www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Improving the Developer Experience when Building Web Applications

Ingo Andelhofs

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Maarten WIJNANTS

BEGELEIDER :
De heer Wouter LEMOINE
De heer Hendrik LIEVENS

2022
2023

Faculteit Wetenschappen
School voor Informatietechnologie
master in de informatica

Masterthesis

Improving the Developer Experience when Building Web Applications

Ingo Andelhofs

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Maarten WIJNANTS

BEGELEIDER :
De heer Wouter LEMOINE
De heer Hendrik LIEVENS

Universiteit Hasselt

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Improving the Developer Experience when Building Web Applications

Auteur:

Ingo Andelhofs

Promotor:

Dr. Maarten Wijnants

Co-promotor:

/

Begeleider(s):

Dhr. Wouter Lemoine
Dhr. Hendrik Lievens

Academiejaar 2022-2023

Acknowledgements

I would like to express my sincere gratitude to the following individuals whose unwavering support and invaluable contributions have made this thesis possible.

First and foremost, I am deeply thankful to my supervisor, Dr. Maarten Wijnants, and advisors Dhr. Wouter Lemoine and Dhr. Hendrik Lievens, for their exceptional guidance, insightful feedback, and continuous encouragement throughout the entire research journey. Even when I struggled to find a path to completion for this thesis, they were consistently there to provide me with new ideas and alternative routes to continue working on the subject. I genuinely appreciate the assistance they provided in guiding me to the point where this thesis stands today.

I am also indebted to those who assisted me with beta testing the toolset that was developed and for their insightful feedback. This has profoundly impacted the thesis. Moreover, it affirmed that I was on the right track with my creative endeavors. I am truly grateful for the support I received on such short notice.

I extend my heartfelt appreciation to my family, friends, and fellow students who offered encouragement, shared their perspectives, and dedicated their time to reviewing my work. Your collective input has been invaluable to me.

I wish to acknowledge the University of Hasselt for providing the essential resources, facilities, and academic environment that nurtured an atmosphere of learning and exploration.

Lastly, I extend my thanks to all those who played a part, no matter how small, in shaping the outcome of this thesis.

I am immensely thankful to all of you for being integral to this remarkable journey.

Summary

In this thesis about ‘Improving the Developer Experience when Building Web Applications’, we explore the possibilities to provide a better DX when building applications for the web.

We have done this by researching existing web technologies like HTML, CSS, and JavaScript and followed by a more thorough search through more modern web technologies, like components, front-end libraries and frameworks, styling libraries, reusable components, back-end libraries and frameworks, and client-server code-sharing libraries. By learning about these technologies, we understood the web development landscape and what tools could be used together to build modern applications for the web. We learned that we need to have a client and server in most cases and what libraries could provide this functionality. This helped us to group different technologies together. This gave us a better understanding of what technologies can be used where and how they interact with each other.

This knowledge was used with information about the development lifecycle to create a toolset to provide the developer with tooling to build applications for the web efficiently. When looking at the development lifecycle, we can determine the different states we always go through when building applications. We must first define our requirements and prepare to build the web application. This can also include picking the correct technologies you will use to build the project. This is then followed by initializing the code for the given project. This step initializes the basic structure of your application. Next, we have the development phase. This is where we develop the back-end and front-end of our application. We also need to make sure that the client and server can communicate with each other. This is another important step in the lifecycle. Lastly, we should be able to test and deploy the application.

To improve the developer experience we build a couple of tools to help the development process at every phase. The first tool we created was the initialization tool. This tool provides the developer with a consistent and easy way to initialize a project, a framework, or libraries. This tool can thus be used to set up a project given the requirements. The tools provided allow us to install a barebones version of a framework into the project we are working on and initialize libraries based on the context or the framework we use. This allows us to install and configure libraries without knowing how to install them for a given framework. This takes away complexity for the developer. This allows the developer to automate the manual steps of setting up a project correctly and configuring everything manually.

The second tool is the generate tool. This tool allows the developer to generate boilerplate code for the common patterns that occur in your codebase. Examples are components, pages, and resources. This tool tries to minimize the code that needs to be written by the user while also being context-aware, meaning that the code generated can depend on the installed framework, libraries, and more. This helps the developer focus on the features rather than considering how certain code should be written in a given framework. Because the generate tool still requires manual execution of the command, the watch tool was created. This tool allows you to generate code based on where you create files in your filesystem. For example, if you create a ‘.tsx’ file, which is a React component file, we could generate the code for a React component. This allows for context-aware snippets.

A similar tool is the introspect tool. This tool requires an extra step. Instead of just generating code, the introspect tool will first retrieve more information about the current state of the codebase. For example, when we have written a back-end, we probably already have a lot of code that explains how the back-end works, so it would be useful to reuse this information. This can be done by using introspection. The introspect tool first parses the backend code to retrieve information about how to call the API. This information can then be used to generate code for a client adaptor written in TypeScript, for example. This tool is useful if you don't want to do the same work twice.

