             create' }}
          buttonLabel="Create Todo"
        />
      </div>
    <div className="flex flex-col space-y-8">
      <SectionTitle title="Todo List" subTitle="Here you can see
         all your todos" />
      <div className="grid grid-cols-1 gap-4 md:grid-cols-2 lg:
         grid-cols-3 xl:grid-cols-4">
        {(todos || []).map((todo) => (
          <Card
            key={todo.id}
            badge={{
              color: todo.completed ? 'primary' : 'gray',
              label: todo.completed ? 'Completed' : 'In Progress',
              position: 'right',
```

```
            size: 'sm',
          }}
          className="flex flex-1 flex-col"
          bodyClassName="flex flex-1 flex-col"
        >
          <div className="flex flex-1 flex-col">
            <div className="flex flex-1 flex-col">
              <h4 className="text-md font-medium text-gray-900
                 dark:text-white">{todo.title}</h4>
              <span className="text-sm font-normal text-gray-600
                  dark:text-gray-500">{todo.description}</span>
            </div>
            <div className="mt-3 border-t border-gray-200 pt-3
                dark:border-gray-700">
              {!todo.completed && (
                <Form
                  schema={TODO_COMPLETE_SCHEMA}
                  hiddenFields={[SESSION_PROPERTIES.csrf, 'id',
                      'completed', 'action']}
                  values={{
                    [SESSION_PROPERTIES.csrf]: csrf,
                    id: todo.id,
                    completed: true,
                    action: 'complete',
                  }}
                  buttonComponent={({ className, ...props }) =>
                     (
                    <BaseButton
                      type="submit"
                      className={cx(
                      className,
                      'gap-x-2 border-primary-400 bg-transparent
                          text-sm text-primary-500
                         transition-all duration-300 ease-in-out
                          hover:border-primary-700 hover:
                         text-primary-700 dark:
                         border-primary-600 dark:hover:
                         border-primary-300 dark:hover:
                         text-primary-300'
                      )}
                      {...props}
                      >
                        <CheckIcon className="h-5 w-5" />
                        Complete
                    </BaseButton>
                  )}
                />
              )}
              {todo.completed && (
              <Form
                schema={TODO_COMPLETE_SCHEMA}
                hiddenFields={[SESSION_PROPERTIES.csrf, 'id', '
                   completed', 'action']}
                values={{
                  [SESSION_PROPERTIES.csrf]: csrf,
                  id: todo.id,
                  completed: false,
                  action: 'complete',
```

```
                }}
                buttonComponent={({ className, ...props }) => (
                  <BaseButton
                    type="submit"
                    className={cx(
                    className,
                    'gap-x-2 border-gray-200 bg-transparent
                       text-sm transition-all duration-300
                       ease-in-out hover:border-gray-300 dark:
                       border-gray-600 dark:hover:border-gray-500'
                    )}
                    {...props}
                  >
                    <XMarkIcon className="h-5 w-5" />
                    Not ready yet
                  </BaseButton>
                )}
              />
              )}
              </div>
            </div>
          </Card>
          ))}
        </div>
      </div>
    </div>
  );
}
```

This code is a bit long, bit it's not complicated, as the most of the code is just UI code written in JSX, the React template language. At the beginning, we export a **loader** function, which is responsible for loading the data that will be used by the UI and we can consume the data using the **useLoaderData** hook. The **action** function is responsible for handling the form submission and it's using the **formAction** helper function that will execute the mutation and redirect the user to the success path.

The **Form** component is a wrapper around the **react-hook-form** library, which is a library that helps us to handle forms in React. We could only pass the schema as an property to the **Form** component, in that case we would get a very simple form, with all the configured properties. But we are also passing some other properties, like the **hiddenFields** property, which is an array of fields that will be hidden from the form and the **values** property, which is an object with the values that will be used to populate the form. This makes sure that the form will be populated with the correct values and that the hidden fields will be sent to the server but not shown to the user.

The rest is just UI code, but we don't go into details here, as React is a prerequisite for this project and we assume that the reader is familiar with it.

This quick example shows how easy it is to create a fully functional, user-friendly web application using CodeFlow. By leveraging the power of CodeFlow's tools and technologies, developers can rapidly develop applications that meet their specific requirements.

CodeFlow provides a solid foundation for building scalable and maintainable applications. With its intuitive code generation capabilities and modular architecture, developers can easily extend the application with additional features and functionalities. The generated codebase follows best practices in software engineering, ensuring clean and efficient code.

Furthermore, CodeFlow's seamless integration with popular frameworks and libraries such as React, Prisma and Tailwind CSS enables developers to leverage the rich ecosystem of tools and resources available in the frontend development community. This fosters productivity and empowers developers to create compelling user experiences.

The example provided here is just a glimpse of what can be achieved with CodeFlow. Its flexibility and extensibility make it suitable for a wide range of projects, from small prototypes to large-scale applications.
The following figure 5.11 shows a screenshot of the final application.

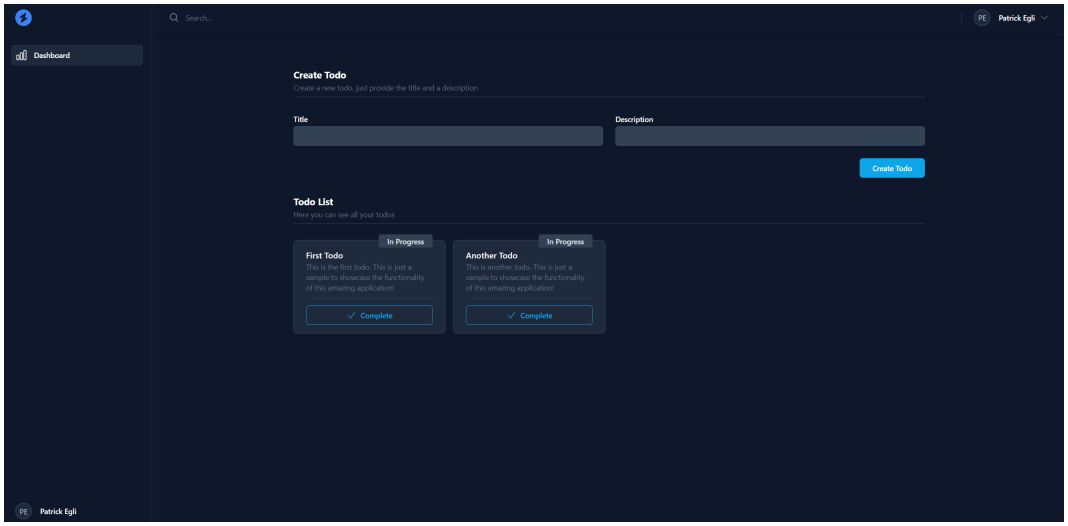


Figure 5.11: Final application

By now, developers should have a good understanding of how CodeFlow works and how it can be used to build web applications.

Chapter 6

Results

This chapter presents the outcomes of our evaluation involving our proposed solution CodeFlow, JHipster, OutSystems and Mendix. In order to facilitate a comprehensive comparison, we carefully implemented the same application across all four platforms and tools. Moreover, we imposed a time constraint of 4 hours for each platform and tool, enabling us to assess their performance within a standardized timeframe as defined in section 3.2.

By employing this unified approach, we gain valuable insights into the strengths and weaknesses of the various platforms and tools, as outlined by the predefined metrics described in section 3.1.

With these parameters in place, we are ready to conduct an in-depth analysis and comparison of the evaluated platforms and tools, shedding light on their performance, efficiency and frontend integration. By examining the results, we can provide developers with actionable information to guide their decision-making process when selecting the most suitable platform or tool for their specific project requirements.

- **Maintainability:** This refers to how easily a software tool can be updated or fixed. A tool with high maintainability will have clear, understandable code and a well-structured design that allows developers to easily understand and alter its functions as needed. This includes updating the tool to fix bugs, improve performance or add new features.
- **Performance:** Performance evaluation assesses the runtime performance of the tool, encompassing factors such as speed, responsiveness and resource consumption. A high-performance tool operates rapidly and efficiently, minimizing lag time and optimizing resource utilization.