Lastly, we created an interaction tool. This tool allows you to interact with the source code as well as interact with the application that is created by running the source code. This is possible because of introspection as well. Because we can retrieve knowledge about the code already written we can use this information to provide a visual user interface for the developer to interact with. This allows the developer to test out the application without defining anything. The source of truth is the application written by the developer itself.

While developing the tools, we conducted two types of evaluation. An internal test, where we iteratively tested the toolset ourselves, and a beta test, where test users tried out the toolset to provide feedback for further improvements.

The internal testing allowed us to improve the toolset because we would use the toolset to solve a simple problem and identify the limitations this way. This was a useful method because we could see for ourselves where the toolset shined. When we identified the problems, we improved the toolset and repeated the process of building a simple application. This allowed us to build a useful toolset for efficiently building web applications without the hassle of figuring out how to integrate different systems and libraries.

On the other hand, the beta test gave us feedback on the elements of the toolset, which we may not have thought of as the developers of the toolset. This showed us that we needed to ensure the documentation and the help information were clear. We also discussed methods of feedforward and the undo functionality. This would be useful to make sure we could always revert certain actions if the user had a different idea of what was about to happen. Overall, the testers were impressed by how efficiently we could create a working web application from scratch with a working back-end, database, and front-end. This showed us the relevance of the work we have done once more.

Samenvatting

In deze scriptie met als onderwerp 'Het Verbeteren van de Ontwikkelaarservaring bij het Bouwen van Webapplicaties', onderzoeken wij de mogelijkheden om een verbeterde ontwikkelaarservaring te faciliteren bij het creëren van applicaties voor het web.

Dit is bewerkstelligd door een grondige verkenning van bestaande webtechnologieën zoals HTML, CSS en JavaScript. Deze initiële verkenning is gevolgd door een diepgaandere verkenning van modernere webtechnologieën, waaronder componenten en herbruikbare elementen, front-end bibliotheken en frameworks, stijlbibliotheken, back-end bibliotheken en frameworks, en bibliotheken voor het delen van client-servercode. Door bekend te raken met deze technologieën hebben wij een meer omvattend inzicht verkregen in webontwikkeling en in de manier waarop diverse tools in combinatie kunnen worden ingezet voor het creëren van moderne webapplicaties. Hierbij hebben wij opgemerkt dat in de meeste gevallen zowel een client als een server noodzakelijk zijn, alsmede welke bibliotheken in deze functionaliteit kunnen voorzien. Deze verkenning heeft ons in staat gesteld om diverse technologieën te categoriseren en heeft ons begrip vergroot over welke technologieën waar ingezet kunnen worden en hoe zij onderling interageren.

De verworven kennis is in samenhang met informatie over de ontwikkelingslevenscyclus gebruikt om een toolkit te ontwikkelen die de ontwikkelaar voorziet van hulpmiddelen om efficiënt webapplicaties te creëren. Bij het overzien van de ontwikkelingslevenscyclus kunnen diverse stappen worden vastgesteld die consistent doorlopen worden bij het ontwikkelen van applicaties. De eerste stap is het definiëren van de vereisten en het voorbereiden van het bouwproces van de webapplicatie. Dit omvat tevens het selecteren van geschikte technologieën voor het project. Hieropvolgend wordt de code geïnitialiseerd voor het betreffende project, wat de basisstructuur van de applicatie vastlegt. Daarna volgt de ontwikkelingsfase, waarin zowel de back-end als de front-end van de applicatie ontwikkeld worden, en waarbij tevens gezorgd moet worden voor onderlinge communicatie tussen client en server. Deze fase is cruciaal in de levenscyclus. Ten slotte is er de fase van het testen en hosten van de webapplicatie.

Met het oog op het optimaliseren van de ontwikkelaarservaring hebben wij diverse tools ontwikkeld om het ontwikkelproces gedurende elke fase te faciliteren. Als eerste hebben wij de 'initialisatie tool' gecreëerd. Deze tool biedt ontwikkelaars een gestandaardiseerde en gebruiksvriendelijke methode om projecten, frameworks of bibliotheken te initialiseren. Hierdoor kan een project opgezet worden op basis van specifieke vereisten. De verstrekte hulpmiddelen stellen ons in staat om een basisversie van een framework te installeren en bibliotheken te initialiseren in lijn met de context van het gebruikte framework. Hierdoor kunnen bibliotheken geïnstalleerd en geconfigureerd worden zonder de noodzaak om gedetailleerde kennis te hebben van de installatieprocedure binnen een specifiek framework. Dit vereenvoudigt het proces voor de ontwikkelaar en automatiseert het correct opzetten van projecten ten opzichte van handmatige configuratie.

Als tweede tool hebben wij de 'generatie tool' ontwikkeld. Deze tool stelt ontwikkelaars in staat om boilerplate te genereren voor veelvoorkomende patronen binnen de codebase. Voorbeelden hiervan zijn componenten, pagina's en resources. Deze tool beoogt de hoeveelheid door de gebruiker te schrijven code te minimaliseren, waarbij tevens rekening gehouden wordt met de context. De gegenereerde code kan afhankelijk zijn van het geïnstalleerde framework,

de gebruikte bibliotheken en andere factoren. Hierdoor kan de ontwikkelaar zich richten op de functionaliteit in plaats van de specifieke wijze waarop bepaalde code binnen een gegeven framework geschreven dient te worden. Aangezien het gebruik van de ‘generatie tool’ nog steeds handmatige uitvoering vereist, hebben wij de ”watch tool” in het leven geroepen. Deze tool stelt ontwikkelaars in staat om code te genereren op basis van de locaties waar bestanden in het bestandssysteem aangemaakt worden. Bijvoorbeeld, wanneer een ‘.tsx’-bestand aangemaakt wordt, wat doorgaans een bestand voor een React-component vertegenwoordigt, kan automatisch de bijbehorende code gegenereerd worden. Dit draagt bij aan het contextbewust genereren van codefragmenten.

Een vergelijkbare tool is de ‘introspect tool’. Deze tool vereist een extra stap. In plaats van uitsluitend code te genereren, zal de ‘introspect tool’ eerst aanvullende informatie verzamelen over de huidige status van de codebase. Indien er bijvoorbeeld reeds een back-end geschreven is, is het waarschijnlijk dat er al veel code bestaat die uitleg verschaft over de werking van de back-end. Het hergebruik van deze informatie kan nuttig zijn. Dit wordt bereikt door middel van introspectie. De ‘introspect tool’ analyseert eerst de backend-code om te achterhalen hoe de API aangeroepen moet worden. Vervolgens kan deze informatie benut worden om code te genereren voor een TypeScript-clientadapter. Deze tool komt van pas wanneer dubbel werk vermeden dient te worden.

Tot slot hebben wij de ‘interactie tool’ ontworpen. Deze tool stelt ontwikkelaars in staat om te interacteren met de applicatie die gecreëerd wordt door de broncode uit te voeren. Dit is mogelijk dankzij introspectie. Doordat wij informatie vergaren over reeds geschreven code, kunnen wij deze informatie aanwenden om een visuele gebruikersinterface te bieden waarmee ontwikkelaars kunnen interacteren. Dit geeft ontwikkelaars de mogelijkheid om de applicatie uit te proberen zonder noodzaak tot expliciete definities. De bron van waarheid is de broncode van de applicatie die door de ontwikkelaar zelf gecreëerd is.

Tijdens de ontwikkeling van de tools hebben wij twee vormen van evaluatie uitgevoerd. Allereerst hebben wij een interne test uitgevoerd, waarbij we het toolkit iteratief zelf hebben getest. Vervolgens hebben we een bètatest uitgevoerd, waarbij testgebruikers de toolkit hebben uitgeprobeerd om feedback te verstrekken ter bevordering van verdere verbeteringen.

De interne test heeft ons in staat gesteld om de toolkit te verfijnen, aangezien wij deze hebben toegepast om een eenvoudig probleem op te lossen, waardoor beperkingen konden worden geïdentificeerd. Deze aanpak was uitermate vruchtbaar, aangezien wij uit eerste hand hebben kunnen zien waar de toolkit uitblonk. Zodra wij de knelpunten hadden vastgesteld, hebben wij de toolkit verbeterd en het proces van het ontwikkelen van een eenvoudige applicatie herhaald. Dit heeft ons in staat gesteld om een waardevolle toolkit te ontwikkelen voor het doeltreffend creëren van webapplicaties, zonder de complexiteit van het configureren van de verschillende systemen en bibliotheken.

Aan de andere kant heeft de bètatest ons waardevolle feedback verstrekt met betrekking tot aspecten van de toolkit die wellicht niet waren overwogen in onze rol als ontwikkelaars van de toolkit. Dit heeft duidelijk gemaakt dat we aandacht moesten besteden aan heldere documentatie en gebruiksinstructies. Tevens hebben wij gesprekken gevoerd over de implementatie van voorwaartse feedback en de functionaliteit van herstelbewerkingen. Deze overwegingen waren essentieel om ervoor te zorgen dat gebruikers te allen tijde in staat waren om acties ongedaan te maken, mochten zij een andere opvatting hebben over de verwachte uitkomst. Over het algemeen heeft het positieve oordeel van de testers ons getoond hoe effectief wij een functionerende webtoepassing konden genereren, geheel vanaf de basis, inclusief een operationele backend, database en frontend. Dit heeft nogmaals benadrukt hoe waardevol en relevant ons werk is geweest.