- **Scalability:** This refers to the tool's capacity to handle increasing amounts of work and its potential to be enlarged to accommodate that growth. Tools that are scalable will still perform effectively even as the size or complexity of the software system grows.
- **Version control:** This aspect evaluates how well the tool manages different versions or iterations of the software. Good version control allows developers to track changes, rollback to previous versions when necessary and coordinate work between different team members.
- **Reusability:** This aspect relates to whether the tool (or parts of it) can be used multiple times for different purposes or projects. A reusable tool will have components that can be employed in various contexts, which can lead to more efficient and cost-effective development.

- **Extendability:** This refers to how easy it is to add new features or capabilities to the tool. A tool is extendable if it is designed in a modular way, which allows for additions and expansions in functionality without disrupting the existing system.
- **Documentation:** This evaluates the completeness, clarity and usefulness of the tool's documentation. High-quality documentation includes thorough explanations of the tool's functions, examples of how to use it and solutions to common problems.
- **Vendor Lock-in:** This relates to the extent to which users are dependent on a specific vendor for services and products. A tool with high vendor lock-in means users will have difficulty moving to a different product or vendor without substantial transition costs.
- **Deployment:** This evaluates how easy it is to distribute and implement the software created with the tool in a live environment. This might include considerations about the tool's compatibility with different systems, the ease of setting up the software and the process of updating the software once it's been deployed.
- **Frontend Integration:** This evaluates how easy it is to integrate a customer-facing frontend application with the given tool. The frontend application should be able to communicate with the backend application created with the tool.
- **Time to Market**: This evaluates how quickly a software tool can create a working application. A tool with a low time to market will allow developers to quickly create a functional application that can be deployed to production.

6.1 Addressed Challenges

This section provides an in-depth exploration of how the challenges presented in section 5 are not only acknowledged but also successfully addressed within the prototype. By delving into the specific strategies and solutions implemented, we aim to provide a comprehensive understanding of how the prototype effectively tackles these challenges head-on.

6.1.1 Maintainability

A web application framework should prioritize ease of maintenance and updates. The implemented prototype effectively addresses the challenge of maintainability by granting developers complete control over the source code and dependencies.

By having full control over the source code and dependencies, developers can easily update and modify the prototype as needed. This flexibility enables seamless updates to the source code and the integration of new dependencies, ensuring the prototype remains up-to-date and adaptable.

Additionally, the prototype places emphasis on comprehensive documentation of its source code. This documentation serves as a valuable resource, making it easier for developers to understand and maintain the codebase. With well-documented source code, developers can quickly navigate through the codebase, troubleshoot issues and make necessary updates or modifications with confidence.

By providing developers with control, flexibility and well-documented source code, the implemented prototype promotes the long-term maintainability of web applications. This focus on maintainability allows developers to efficiently manage updates, implement enhancements and ensure the continued reliability and functionality of the prototype.

6.1.2 Performance

The implemented prototype effectively addresses the performance challenge by employing a modular architecture. The modular architecture provides developers with the necessary flexibility to optimize the prototype for enhanced performance.

By utilizing a modular architecture, the prototype can be divided into separate modules or components, each with its specific functionality. This modularization enables developers to optimize individual modules or components to improve performance. They can focus on optimizing critical sections, enhancing algorithms and fine-tuning resource utilization to achieve optimal performance.

The modular architecture facilitates targeted performance optimizations by allowing developers to isolate and analyze specific areas of the prototype. By focusing on performance bottlenecks within individual modules, developers can apply tailored optimization techniques to enhance overall system performance.

Furthermore, the modular architecture promotes efficient resource utilization by enabling developers to allocate resources based on specific module requirements. This allows for effective management of computational resources, memory and other system assets, ultimately contributing to improved performance.

By leveraging a modular architecture and the associated optimization opportunities, the implemented prototype prioritizes performance enhancements. This approach empowers developers to fine-tune the system, ensuring that it delivers optimal performance while meeting the demands of the intended use cases.

6.1.3 Scalability

The implemented prototype effectively tackles the scalability challenge by leveraging a modular architecture. This modular architecture provides the developer with the necessary flexibility to scale the prototype and support the development of large software systems.

The modular architecture enables the prototype to be divided into separate modules or components, each responsible for specific functionalities. This modularization promotes loose coupling and high cohesion, allowing for easier management and expansion of the system as it grows in complexity. Developers can selectively scale individual modules without affecting the entire system, making it easier to accommodate evolving requirements and handle increased demands.

Furthermore, the prototype grants developers full control over the source code and dependencies, enhancing scalability capabilities. Developers have the freedom to adapt, optimize and extend the prototype as needed to meet the requirements of large-scale software systems. With complete control, developers can introduce optimizations, fine-tune performance and incorporate additional functionality to ensure scalability as the project expands.

By embracing a modular architecture and providing developers with control over the source code and dependencies, the implemented prototype lays a strong foundation for scalable software development. This approach empowers developers to effectively manage the growth and complexity of large software systems while maintaining flexibility, maintainability and performance.

6.1.4 Version Control

The implemented prototype effectively tackles the version control challenge by utilizing the Git version control system. By leveraging Git, developers gain precise control over the source code and dependencies, facilitating seamless management of versioning throughout the development process.

Git empowers developers with a robust set of version control features, allowing for efficient tracking of changes, seamless collaboration among team members and the ability to revert to previous versions when necessary. With Git, developers can maintain a comprehensive history of their project's evolution, ensuring transparency, accountability and the ability to navigate through different versions of the codebase.

By adopting Git as the version control system, the prototype enables developers to embrace industry-standard practices and workflows for managing code changes effectively. The use of Git instills confidence in developers, as they have full control over the source code and can easily manage and organize the versioning of their work.

Overall, the integration of Git into the prototype's development workflow provides developers with a powerful and flexible version control solution. This integration empowers developers to effectively manage and track changes, collaborate seamlessly and maintain a well-documented history of their project's progression.

6.1.5 Reusability

The implementation of a clean architecture design, along with the principles of separation of concerns and modularization, fosters low coupling and high cohesion within the codebase. This architectural approach ensures that different components are decoupled and have minimal dependencies on each other, promoting flexibility and maintainability.

By adhering to the principles of separation of concerns, each module or component focuses on a specific responsibility or functionality. This clear separation allows for better organization and comprehension of the codebase, making it easier to manage and maintain over time.

Modularization further enhances the codebase by promoting reusability. Shared functionality can be encapsulated within separate modules, enabling developers to reuse that functionality in multiple parts of the application. This approach eliminates code duplication, reduces development effort and improves overall code quality.

By embracing clean architecture, separation of concerns and modularization, the codebase benefits from improved maintainability, flexibility and extensibility. The low coupling and high cohesion achieved through these practices create a solid foundation for a scalable and adaptable software solution.

6.1.6 Developer Experience

While quantifying developer experience from an external perspective may be challenging, it remains a crucial factor when evaluating a web application framework. Recognizing the significance of developer experience, the implemented prototype takes specific measures to address this challenge and prioritize developers' needs.

Firstly, the prototype provides comprehensive documentation of the codebase. This documentation serves as a valuable resource, helping developers gain a deep understanding of the code and facilitating smoother development processes.

In addition to documentation, a video course has been created, covering both fundamental and advanced topics of the prototype. This course equips developers with the necessary knowledge and skills to effectively work with the prototype, including creating applications and extending functionality. By offering this comprehensive learning material, developers can quickly familiarize themselves with the prototype and maximize their productivity.

To streamline the development workflow, the prototype offers a CLI (Command-Line Interface) tool. This tool simplifies project creation and management, allowing developers to efficiently create new projects and effectively handle existing ones.

Furthermore, the prototype incorporates Storybook, enabling developers to preview UI components and visualize them in different states. This feature provides developers with a convenient overview of UI components, enhancing their ability to design and develop user interfaces effectively.

To further enhance the developer experience, the prototype employs automatic code formatting and linting. These tools ensure that the codebase remains consistent and adheres to best practices, promoting readability and maintainability.

Collectively, these features and tools work in tandem to enhance the overall developer experience of the prototype. By prioritizing clear documentation, comprehensive learning materials, streamlined workflows, visual component previews and code consistency, the prototype aims to provide a developer-centric environment that fosters productivity, ease of use and adherence to industry standards.

6.1.7 Extendability

By providing full access to the source code, the implemented prototype empowers developers to seamlessly extend its functionality. The prototype is intentionally designed with the expectation that developers will extend its capabilities to meet their specific requirements.

A key feature of the prototype is its templating mechanism, which enables developers to create new templates and expand upon existing ones. This mechanism provides a flexible framework for developers to customize and enhance the prototype according to their unique needs. By leveraging the templating system, developers can create new templates from scratch or build upon existing ones to tailor the prototype to their desired specifications.

With the ability to extend the prototype's functionality through source code access and the templating mechanism, developers have the freedom and flexibility to customize the prototype to suit their project's evolving needs. This adaptability ensures that the prototype remains a versatile and scalable solution capable of accommodating diverse development requirements.

6.1.8 Documentation

The implemented prototype effectively addresses the challenge of documentation by providing comprehensive documentation of the codebase. The majority of the codebase is carefully documented using JSDoc, enabling developers to gain a thorough understanding of the code's functionality, structure and usage.

To further support developers, we have also created a video course that covers both the fundamentals and advanced topics of the prototype. This course serves as a valuable resource, equipping developers with the knowledge and skills needed to leverage the prototype effectively. By providing this additional learning material, developers can quickly familiarize themselves with the prototype and begin utilizing its capabilities with confidence.

The combination of detailed codebase documentation and the supplementary video course ensures that developers have the necessary resources to utilize the prototype promptly. With these comprehensive learning materials at their disposal, developers can easily navigate the prototype, harness its features and streamline their development process.

6.1.9 Vendor Lock-in

The implemented prototype effectively mitigates the issue of vendor lock-in by leveraging open source technologies. By utilizing these technologies, the prototype ensures that developers retain complete control over the source code and dependencies, granting them the freedom to switch vendors whenever necessary.

The generated workspace further reinforces this flexibility by providing built-in support for Docker. This integration enables developers to deploy the generated web application seamlessly to any container engine that supports OCI images. This versatility opens up a plethora of deployment options, allowing developers to choose the environment that best suits their specific needs and preferences.

In addition to Docker integration, the prototype affords developers the choice to deploy the application without Docker. With full access to the source code and dependencies, developers have the autonomy to explore alternative deployment methods that align with their unique requirements.

This inherent support for open source technologies and the absence of vendor lock-in empowers developers to maintain control over their projects and adapt to changing circumstances. Developers can confidently make decisions regarding vendor selection, deployment strategies and future scalability, knowing that they have the freedom to choose the most suitable options throughout the development and deployment lifecycle.

6.1.10 Deployment

Deploying the prototype is a seamless process, facilitated by the built-in Docker integration within the generated codebase. This integration allows developers to effortlessly deploy the web application to any container engine that supports OCI images.

The deployment options for the generated web application are virtually limitless. With complete access to the source code and dependencies, developers have the freedom to deploy the application to any platform that supports either Node.js or Docker. This flexibility empowers developers to choose the deployment environment that best suits their specific project requirements and infrastructure preferences.

Whether deploying to a cloud-based platform, a self-hosted server or any other compatible environment, the prototype ensures developers retain full control over the deployment process. By leveraging the benefits of Docker and OCI images, developers can confidently deploy the application with ease, opening up a vast array of possibilities for hosting and scaling the web application according to their needs.

6.1.11 Frontend Integration

The implemented prototype effectively tackles the frontend integration challenge by offering a wide array of components, utilities and exceptional libraries like Storybook and React. By leveraging these resources, developers can seamlessly integrate the frontend with the backend, minimizing the need for excessive boilerplate code.

The prototype equips developers with an extensive collection of pre-existing components and utilities. These readily available building blocks empower developers to create a frontend that harmoniously integrates with the backend functionalities. By utilizing these pre-built elements, developers can accelerate their development process, saving valuable time and effort.

In addition to the pre-existing resources, the prototype also allows developers to create custom components and utilities. This flexibility enables developers to extend the functionality of the prototype according to their specific project requirements. By leveraging the ability to tailor and expand the prototype, developers can unlock new possibilities and cater to the unique needs of their frontend development.

In summary, the implemented prototype offers a robust solution for frontend integration challenges. With its rich collection of components, utilities and exceptional libraries, developers can seamlessly integrate frontend and backend functionalities while minimizing boilerplate code. The prototype's adaptability allows for the creation of custom components and utilities, further enhancing its versatility and empowering developers to achieve their desired frontend integration goals.

6.1.12 Time to Market

Our tool, CodeFlow, tackles the time-to-market challenge head-on by offering a convenient CLI (Command-Line Interface) tool. With just a single command, developers can create a new workspace rapidly and efficiently. This CLI command prompts the developer with a series of questions to customize the workspace according to their specific needs.

The resulting workspace encompasses a wide range of components and utilities carefully curated to facilitate the creation of exceptional web applications. Leveraging the power of renowned libraries and frameworks like Remix, React, Storybook and Tailwind CSS, CodeFlow generates a workspace that seamlessly integrates these tools. This integration significantly streamlines the development process, empowering developers to rapidly craft stunning web applications.

By automating the setup and configuration of the workspace, CodeFlow eliminates time-consuming manual tasks, enabling developers to focus on building innovative and captivating user experiences. Whether it's leveraging Remix for enhanced server-rendered React applications, harnessing the versatility of Storybook for interactive component development or utilizing the utility-first approach of Tailwind CSS for effortless styling, CodeFlow ensures these libraries and frameworks are readily available within the generated workspace.

With CodeFlow, developers can unlock their productivity potential, confidently embarking on the journey of web application development while benefiting from the pre-configured environment and the seamless integration of powerful tools and libraries.

6.2 Practical Results

This section presents the results of the evaluation of our own solution CodeFlow, JHipster, OutSystems and Mendix in respect to the generated sample application and the previously described metrics. To facilitate the evaluation process, we have devised a rating system on a scale of 1 to 3, where a score of 1 represents "Not good", 2 signifies "Ok" and 3 indicates "Good". This metric allows us to provide clear and concise assessments of each platform and tool, helping developers make informed decisions based on their specific requirements.

6.2.1 JHipster

JHipster is a powerful development platform designed for the generation, development and deployment of Spring Boot + Angular/React/Vue web applications. It offers the flexibility to create applications using either a monolithic or microservice architecture, while providing developers with a plethora of boilerplate code that accelerates application development.

One of the significant advantages of JHipster is that it grants developers complete access to the codebase and allows them to leverage state-of-the-art technologies. Assuming familiarity with JHipster's chosen technologies, developers can seamlessly integrate their preferred tools and libraries, enabling a streamlined development experience. Moreover, JHipster empowers developers with full control over the generated source code, facilitating easier maintenance and updates tailored to their specific needs.

In terms of performance, JHipster harnesses the power of robust frameworks and libraries like Spring Boot and Hibernate, renowned for their high-performance capabilities. Java, as a compiled language, undergoes bytecode compilation and is executed by the Java Virtual Machine (JVM). Notably, Java's Just-In-Time (JIT) compilation optimizes bytecode at runtime, further enhancing performance.

JHipster also embraces microservice architectures, enabling horizontal scalability. Developers can effortlessly deploy multiple instances of the application and effectively distribute the workload across these instances using load balancers. This scalability feature empowers applications to handle increased traffic and ensure smooth operation.

Adhering to a more traditional approach to software development, JHipster seamlessly integrates with popular version control systems like Git or SVN. This compatibility allows developers to track changes, revert to previous versions when necessary and collaborate efficiently among team members. As a result, developers can utilize the version control system they are most familiar with, promoting a cohesive workflow.