Contents

Chapter 1

Introduction

In the ever-evolving landscape of web application development, optimizing and improving the Developer Experience (DX) has emerged as a critical focus area. A seamless DX not only boosts developer productivity but also contributes to the overall success of web applications.

This master thesis will explore how we can contribute to "Improving the Developer Experience when building Web Applications". As a fundamental first step, a thorough exploration of the web development lifecycle is essential. This will help us understand the modern web development landscape and give us insights into modern technologies and why they are used.

The lifecycle includes things such as initialization of the project and packages needed to build a modern web application, the different tools needed to accomplish this, the different libraries available to help solve common problems, recurring patterns that we blindly repeat when developing our web application, the process we use to structure our project, the process we use to be consistent throughout the development process and more.

However, because the landscape of web development and the number of packages available is overwhelming, we will only be able to explore a fixed set of frameworks, libraries, and technologies. To limit our scope, we will use the "Stack Overflow" survey [44] to define the state-of-the-art technologies. To enrich our exploration, we will also include a set of newer and more unconventional technologies. By incorporating these emerging paradigms, we encourage more valuable insights and inspiration for further improvements.

In conclusion, this thesis will dive into "Improving the Developer Experience when Building Web Applications". A better DX, resulting in higher developer productivity, as well as increasing the overall success of web applications, is an important aspect of modern competitive markets. This thesis will explore the possibilities for improvement in DX of web application development by researching the modern development lifecycle and individually improving the aspects that build up that lifecycle. We will do this by including both modern and more experimental technologies in our journey to strive for a better DX.

1.0.1 Research Questions

RQ1: What tools can we create to improve the DX when building web applications?

RQ2: What libraries can we create to improve the DX when building web applications?

RQ3: How can we reduce the amount of boilerplate code we write?

RQ4: What types of patterns can we abstract?

RQ5: Can we generate boilerplate code for these abstracted patterns?

RQ6: What are the problems with generating boilerplate code?

RQ7: What techniques can be used to build web applications efficiently?

RQ8: What techniques can be used to build web applications consistently?

1.0.2 Chapters

The first chapter presents an overview of the literature pertaining to the subject of the thesis. The primary objective of this chapter is to establish a solid foundation by examining the core web technologies upon which subsequent advancements have been built. This chapter will explore various concepts such as components, reusability, and code sharing, as well as highlight modern as well as emerging frameworks, libraries, or technologies. This will go in hand with a critical evaluation of their utility and inherent constraints. Through this comprehensive analysis, the reader will gain valuable insights into the evolution and current state of web technologies, providing essential context for the subsequent chapters.

The second chapter presents a comprehensive examination of the development lifecycle in the context of web application development. Building upon the foundational insights provided in the preceding chapter, this chapter redirects the attention toward the development process itself. The primary focus is to capture the traditional development methodology, discovering the distinct stages when building modern web-based applications. Furthermore, the chapter will incorporate the exploration of the various tools and methodologies we could use at each stage of the development lifecycle. By documenting these tools and practices, developers can streamline their workflow and possibly optimize the overall efficiency of the development process.

In the next chapter, we will focus on the practical implementation of the toolset discussed in the previous chapter. We will provide different tools for the different stages in the development process and combine them in a single Command Line Interface (CLI) to gain easy access to the different tools. This chapter aims to give you a better understanding of how such a toolset could work but also dives deeper into the internal architecture of the toolset itself.

In the fourth chapter, we will provide you with an in-depth examination of the evaluation process employed in this thesis. This chapter will introduce the specific evaluation methods that could be adopted for the toolset, elaborating on the rationale behind their selection. However, the primary objective of this chapter is to conclude with a method of evaluation that can be used to evaluate the toolset subjectively. To accomplish this, we conducted two distinct types of evaluation, namely, a form of self-evaluation and a beta test involving other developers. Finally, we will present the reader with the results obtained from both evaluations.

In the fifth chapter, we will arrive at the conclusion of this thesis. This chapter will present conclusive insights obtained from the entirety of this research journey. By revisiting the research questions established in the introduction, we strive to provide a comprehensive and well-grounded response based on the knowledge acquired. In addition to addressing the research questions, this section will offer a broader perspective of using the toolset from a developer's perspective. This encapsulates not only theoretical implications but also the practical relevance of the toolset being used when building web applications.

In the final chapter, we will explore potential routes for future work based on the comprehensive examination of both the improvements achieved and the identified limitations. Acknowledging that this research can be further refined, this chapter aims to offer valuable insights to guide subsequent investigations in this domain. By providing the reader with this information, we aim to provide knowledge that could advance and improve the current state of web development technologies with a focus on improving the DX when building applications for the web.