JHipster's extensive boilerplate code facilitates accelerated development by allowing developers to focus more on the business logic of their applications. The provided boilerplate code is reusable, allowing developers to leverage it across various services and components, further enhancing productivity.

While JHipster leverages the well-established Java programming language, known for its large developer community and abundance of libraries and frameworks, it is important to note that Java may lack certain features present in newer languages like Kotlin or Go. Consequently, Java's relatively verbose type system may require developers to write more code to achieve the same results, potentially impacting the developer experience.

JHipster's development platform offers the flexibility to extend generated web applications with additional features. This can be accomplished through the utilization of provided JHipster modules or by writing custom code. Leveraging the capabilities of the Spring Boot framework, developers can create extensive codebases and seamlessly incorporate supplementary functionalities.

The comprehensive documentation provided by JHipster serves as a valuable resource, offering detailed insights into its various features. In addition to the official JHipster documentation, developers can access a wide array of tutorials and blog posts that provide additional information and guidance. Furthermore, documentation for associated technologies such as Spring Boot, Angular, React and Vue is readily available, supported by vibrant communities offering tutorials, courses and blog posts.

As an open-source project, JHipster offers developers freedom from vendor lock-in. The generated source code can be utilized without any restrictions and developers have the flexibility to deploy their applications on any server of their choice. Whether deploying to a cloud provider or a self-hosted server, developers retain complete control over the deployment process.

One notable limitation of JHipster is its lack of a dedicated customer-facing frontend application. While JHipster excels at generating robust backend applications complemented by comprehensive administration frontends, it does not provide a dedicated frontend for customers to interact with. This limitation can prove challenging for developers seeking to create fully-fledged web applications that encompass user-facing interfaces.

In today's landscape, where frontend applications are progressively growing in complexity, the absence of a customer-oriented frontend solution is a notable drawback. The demand for intuitive, user-friendly interfaces has become paramount, emphasizing the importance of having a frontend application that is both accessible and navigable.

To address this limitation, developers utilizing JHipster may need to explore additional frontend frameworks or technologies to build a customer-facing interface that meets their specific requirements. By integrating these tools alongside JHipster's backend capabilities, developers can create a more comprehensive and user-centric web application experience.

The absence of a dedicated customer-facing frontend application in JHipster significantly impacts the time-to-market of the generated application. Developers are faced with the additional task of researching and integrating frontend technologies, which can result in delays in the application's release. This limitation necessitates extra time and effort to ensure the frontend meets the desired functionality and user experience standards.

Overall, JHipster presents a robust development platform, combining the power of established frameworks and libraries, extensive documentation and the flexibility to adapt and extend applications. Its emphasis on developer productivity and choice, combined with performance optimization and scalability, make it a compelling option for building Spring Boot + Angular/React/Vue web applications.

Criteria	Score
Maintainability	3
Performance	3
Scalability	3
Version Control	3
Reusability	3
Developer Experience	2
Extendability	3
Documentation	3
Vendor-Lock In	3
Deployment	3
Frontend Integration	1
Time to Market	1

Table 6.1: Scoring for JHipster

6.2.2 OutSystems

OutSystems is a low-code development platform that empowers developers to build web and mobile applications using a visual development environment. The platform offers an array of features, including a visual development environment, a visual modeling language and a visual debugger.

The visual development environment provided by OutSystems allows developers to create applications by simply dragging and dropping components onto a canvas. This visual approach to development eliminates the need for extensive coding, resulting in significant time and effort savings. Additionally, OutSystems' visual modeling language, Reactive Web, enables developers to design responsive web applications that seamlessly adapt to various devices. This capability ensures accessibility across desktops, tablets and mobile phones.

However, a limitation of the platform is that developers do not have direct access to the generated application's source code. While the visual development environment allows some adjustments to the application's structure, full control over shaping the application to specific requirements is limited.

In terms of performance analysis, developers using OutSystems rely on the platform's proprietary visual modeling language, which restricts the ability to analyze source code performance directly. While performance issues were not encountered during implementation, developers must trust in OutSystems' performance optimizations.

OutSystems' modular architecture provides scalability benefits by allowing developers to create applications using multiple modules. This modular approach enhances scalability and is complemented by built-in support for vertical and horizontal scaling, as well as a load balancer for efficient load distribution across multiple servers.

While OutSystems offers a built-in version control system, it lacks advanced features compared to Git. The system creates a new version of the application upon publishing, but branching, merging and cherry-picking capabilities are limited.

The platform provides a rich set of reusable components that accelerate application development. Developers can leverage pre-existing components and create their own, reducing the need to build every component from scratch.

In terms of developer experience, our experience found OutSystems' visual development environment to be unintuitive and cumbersome, with some tasks being more efficiently accomplished using code rather than the visual interface. However, developer experience is subjective and less experienced developers may find the envi-

ronment more user-friendly.

Being a closed-source platform, OutSystems limits developers' ability to extend the platform beyond its provided features. While the platform offers a broad range of functionalities, extending it beyond the available capabilities is not possible.

OutSystems provides extensive documentation and a wide range of tutorials, facilitating learning and usage of the platform. The well-structured documentation allows developers to easily find the information they need.

One notable limitation of OutSystems is the vendor lock-in it imposes. Applications generated using OutSystems' proprietary visual modeling language are not compatible with other platforms, making migration challenging. Switching platforms would require rebuilding the application from scratch, as the source code cannot be reused.

OutSystems offers diverse deployment options, including deployment to their cloud infrastructure or on-premise installations, allowing developers to choose the environment that best suits their needs.
Regarding frontend integration, OutSystems simplifies the process by enabling developers to create responsive web applications through visual component manipulation. Connecting the frontend to the backend is facilitated by the built-in REST API. However, styling components can be challenging, as developers need to familiarize themselves with OutSystems' visual editor.

Considering the experience during implementation, challenges were encountered due to the learning curve associated with OutSystems. Given more familiarity with the platform, completing the application within the allocated timeframe would likely have been achievable.

In summary, OutSystems is a low-code development platform that offers a visual development environment, visual modeling language and various deployment options. While it simplifies development and provides scalability benefits, limitations such as limited control over source code, vendor lock-in and some challenges with the visual environment should be considered when evaluating the platform for specific project requirements.

Criteria	**Score**
Maintainability	2
Performance	3
Scalability	3
Version Control	2
Reusability	3
Developer Experience	2
Extendability	1
Documentation	3
Vendor-Lock In	1
Deployment	3
Frontend Integration	3
Time to Market	3

Table 6.2: Scoring for OutSystems

6.2.3 Mendix

Mendix is a low-code development platform that empowers developers to create web and mobile applications using a visual development environment. The platform offers a range of features, including a user-friendly visual development environment, a visual modeling language and powerful debugging capabilities.

Mendix's visual development environment allows developers to build applications by intuitively dragging and dropping components onto a canvas. This approach streamlines development by reducing the need for extensive coding, resulting in significant time and effort savings. Moreover, Mendix's visual modeling language enables the creation of responsive web applications that seamlessly adapt to different devices, ensuring optimal user experiences across desktops, tablets and mobile phones.

However, a limitation of the platform is that developers do not have direct access to the underlying source code of the generated applications. While adjustments to the application's structure are possible within the visual environment, full control over shaping the application to meet specific requirements is limited.

In terms of performance analysis, Mendix provides performance optimization features within its visual modeling language. Developers can leverage these features to optimize application performance, although the level of control over low-level source code optimizations may be constrained compared to traditional development approaches. Nonetheless, our experience with Mendix has not encountered performance issues during implementation.

Mendix's modular architecture allows developers to create applications using multiple modules, promoting scalability and flexibility. The platform offers built-in support for vertical and horizontal scaling, along with load balancing capabilities, enabling efficient load distribution across multiple servers.

Regarding version control, Mendix incorporates a versioning system that tracks changes made to the application over time. While the version control capabilities may not be as advanced as dedicated version control systems like Git, developers can effectively manage changes and rollbacks within the Mendix environment.

Mendix provides an extensive library of reusable components, empowering developers to accelerate application development. These pre-built components, combined with the ability to create custom components, significantly reduce the need to build every aspect of an application from scratch.