Chapter 2

Literature

This literature review aims to look deeper into the world of modern web development and the various technologies that have reshaped the way developers build web applications. At the core of this rapid cycle of new emerging technologies lies the concept of DX, which encompasses all aspects of a developer's development life cycle, from the initial project setup to deployment and maintaining the application. This thesis will explore possible factors that influence the DX, focusing on web technologies, such as frameworks and libraries.

Before we can envision an improved DX in web application development, it is crucial to comprehend the foundational technologies that support this process. Hypertext Markup Language (HTML) provides the essential structure and semantics defining the overall skeleton of the web page. Cascading Style Sheets (CSS) provide a way to style the skeleton to build visual designs and layouts. JavaScript enables interactivity, dynamic content, client-server communication, and more, making it the backbone of modern web development. An ecosystem of web libraries and frameworks has emerged to augment development speed and efficiency. These libraries and frameworks range from small, specialized libraries to full-fledged frameworks.

In this chapter, we will look at these modern web technologies. Discuss their use cases and possible limitations while giving the reader a general overview of the web development landscape.

2.1 Web Technologies

In this section, we will discuss the core web technologies. We will introduce HTML, CSS, and JavaScript. We will also look at more modern technologies that are used more frequently, like TypeScript and JavaScript Extensible Markup Language (JSX).

2.1.1 HTML

HTML [24] defines the structure of a web application. HTML consists of a set of HTML elements that can be represented as HTML tags. A tag consists of < and > with the element's name in between. An HTML element can contain other HTML elements and thus represent structure and meaning. HTML is generally used in combination with CSS and JavaScript, which we discuss in the next sections.

2.1.2 CSS

CSS [8] is a styling language used to describe a web page's presentation and layout. It allows developers to define styles for HTML elements, such as fonts, colors, spacing, positioning, and more. Modern CSS thus allows us to create responsive and adaptive layouts that provide a seamless browsing experience. Another benefit of using CSS is that we can update the visual appearance of the web application without altering the underlying HTML, which could enhance the overall maintainability. Although CSS can also be used with Extensible Markup Language (XML) and some other XML dialects, in the context of web development, CSS is used with HTML.

2.1.3 JavaScript

JavaScript, or JS [26], is a scripting language widely used for front-end and back-end web development. It is an interpreted programming language, meaning the code is interpreted directly by the browser. However, modern browsers support Just In Time (JIT) [28] compilation, which compiles some parts of the code into bytecode at runtime. This can improve the performance in certain cases. JavaScript is used to describe a web page's functionality or behavior. The ECMAScript Language Specification standardizes JavaScript [14].

As a client-side language, JavaScript enables interactive and dynamic features on the web. Because it runs directly in the browser, we can utilize JavaScript to respond to user actions in real time without communicating with a server. Developers can also implement their own logic on the client to handle content manipulation or implement any desirable interactive or dynamic feature.

On the server, JavaScript can run using a JavaScript runtime. The most well-known JavaScript runtime is Node.js [34]. Node.js can handle both front-end and back-end tasks and has an extensive set of libraries that offers a rich ecosystem for developers to use.

2.1.4 Web APIs

When using the above web technologies, we can access a set of web APIs [59]. These APIs are accessible via JavaScript and have different end goals.

The Document Object Model (DOM) [12] Application Programming Interface (API) allows the developer to access the DOM. The DOM is a representation of the HTML document. Each DOM element or HTML tag is represented as a DOM node. The DOM API can be useful for dynamically updating the web page's structure. We could perform actions such as adding text to a DOM node, removing a DOM node, searching for DOM nodes, and more. Before modern JavaScript frameworks, the DOM API updates your web page dynamically.

The Canvas API [7] allows the developer to render graphics to the HTML canvas element. This can be useful when rendering complex graphics on the screen.

The Web Worker API [62] allows the developer to create worker threads that do not block the main thread. This can be useful when you need to do intense calculations in the browser without blocking the user from interacting with the web page.

There are additional APIs available that won't be covered in this discussion. For a comprehensive list of web APIs, refer to the web API reference [60] provided by Mozilla.

However, a problem with this set of APIs is that not all browsers support every API or even parts of a given API. This can be a problem when working with (older) browsers that do not support the newest APIs. Some APIs are complex or feel incomplete. We can improve these APIs by wrapping each API into a more usable API which could improve the developer experience when working with these web APIs.

2.1.5 Web Assembly

Recently the web has supported a new standard (by the W3C Web Assembly Working Group) called Web Assembly or WASM [63]. It is a low-level assembly-like language that can be run on the web. This allows for running any language that can be compiled to Web Assembly.

2.1.6 TypeScript

TypeScript or TS [53], is a superset of JavaScript. This means that any valid JavaScript is also valid TypeScript. However, this does not mean every TypeScript file will pass the type-checking phase.