In terms of developer experience, Mendix's visual development environment is generally well-regarded for its user-friendliness. However, the learning curve associated with any new platform may present challenges and certain complex tasks may be more efficiently accomplished using code rather than relying solely on the visual interface.

As a closed-source platform, Mendix restricts developers from extending the platform beyond its provided features. While the platform offers a broad range of functionalities, customization beyond these capabilities is not possible.

Mendix offers comprehensive documentation and a wide array of tutorials, providing developers with the necessary resources to learn and effectively use the platform. The well-structured documentation enables easy access to information, assisting developers in leveraging Mendix's features and functionalities effectively.

It's important to note that Mendix's proprietary visual modeling language may impose a level of vendor lock-in. Applications developed using Mendix may not be easily transferable to other platforms, potentially requiring redevelopment from scratch

if a switch is desired.

Mendix offers flexible deployment options, including cloud infrastructure and on-premise installations, allowing developers to choose the deployment environment that aligns with their specific needs and preferences.

When it comes to frontend integration, Mendix simplifies the process by enabling developers to create responsive web applications through visual component manipulation. The platform provides a built-in REST API to facilitate seamless communication between the frontend and backend systems. However, styling components within Mendix's visual editor may present challenges that require developers to familiarize themselves with the platform's specific styling capabilities.

Considering the learning curve associated with any new platform, our experience with Mendix during implementation revealed initial challenges. However, with increased familiarity, completing applications within the allocated timeframe would likely be achievable.

In summary, Mendix is a low-code development platform offering a user-friendly visual development environment, visual modeling language and various deployment options. While Mendix simplifies development and provides scalability benefits, it also has limitations that should be considered for specific project requirements.

Criteria	**Score**
Maintainability	2
Performance	3
Scalability	3
Version Control	2
Reusability	3
Developer Experience	2
Extendability	1
Documentation	3
Vendor-Lock In	1
Deployment	3
Frontend Integration	3
Time to Market	3

Table 6.3: Scoring for Mendix

6.2.4 CodeFlow

CodeFlow is our proposed solution for the development of a modern web application. This code generation tool fulfills all the key metrics in software engineering and also delivers great results in the frontend integration as well as in time to market.

With CodeFlow, developers can easily generate an entire workspace by utilizing a CLI tool. This tool enables the creation of a customized workspace with a single command, allowing developers to provide input for the desired workspace configuration. Based on this input, CodeFlow generates the complete workspace, including the necessary boilerplate code. Moreover, developers have the freedom to create multiple applications within the workspace, promoting flexibility and scalability.

One of the key strengths of CodeFlow is its provision of full access to the generated codebase. Developers can adjust the source code as needed, enhancing the maintainability of the application. Furthermore, developers have the option to extend or modify the code generator itself. By tweaking the generator to their specific requirements and regenerating the codebase, developers ensure that future projects can benefit from these customized adjustments.

The generated codebase leverages the power of Remix, a robust framework for building modern web applications. Built on top of React and Node.js, Remix utilizes the single-threaded event-driven JavaScript runtime of Node.js, which is performant enough for many use cases. This runtime enables the development of scalable network applications, real-time applications, streaming applications and more.

CodeFlow supports the creation of multiple applications within a single workspace, facilitating the implementation of a microservices architecture. This approach allows developers to horizontally scale the application by adding more instances, ensuring flexibility and efficient resource utilization.

To ensure efficient version control, CodeFlow integrates Git as the chosen system for the generated codebase. Developers can leverage all the features provided by Git, including branching and merging, while also benefiting from seamless collaboration within a team of developers. Additionally, developers have the flexibility to host the codebase on their preferred Git provider, such as GitHub, GitLab or Bitbucket.

The generated workspace in CodeFlow includes a comprehensive set of libraries and utilities designed for reuse across the entire application. This promotes code reusability, enabling developers to avoid reinventing the wheel for common components and utilities.

Developer experience is a priority for CodeFlow and the tool comes equipped with a range of features to enhance productivity. Automatic code formatting, linting, excellent IDE support and comprehensive documentation are all part of the developer experience offered. By integrating these tools and libraries, developers can focus on writing code and building applications without the hassle of setting up the development environment.

CodeFlow builds on the foundation of NX workspaces, a collection of powerful tools for mono-repo development. With NX, developers can create extensible mono repositories that can accommodate multiple applications and libraries. This scalability enables easy extension of the existing workspace with new applications and libraries as needed.

To ensure that developers can quickly understand and navigate the codebase, CodeFlow places great emphasis on documentation. The majority of the codebase is thoroughly documented, providing developers with clear insights into the functionality and structure of the application. Additionally, a video course has been created to

guide developers through the usage of CodeFlow, enabling them to quickly grasp the tool's capabilities and get started efficiently.

Recognizing the increasing significance of frontend development, CodeFlow includes a fully functional frontend application built on the powerful Remix framework. Remix adopts a more traditional approach to web application development, leveraging server-side rendering, progressive enhancement and the inherent capabilities of the browser. This approach enables simplified data fetching and mutations, allowing developers to focus on the business logic and user interface design rather than intricate technical details.

CodeFlow generates a complete application that encompasses essential features of modern web applications, such as authentication, authorization, service worker integration, internationalization, theming and database integration. By handling these technical aspects, CodeFlow enables developers to concentrate on the core business logic of the application, reducing the burden of integrating various tools and libraries. This streamlined development process significantly accelerates time to market, enabling developers to deliver their applications quickly and efficiently.

In conclusion, CodeFlow is a comprehensive solution for modern web application development. It empowers developers by providing a robust code generation tool that addresses key metrics in software engineering. With its CLI tool, developers can effortlessly create customized workspaces and applications. By leveraging the power of Remix and Node.js, CodeFlow ensures high-performance applications that are scalable and adaptable.

Criteria	**Score**
Maintainability	3
Performance	3
Scalability	3
Version Control	3
Reusability	3
Developer Experience	3
Extendability	3
Documentation	3
Vendor-Lock In	3
Deployment	3
Frontend Integration	3
Time to Market	3

Table 6.4: Scoring for CodeFlow

6.2.5 Comparison

The table below presents a comprehensive comparison of various software development platforms. To simplify the evaluation process, we have implemented a rating system on a scale of 1 to 3, with 1 indicating "Not good", 2 representing "Ok" and 3 denoting "Good". This standardized metric enables us to provide concise and insightful assessments of each platform and tool, empowering developers to make well-informed decisions tailored to their specific needs.

Criteria	Codeflow	JHipster	Mendix	Outsystems
Maintainability	3	3	2	2
Performance	3	3	3	3
Scalability	3	3	3	3
Version Control	3	3	2	2
Reusability	3	3	3	3
Developer Experience	3	2	2	2
Extendability	3	3	1	1
Documentation	3	3	3	3
Vendor-Lock In	3	3	1	1
Deployment	3	3	3	3
Frontend Integration	3	1	3	3
Time to Market	3	1	3	3

Table 6.5: Comparison of different software development platforms

Based on the evaluation of different software development platforms, we have gathered valuable insights regarding their performance across various criteria. CodeFlow emerges as a strong contender, consistently scoring highly in all evaluated areas, including maintainability, performance, scalability, version control, reusability, developer experience, extendability, documentation, vendor lock-in, deployment, frontend integration and time to market. This indicates that CodeFlow offers a comprehensive solution that excels in all key aspects of modern web application development.

JHipster also showcases strong performance in many areas, particularly in maintainability, performance, scalability, version control and reusability. It provides developers with full control over the source code and dependencies, allowing for easy maintenance and customization. However, it lags behind in terms of frontend integration and time to market, as it does not offer a dedicated customer-facing frontend application.

Mendix and OutSystems demonstrate competence in certain areas, such as scalability, documentation and deployment. However, they fall short in terms of extendability and vendor lock-in, as developers have limited control and customization options.

Overall, CodeFlow emerges as the preferred choice for developers seeking a comprehensive solution that excels in key metrics while providing full control, extensive documentation and an excellent developer experience.

6.2.6 Sample Application

In this section, we would like to provide an overview of the sample application, "TeamUp", that we developed using all four platforms. TeamUp is a collaborative

platform designed to facilitate project creation and foster connections among users with shared interests.

The application allows users to create projects by providing essential details such as a title, description, related study course and a list of tags. By scrolling through a list of projects, users can explore various topics and find projects that align with their interests. Interactivity is a key feature of TeamUp, enabling users to engage with projects by liking or commenting on them.

The primary objective of TeamUp is to connect students who are interested in similar subjects, particularly in cross-course collaborations. By facilitating the discovery of like-minded individuals from different courses, the application aims to bridge the gap and foster collaboration among students who share common interests. This functionality is particularly valuable in facilitating connections that would otherwise be challenging to establish.

TeamUp serves as a valuable tool for finding and connecting with students who share similar academic interests, promoting collaboration and knowledge-sharing across different study courses. Through this application, users can discover new project opportunities, engage in meaningful discussions and expand their network within the academic community.

6.2.7 Comparing the final Applications

In this section, we will compare the final applications developed using different platforms, namely CodeFlow, JHipster, OutSystems and Mendix. Each platform offers unique features and capabilities for rapid application development and we will outline what we could achieve using each platform.

CodeFlow

As we were most familiar with our own solution, we began the development process by creating the first application using CodeFlow. Within the given time frame of 4 hours, we successfully built a fully functional application that is user-friendly, responsive, intuitive and ready for deployment.

The application includes an authentication system, allowing users to sign up and log in to their accounts securely. To ensure the security of all requests, we implemented CSRF protection, which authenticates and authorizes each request. All data is stored in a PostgreSQL database, hosted on the same server as the application, ensuring fast and efficient data retrieval.

CodeFlow empowers us to easily extend the application with new features and functionalities by leveraging the tools and technologies it provides. Additionally, we prioritize the reliability and scalability of our application through the use of Prisma, which enables us to create and apply database migrations seamlessly. These migrations are automatically applied when the application is deployed to production.

To enhance session management, we utilize Redis to store session data, which is also hosted on the same server as the application. This allows for efficient session handling and improves the overall performance of the application.

We invite you to access the application at the following URL: `https://team-up.appdesigns.pro/`. Explore the functionality and experience firsthand how CodeFlow enables the rapid development of robust, user-friendly web applications.

JHipster

After successfully developing the application using CodeFlow, we decided to replicate the same application using JHipster. However, we encountered several challenges during the development process that affected our ability to complete the application within the given time frame.

One notable challenge we faced was the lack of a dedicated frontend application provided by JHipster. Unlike CodeFlow, which generates a complete web application encompassing both frontend and backend functionality, JHipster generates a backend application that serves as an API for the frontend. As a result, we had to develop the frontend separately, which required additional time and effort.

Furthermore, due to time constraints, we were unable to deploy the JHipster application. This, combined with the fact that the generated application primarily served as an administrative application rather than a user-facing one, influenced our decision to focus our efforts on other aspects of the project. While JHipster excels in generating a robust backend and a comprehensive admin control panel, it did not align with our goal of building a complete user-facing application.

Given the constraints of the time frame and the specific requirements of our project, we prioritized the development and deployment of the applications generated by CodeFlow, Mendix and OutSystems, which provided a more comprehensive solution for user-facing applications.

OutSystems

Next, we proceeded to develop the application using OutSystems, which proved to be a powerful platform for rapid application development, encompassing both the frontend and backend aspects of the application. Leveraging the wide range of tools and technologies provided by OutSystems, we were able to build a functional application within the given time frame.

Although we encountered some challenges during the development process that required additional time and effort, we were able to deploy a functional application. However, due to the time constraints, we were unable to implement all the features that were present in the application developed using CodeFlow. But, we firmly believe that with more experience with the OutSystems platform, we would be able to develop a fully functional application that meets all the requirements.

We invite you to explore the application we built using OutSystems by visiting the following URL `https://personal-ndced8cv.outsystemscloud.com/TeamUp/Login`. This application showcases the capabilities of OutSystems in delivering a fully functional and user-friendly web application.

Mendix

Lastly, we proceeded to develop the application using Mendix, a robust platform known for its rapid application development capabilities. Our experience with Mendix closely mirrored that of OutSystems, as both platforms offered comprehensive solutions for efficient application development.

During the development process, we encountered similar challenges that necessitated additional time and effort to overcome. As a result, we were unable to implement all the features present in the application developed using CodeFlow. However, we are confident that with more experience and familiarity with the Mendix platform, we would be able to create a fully functional application that fulfills all the specified requirements.

To explore the application developed with Mendix, please visit the following URL: `https://mendix-prototype-app-sandbox.mxapps.io/login.html`. This application showcases the potential of Mendix in delivering rapid and intuitive web applications.

6.3 Interpretation of the results

Low code development platforms offer advantages in terms of fast prototyping, enabling developers to quickly build applications with minimal coding effort. However, one notable challenge with low code platforms is the potential difficulty in extending them with custom functionality due to vendor lock-in. While they provide an efficient way to create applications rapidly, developers may encounter limitations when attempting to tailor the platform to specific requirements.

In terms of version control, low code platforms often offer proprietary solutions, which may not provide the same level of flexibility and robustness as proven solutions like Git. Traditional approaches, on the other hand, can leverage the power of established version control systems, allowing developers to take advantage of advanced features such as branching, merging and cherry-picking.

JHipster stands out for its ability to generate extensive boilerplate code and provide a comprehensive admin control panel. However, a notable drawback is the absence of a dedicated frontend application for end customers. This limitation necessitates additional effort from developers to create a customer-facing frontend, making it an area where further development and customization are required.

In contrast, our solution, CodeFlow, follows a more traditional approach to software engineering. By granting developers full access to the codebase, CodeFlow allows them to integrate any tools they prefer, assuming compatibility with the generated codebase. This flexibility empowers developers to utilize their preferred technologies and libraries, enabling seamless integration and customization to meet specific project requirements.

During our evaluation, CodeFlow enabled us to build the most feature-rich application within the given time frame. By supporting both frontend and backend capabilities, CodeFlow provides a comprehensive solution that facilitates the development of complex web applications. The ability to leverage a traditional approach, coupled with the flexibility to integrate preferred tools, contributed to our successful implementation.

In conclusion, while low code platforms excel in rapid prototyping, they may pose challenges when extending functionality due to vendor lock-in. Traditional approaches, such as CodeFlow, offer greater flexibility and control over the codebase, allowing for seamless integration of preferred tools and libraries. By leveraging a more traditional approach, developers can build robust and feature-rich applications while maintaining the ability to customize and extend them as needed.

6.4 Suggestions

Based on our evaluation and comparison of the different software development platforms, we offer the following suggestion to the reader:

Consider your specific project requirements and priorities when selecting a development platform. If rapid prototyping and customization are crucial, CodeFlow may be an excellent choice due to its modular architecture, full codebase access and support for key software engineering metrics. However, keep in mind that familiarity with the associated tools and technologies, such as TypeScript, Remix, Prisma, React and Tailwind CSS, is essential.

For those looking for fast backend prototyping with a great admin dashboard, JHipster can be a viable option. However, be prepared to develop the user-facing frontend separately and ensure you have the necessary expertise in Java, Spring Boot, JPA and related technologies.

If simplicity and built-in components are your priorities, OutSystems and Mendix offer efficient prototyping and ease of use. However, be cautious of the vendor-lock in, limited codebase access and potential limitations in addressing all key software engineering metrics.

Ultimately, the best choice depends on your project's specific needs, team expertise and long-term goals. We encourage you to thoroughly evaluate each platform's strengths and weaknesses and choose the one that aligns most closely with your requirements and vision.

Chapter 7

Conclusion

In conclusion, our thesis has explored the landscape of software development platforms and tools, focusing on their strengths and limitations and proposing a new tool called CodeFlow. Throughout this journey, we have uncovered valuable insights and made significant contributions to the field of rapid application development.