TypeScript introduces optional static typing and enhanced features like classes, interfaces, and enums. The benefit of TypeScript is that its static typing enables IDEs to provide intelligent code suggestions and catch potential errors during development instead of at runtime. This can greatly improve the developer experience because we can catch errors early before running the JavaScript code. This will result in fewer runtime exceptions and difficult-to-find errors. This can also mean that your code is easier to maintain and scale.

TypeScript is transpiled to JavaScript. This means that the TypeScript code is transformed into JavaScript code. We can also specify a transpilation target. This is the JavaScript or ECMAScript version we want to transpile to. This means that older systems that only can run older versions of JavaScript could also run the transpiled TypeScript code. This allows us to use TypeScript everywhere, where we could previously use JavaScript. While also benefiting from the enhanced features that TypeScript provides.

2.1.7 JSX

JavaScript XML or JSX [64], is an extension of JavaScript primarily used by component-based frameworks like React. JSX allows the developer to create and define the structure of user interfaces in an HTML-like syntax directly in a JavaScript file. This means we could combine the structural UI elements with JavaScript control flow elements, such as loops, conditionals, and more. The JSX syntax promotes component-based development, which could benefit reusability and maintainability.

2.2 Components

This section will discuss the need for components when developing web applications. A problem with traditional HTML is that we can only create HTML files, each representing a web page. This is not a big problem when building a small website. However, when we wanted to create a more complex website that contains repeated structures, we could not separate them into different files. This resulted in HTML files that contained a lot of duplicate structures.

Another reason why we needed to repeat these HTML structures were dynamic websites. Dynamic websites are websites that are rendered based on dynamic data. An example is a Todo website, where a user can create todos. Because we want to list all the todos created by the user, we need to render the todos dynamically into the webpage. Note that each todo will probably look the same and thus have the same HTML structure.

Template Engines

Because the HTML standard did not allow Web Components or other methods of reusing a part of the HTML structure, template engines [55] were created. A template engine is a method that allows you to create static template files with syntactic sugar (for loops, conditionals, ...). Then at runtime, when a file is requested, the template engine replaces variables in the template file and evaluates other constructs, thus transforming the template into a static HTML file. This HTML file could then be sent to the client, making it simpler to create complex HTML structures while reusing partial structures. This shows us the importance of Components for the web.

Examples of template engines are PUG [19] and Blade [4], which is used by Laravel [29]. These template engines allow alternative syntax, directives like if statements and loops, and more.

Web Components

Web Components [61] is a web API that allows the user to create custom HTML elements. This improves the reusability of custom markup structures which was not possible or user-friendly in standard HTML. By allowing the developer to create reusable components, the developer could abstract the internal information into the custom component without worrying about messy and dangerous code. We can also see that modern frameworks use the concept of components by using technologies like JSX [25], which we will discuss later. This is often more developer-friendly than plain Web Components API.

Reusable Components

Headless UI [21], or similar headless UI libraries, are libraries that provide completely unstyled components or component helpers that are fully accessible and can be integrated with most styling systems. These components include dropdown menus, toggles, dialogs or modals, tabs, slides, and more. These libraries can also include things like transitions or other utility helper functions. Other examples of these helpers are 'useToggle', 'useClickOutside', and more. These can then be used to create a custom dropdown menu yourself. You can use the 'useToggle' to open and close the dropdown. However, you may also want to close the dropdown when you click outside the component. You can then use 'useClickOutside' to implement this behavior without writing your own code. The main benefit of these libraries is that they provide great accessibility out of the box and allow the developer to use their own custom styles without the problem of conflicting with the styles of the library, as seen with other styling libraries.

Other libraries include **React Radix** [38], **React Aria** [40] and more.

2.3 Front-end Frameworks

Front-end frameworks, referred to as client-side frameworks, are programming frameworks designed to tackle the ever-evolving demands of user interface (UI) design and interaction with these user interfaces. A front-end framework can be seen as a set of pre-written code and standardized practices that streamline the creation of client-side applications for the web. Some frameworks also refer to themselves as front-end libraries. The main difference between a framework and a library lies in the level of abstraction and the degree of freedom they provide to the developer. A library often provides more flexibility than a framework that pre-defines a set of conventions and rules the developer should follow. However, in this section, we will focus on the concept that a front-end framework is a library that provides a way for the developer to create web-based user interfaces in a more structured way.

2.3.1 React.js

React.js [41], commonly referred to as React, is an open-source JavaScript library used for building user interfaces. Developed by Facebook and open-sourced in **2013**, React allows developers to create reusable UI components and manage the state of their applications. React uses a declarative approach, meaning that developers specify the UI based on the application's state rather than directly manipulating the DOM. This approach increases development efficiency and helps to ensure that the UI remains consistent with the application's underlying data. React also supports server-side rendering, making it ideal for building large-scale, high-performance web applications. React has become a cornerstone of modern web development and is widely used in industry and academia.