CodeFlow has demonstrated its strengths as a powerful tool for efficient prototyping and development of modern web applications. It has proven that it is possible to achieve rapid development without solely relying on low-code platforms. By following a more traditional approach to software engineering, CodeFlow empowers developers with full access to the codebase, flexibility in choosing technologies and libraries and the ability to integrate any desired tools.

Through our evaluation and comparison of CodeFlow with other platforms, including JHipster, OutSystems and Mendix, we have highlighted the unique features and benefits of each tool. While low-code platforms offer certain advantages in terms of ease of use and rapid initial setup, CodeFlow has showcased its capability to deliver highly customizable and scalable applications, tailored to specific project requirements.

One of the key findings of our research is that low-code platforms may present limitations in terms of vendor lock-in, limited customization options and challenges in extending the platform with custom functionality. CodeFlow has overcome these limitations by providing developers with full control over the codebase, modularity and the ability to integrate various tools and libraries.

Furthermore, CodeFlow has demonstrated its effectiveness in prototyping, enabling developers to rapidly build fully functional applications within the given time frame. Its support for frontend and backend capabilities, comprehensive tooling and adherence to key metrics in software engineering have made it a reliable and efficient tool for developers.

In conclusion, CodeFlow has paved the way for a new paradigm in rapid application development. It has proven that efficiency in prototyping and development can be achieved without relying solely on low-code platforms. By providing developers with the flexibility and control to tailor their applications to specific requirements, CodeFlow empowers them to create high-quality, scalable and customized web applications. Having full control over the codebase, developers can leverage their preferred technologies and libraries, enabling seamless integration and customization to meet specific project requirements.

7.1 Summary of the research questions and objectives

The research conducted in this thesis aimed to answer the following research questions:

RQ1: What are the key challenges associated with rapid software development and how do current low-code and no-code platforms address these challenges?

RQ2: How can a new code generator tool be designed and implemented to generate high-quality, reusable code for both backend and frontend development?

RQ3: What are the advantages and disadvantages of the new tool in comparison to existing code generators and low-code platforms and how can these be addressed to optimize its usability and effectiveness?

In response to **RQ1**, the key challenges identified in rapid software development include maintainability, scalability, performance, version control, reusability, developer experience, extendability, documentation, vendor lock-in, deployment, frontend integration and time-to-market. Current low-code platforms attempt to address these challenges through features such as visual development environments, code generation, built-in components and libraries, automated documentation, version control systems and deployment options. However, they may also introduce limitations in terms of customization, flexibility and code ownership. On the other hand, current code generator tools such as JHipster create an entire codebase that is fully customizable and extendable. However, JHipster does not provide a customer-facing frontend application. Therefore, it requires additional development efforts to build a frontend that can be used by end users.

RQ2 focused on the design and implementation of a new code generator tool, CodeFlow, to tackle these challenges. CodeFlow was designed as a comprehensive solution for backend and frontend development, leveraging the power of TypeScript, Remix, Prisma, React, Tailwind CSS and other technologies. The tool aims to enable efficient prototyping and code generation, emphasizing code quality, adherence to software engineering best practices and the generation of reusable code. Its modular architecture and full codebase access provide developers with the flexibility to integrate custom tools and extend its functionality.

Addressing **RQ3**, CodeFlow exhibits several advantages over existing code generators and low-code platforms. It allows for efficient prototyping without strict vendor lock-in, empowering developers to customize and adapt the codebase to their specific requirements. CodeFlow's comprehensive documentation, automated code formatting and linting features enhance developer experience and maintainability. However, it should be noted that CodeFlow requires familiarity with TypeScript and the integrated technologies and it may require additional adjustments for specific frameworks or libraries.

In conclusion, this research has provided insights into the key challenges of rapid software development, how current code generators and low-code platforms address these challenges and the design and implementation of CodeFlow as a maintainable and customizable code generator tool. CodeFlow demonstrates that efficient prototyping and high-quality code generation can be achieved without solely relying on low-code platforms, offering developers a flexible and extensible solution for rapid application development. The findings highlight the strengths and limitations of CodeFlow in comparison to existing platforms and suggest measures to optimize its usability and effectiveness.

7.2 Contributions of the study

In this chapter, we present our contributions based on the evaluation and comparison of different software development platforms. Our primary objective was to introduce a new tool that facilitates the rapid development of modern full-stack web applications, while also harnessing the power of key software engineering metrics. Our key contributions include:

1. Evaluation Framework: We developed an evaluation framework that encompasses key metrics in software engineering. This framework allowed us to assess each platform based on maintainability, performance, scalability, version control, reusability, developer experience, extendability, documentation, vendor lock-in, deployment, frontend integration and time to market.

2. CodeFlow: We introduced CodeFlow, our proposed solution for modern web application development. CodeFlow offers a CLI-based code generation tool that enables developers to quickly set up customized workspaces and applications. With its modular architecture, full codebase access and support for key software engineering metrics, CodeFlow provides developers with the flexibility and control needed for efficient development.

3. Platform Comparisons: We conducted in-depth comparisons of CodeFlow with other popular low-code and traditional software development platforms, including JHipster, OutSystems and Mendix. By analyzing their strengths and weaknesses, we aimed to provide a comprehensive understanding of each platform's capabilities and limitations.

4. Sample Applications: We built sample applications on each platform to showcase their functionalities and demonstrate their suitability for various project requirements. These applications, including the TeamUp application developed with CodeFlow, allowed us to explore and evaluate the platforms in real-world scenarios.

5. Recommendations: Based on our evaluations, we provided recommendations and suggestions to help developers make informed decisions when selecting a software development platform. We highlighted the benefits and drawbacks of each platform, considering factors such as prototyping capabilities, codebase access, customization options, vendor lock-in and support for software engineering metrics.

We believe that these contributions will serve as a valuable resource for developers seeking guidance in choosing the right software development platform. By considering our evaluations, recommendations and insights, developers can make informed decisions that align with their project requirements, team expertise and long-term goals.

7.3 Limitations of the study

The chapter on limitations aims to shed light on certain constraints and considerations associated with the use of CodeFlow. While CodeFlow offers numerous benefits and features, it is important to acknowledge the following limitations:

Language Restriction: CodeFlow currently exclusively supports TypeScript as the primary programming language. While TypeScript offers strong typing and other advantages, if your project requires the use of a different programming language such as Java or Python, CodeFlow may not be the most suitable choice.

Given the limited development time for CodeFlow, although it has shown to be an effective tool for accelerated development, there are still certain features that need to be addressed. Specifically, the inclusion of a Command Line Interface (CLI) that directly integrates with the target workspace, enabling direct code creation within the workspace. Such an enhancement would significantly improve the development process, further enhancing the efficiency of CodeFlow. This CLI could generate schemas automatically by inferring the Prisma schema provided by the developer or it could automatically generate business logic based on the provided schema and the developer could then customize the generated code.

Design Style Limitations: CodeFlow provides a set of built-in components that follow a specific design style. While these components offer a cohesive and consistent user interface, it's important to note that they may not align perfectly with your unique design requirements. Customizing these components to meet specific design needs may necessitate modifications across multiple components, potentially adding complexity and development effort.

Dependency Constraints: CodeFlow relies on various dependencies such as Remix, React, Tailwind CSS, Storybook and others to enable its functionality. While these dependencies provide powerful capabilities and tooling, if you prefer to use alternative frameworks or libraries (e.g., Next.js instead of Remix), creating a custom generator becomes necessary. This requires additional development effort and expertise to ensure seamless integration with the desired dependencies.

Database Compatibility: CodeFlow leverages the power of Prisma, which supports a wide range of databases, including PostgreSQL, SQLite, Oracle, Microsoft SQL Server, MySQL, MariaDB, AWS Aurora, AWS Aurora Serverless, Azure SQL, CockroachDB and MongoDB. However, it's important to note that if your project requires the use of a non-supported database, you would need to replace Prisma with an alternative database integration solution. This may introduce additional development effort and potentially impact the seamless integration and functionality provided by CodeFlow.