2.3.2 Vue.js

Vue.js [58], commonly referred to as Vue, is an open-source JavaScript framework used for building user interfaces and single-page applications. Created by Evan You in **2014**, Vue emphasizes simplicity and ease of use, allowing developers to create highly responsive and dynamic applications with less code than other frameworks. Vue uses a reactive data binding system, meaning changes to the application's data are automatically reflected in the user interface. This makes it easier for developers to manage application state and create complex, interactive interfaces. Vue also supports server-side rendering and has a large ecosystem of plugins and extensions, making it a versatile choice for web development. Vue has gained popularity in recent years and has become a popular choice for developers looking for a lightweight and flexible framework for building web applications.

2.3.3 Angular

Angular [1], commonly referred to as Angular 2+, is a popular open-source framework for building web applications. Developed by Google in **2010**, Angular uses a component-based architecture, allowing developers to create modular and reusable components that make up the application's user interface. Angular also supports two-way data binding, meaning that changes to the UI are automatically reflected in the application's data and vice versa. This feature simplifies the process of managing the application's state and makes it easier for developers to create complex, dynamic interfaces. Additionally, Angular includes features such as dependency injection and testing tools, which make it a comprehensive and powerful framework for building web applications. Despite its learning curve, Angular has gained significant popularity among developers due to its robust feature set and community support.

2.3.4 Svelte

Svelte [48] is a relatively new open-source JavaScript framework that has gained attention for its innovative approach to building web applications. Developed by Rich Harris in **2016**, Svelte focuses on compiling application code during the build process rather than relying on client-side JavaScript to run at runtime. This approach results in smaller and faster applications with reduced runtime overhead. Svelte also uses a component-based architecture similar to the previously mentioned frameworks, allowing developers to create reusable UI components. Additionally, Svelte features a reactive data binding system that automatically updates the application's UI when its underlying data changes. The framework's simplicity and efficiency have made it popular among developers prioritizing performance and ease of use. While Svelte's adoption is still growing, it has shown promise as a lightweight and efficient alternative to more established frameworks.

2.3.5 Solid

Solid.js [43] is a relatively new open-source JavaScript library for building web applications. Developed by Ryan Carniato in **2019**, Solid.js uses a reactive programming model to create highly efficient and performant web applications. Like React, Solid.js relies on a declarative approach, where developers specify what the UI should look like based on the application's state rather than directly manipulating the DOM. However, Solid.js differs from React because it uses a technique called reactive programming to update the UI. This means that changes to the application's state are automatically reflected in the UI, resulting in faster and more efficient updates. Solid.js also features a simple and 'easy-to-learn" API, making it accessible to developers of all levels of expertise. While Solid.js is still a new library, it has gained attention for its performance and developer experience and is poised to become a popular choice for building web applications.

2.4 Front-end Libraries

Front-end libraries are essential to modern web development. These tools provide developers with reusable and efficient tooling to build sophisticated user interfaces and enhance user experiences. Unlike full-fledged front-end frameworks, which often force the developer to follow conventions and rules, libraries focus on functionality allowing the developer to construct a web application build using a collection of different libraries, providing a cohesive and feature-complete web application.

We will focus on a common set of libraries often used with modern front-end frameworks. These libraries include libraries such as routing libraries, state management libraries, styling libraries, validation libraries, and utility libraries.

2.4.1 Routing Libraries

Routing libraries are used with single-page applications (SPAs) to handle navigation. They provide the front-end framework with the ability to render different components or pages based on the URL. These types of libraries are useful when a web application consists of multiple pages with links to other web pages. Another benefit over the default anchor tag is that these routing libraries can navigate to other pages without full page reloads. This enables a seamless user experience.

React-router is a library that handles routing for React applications. Because the React library does not support routing out of the box, we can use libraries like react-router to implement navigation functionality in our React application. Other front-end frameworks, such as Vue, also have their own router libraries, such as vue-router, which provide similar functionality for your Vue application.

2.4.2 State Management Libraries

State management libraries help you with managing the state of your web application. This ensures that different components in your application have access to and can efficiently update the same shared data. When your application grows, managing the state correctly can become crucial to avoid data inconsistencies and improve the maintainability of your application. However, there are a lot of different types of state management that we can use in modern web applications.

The first type of state is local state. Local state is the state that is local to a given component. An example could be a checkbox component. The state to determine if the checkbox is toggled on or off is local state. This state is isolated from the rest of the application so that every checkbox rendered on the page can manage its own state, on or off.

A second type of state is global state. Multiple different components consume this type of state. An example could be the state that decides if dark mode on a website is enabled. This state can be consumed by every component that supports dark mode. To allow the component to render either light or dark themes. However, we may also need to update the state from dark to light mode so that the toggle component can update the global state.