These limitations should be taken into consideration when evaluating the suitability of CodeFlow for your specific project requirements. It is crucial to assess whether these constraints align with your desired programming language, design flexibility, dependency preferences and database compatibility.

7.4 Future Work

The future of software development holds immense potential and one key ingredient that is guaranteed to revolutionize the field is Artificial Intelligence (AI). AI has the capability to play a crucial role in the automated software generation process, opening up new possibilities and opportunities for developers.

With the advancements in AI technologies, we can envision a future where intelligent algorithms and machine learning models can analyze vast amounts of code, data and user requirements to generate high-quality software solutions. AI-powered tools can assist developers in automating repetitive tasks, generating boilerplate code and even suggesting optimized design patterns based on specific project requirements. One of the most exciting aspects of AI in software development is its ability to learn from existing codebases and leverage that knowledge to enhance the development process. By analyzing large repositories of open-source projects, AI algorithms can identify patterns, best practices and efficient solutions that can be applied to generate code snippets or provide intelligent suggestions during development.

Furthermore, AI can greatly contribute to improving the quality of software through automated testing and bug detection. Machine learning models can learn from historical data to identify potential bugs or vulnerabilities in the code, enabling developers to proactively address these issues before they manifest in the production environment.

In addition to code generation and testing, AI can also enhance the user experience by automating user interface design, natural language processing for requirements gathering and intelligent recommendation systems for personalized software solutions.

However, it is important to recognize that the integration of AI into software development processes comes with its own set of challenges. Ethical considerations, data privacy and the need for interpretability and explainability of AI algorithms are critical factors that must be carefully addressed.

As we look to the future, it is clear that AI will continue to shape and transform the software development landscape. Embracing AI technologies and exploring their potential applications can lead to more efficient, productive and innovative software development processes. By harnessing the power of AI, we can unlock new levels of automation, optimization and creativity, paving the way for a future where software development becomes more intelligent, collaborative and adaptive.

In addition to the exciting prospects of AI in software development, we can envision the integration of AI technologies into our own tool, CodeFlow, to further enhance its capabilities. By leveraging the power of pretrained transformer models, we could enable the AI to scan the entire CodeFlow workspace, gaining knowledge and insights from the codebase.

With this knowledge, the AI could assist developers by generating individual code snippets that are tailored to specific development scenarios. These code snippets could provide intelligent suggestions, offer optimized implementations of common tasks or even propose alternative solutions based on best practices learned from the codebase.

Imagine a developer working on a feature and encountering a specific coding problem. The AI-powered CodeFlow tool could analyze the context, understand the problem and generate a relevant code snippet that solves the issue efficiently. This would save developers valuable time and effort, accelerating the development process and

promoting code consistency.

Furthermore, the AI could continually learn and adapt based on developer feedback and usage patterns within the CodeFlow workspace. Over time, the AI would become more proficient at providing accurate and helpful code suggestions, aligning with the specific coding style and preferences of the development team.

However, it is essential to approach the integration of AI into CodeFlow with care. Ensuring the privacy and security of the codebase, protecting sensitive information and maintaining a transparent and understandable AI system are crucial considerations. Striking the right balance between automation and developer control will be key to creating a productive and trusted AI-enhanced development environment.

By combining CodeFlow with AI technologies, we believe to unlock a new level of developer productivity and efficiency. The AI-powered code generation capabilities would augment developers skills and expertise, enabling them to leverage the collective knowledge of the codebase to deliver high-quality software solutions faster and more effectively.

Bibliography

[1] A. Bucaioni, A. Cicchetti, and F. Ciccozzi, "Modelling in low-code development: A multi-vocal systematic review," *Software and Systems Modeling*, vol. 21, Jan. 2022.

[2] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and challenges of low-code development: The practitioners' perspective," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21, Bari, Italy: Association for Computing Machinery, 2021, ISBN: 9781450386654. DOI: 10.1145/3475716.3475782. [Online]. Available: https://doi.org/10.1145/3475716.3475782.

[3] T. C. Lethbridge, "Low-code is often high-code, so we must design low-code platforms to enable proper software engineering," in *Leveraging Applications of Formal Methods, Verification and Validation*, B. Margaria Tizianaand Steffen, Ed., Cham: Springer International Publishing, 2021, pp. 202–212, ISBN: 978-3-030-89159-6.

[4] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, "Supporting the understanding and comparison of low-code development platforms," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178. DOI: 10.1109/SEAA51224.2020.00036.

[5] F. Fagerholm and J. Münch, "Developer experience: Concept and definition," in *2012 International Conference on Software and System Process (ICSSP)*, 2012, pp. 73–77. DOI: 10.1109/ICSSP.2012.6225984.

[6] T. C. Lethbridge, "Low-code is often high-code, so we must design low-code platforms to enable proper software engineering," in *Leveraging Applications of Formal Methods, Verification and Validation*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2021, pp. 202–212, ISBN: 978-3-030-89159-6.

[7] D. Dahlberg, "Developer experience of a low-code platform: An exploratory study," M.S. thesis, Umeå University, Faculty of Social Sciences, Department of Informatics, 2020.

[8] *Jhipster*, Accessed: 2023-03-25. [Online]. Available: https://www.jhipster.tech/.

[9] *Outsystems low-code platform*, Accessed: 2023-03-25. [Online]. Available: https://www.outsystems.com/low-code-platform/.

[10] *Mendix low-code application development platform*, Accessed: 2023-03-25. [Online]. Available: https://www.mendix.com/.

[11] *Remix - build better websites*, https://remix.run/, (Accessed on 06/06/2023).

[12] *React*, https://react.dev/, (Accessed on 06/06/2023).

[13] *Prisma | next-generation orm for node.js & typescript*, https://www.prisma.io/, (Accessed on 06/06/2023).

[14] *Tailwind css - rapidly build modern websites without ever leaving your html.* https://tailwindcss.com/, (Accessed on 06/06/2023).

[15] *Storybook: Frontend workshop for ui development*, https://storybook.js.org/, (Accessed on 06/06/2023).

[16] *Typescript: Javascript with syntax for types.* `https://www.typescriptlang.org/`, (Accessed on 06/06/2023).

[17] *Nx: Smart, fast and extensible build system*, `https://nx.dev/`, (Accessed on 06/06/2023).

[18] *Docker: Accelerated, containerized application development*, `https://www.docker.com/`, (Accessed on 06/06/2023).

[19] *Using workflows - github docs*, `https://docs.github.com/en/actions/using-workflows`, (Accessed on 06/06/2023).

[20] *Zod - typescript-first schema validation with static type inference*, `https://github.com/colinhacks/zod`, (Accessed on 06/06/2023).

List of Figures

5.1 Overview 31
5.2 Project Diagram 34
5.3 Project Diagram 35
5.4 Applications 37
5.5 Dependency Graph 39
5.6 CodeFlow CLI 42
5.7 CodeFlow Dashboard 43
5.8 CodeFlow Storybook instance of the ui/core library 44
5.9 CodeFlow Storybook instance of the ui/app library 45
5.10 Storybook custom UI component 55
5.11 Final application 59

Use of AI Tools

In our thesis development process, we employed various tools to assist our work. Initially, we used ChatGPT to create the outline, which proved to be a valuable starting point. This allowed us to organize our thoughts and structure the content effectively. It is important to note that while ChatGPT assisted us in formulating the initial structure, the content of the thesis itself is a result of our own ideas, research, and analysis.

Furthermore, we leveraged GitHub Copilot, as both of us have integrated it into our IDEs. However, it is crucial to clarify that we did not rely on ChatGPT or GitHub Copilot for generating sources or expressions. Instead, we utilized them solely to rephrase certain sentences, ensuring that the context remained unchanged and no additional information was added.

Throughout the process, we carefully reviewed the generated output and made necessary adjustments when required. The ultimate aim was to maintain the integrity of our own ideas and thoughts. The content of this thesis primarily consists of our original contributions, and we have only refined the wording of paragraphs to enhance readability.