However, not all state needs to be completely global state; it could be possible that only a subsection of the web application needs to share state between its components. An example could be the authentication state only used on the application dashboard. We could then provide the authentication state only to the dashboard and child components. This is called semi-global state.

Server state is the state that contains information only available on the server. This type of state is often the response data from a request to the server. This type of state can also be one of the previously defined types of state. It could be local state or semi-global state. State that is not based on server information is called client or client-side state.

2.5 Styling Libraries

Web development has become increasingly complex and feature-rich in recent years, with modern web applications requiring robust styling to create visually appealing and intuitive user interfaces.

This section will discuss such styling libraries, also known as CSS frameworks. The most common CSS frameworks are frameworks that provide pre-designed user interface components. Components include buttons, forms, typography, and other common design patterns of modern web applications. The main focus of these libraries is to simplify the process of building responsive and modern web applications that can be built quickly without too much design knowledge.

A subcategory here is a framework already built for, for example, React, with a set of pre-designed components. This takes the styling even one more step away from customizability because we can not use these components in a different framework.

In addition to these pre-designed user interface libraries, we will look at other modern types of CSS frameworks that try to solve the styling problem using different methods. One common recent pattern was the CSS-in-js pattern, which allowed us to write CSS in JavaScript. We will examine why they were created and what problems they would solve.

A newer type of CSS framework is a CSS utility-first CSS framework. These frameworks consist of a set of atomic utility classes that can be combined to create a custom style quickly. This allows us to write less code, which results in faster development and automatically provides somewhat consistency throughout your design.

2.5.1 Preprocessors

CSS preprocessors are tools that enhance the functionality of CSS by adding features such as variables, mixins, nesting, and functions. These preprocessors are used to make CSS code more maintainable, scalable, and reusable. Popular CSS preprocessors are Sass and Less [45].

Sass [42], or Syntactically Awesome Style Sheets, is a CSS preprocessor that extends the functionality of CSS by introducing variables, nested rules, mixins, functions, and more. Sass is a superset of CSS, which means that any valid CSS is also valid Sass code. Sass is thus a way to keep your large set style sheets organized by extending upon the basic functionality of CSS. Variables can be used to avoid repetitions and provide consistency. They can be used to store color values for easy theming or to store complex math you want to use in multiple places. Nesting, on the other hand, reduces the need for repeating the same CSS selector repeatedly. You can write a general selector and nest other selectors to extend the general selector. Using nesting, we can now organize and group our code into coherent sections, which can also improve the general readability of our code. When writing complex styling, having access to control flow operations can be useful. We can then use if/else statements or even for loop statements to generate or leave out certain styling based on provided variables. And because Sass is compiled into CSS, we can write complex and organized Sass code and then transpile it into CSS that the browser can understand. Sass also provides an alternative syntax based on indentation, allowing you to replace the curly brackets with indentation.

Less [18], or Leaner Style Sheets, is a backward-compatible language extension for CSS. This means we can write CSS and extend it with additional features that Less supports. Like Sass, Less also includes features such as variables, mixins, nesting, operations, functions, and more. Sass originally inspired Less to provide Less with a leaner feature set and syntax more closely matching CSS. However, Sass later optimized its syntax to look more like Less and thus also closer to CSS itself.

2.5.2 Postprocessors

While preprocessors are used to convert custom syntax into CSS, postprocessors are used to transform CSS into a more compatible version of CSS.

PostCSS [37], is a tool for transforming CSS. Examples of transforming CSS include 'adding vendor prefixes', 'polyfill modern CSS features unsupported browsers', 'CSS modules', 'prevent errors by using stylelint', and more.

2.5.3 CSS frameworks

Bootstrap [5] is a CSS framework that provides a set of prebuilt components in the form of CSS and JS libraries. On the other hand, it also provides a way to customize these prebuild components and mechanisms. Because Bootstrap is built on top of Sass, we can easily extend or overwrite it by adding the correct styling files. We can also customize colors and CSS variables that are defined within the framework so that we can globally change certain values to our liking to customize the default stylings. They also allow you to import only the styling you need for your project and the necessary JavaScript. This can have a positive impact on performance and bundle size. Other than customization, Bootstrap adds support for Layouts, Content, Forms, Components, Helpers, and Utilities. Bootstrap uses CSS classes to expose these functionalities. We can simply add CSS classes to our HTML components and use the features provided by Bootstrap. However, some components, like the dropdown element, need some JavaScript to allow interaction with the element. In this case, we must also import the correct JavaScript to wire the component correctly.

Mantine [30], is a React CSS framework. In contrast to a framework-independent CSS framework, this framework can only be used with React. Mantine has features like theming, hooks, forms, core which includes layouts and components, and a set of dates components. Theming allows users to select the right colors, font, and properties for their specific use case. However, it is not easy to fully customize this framework. It is useful that we have some level of customization. Another interesting part of this CSS library is its hooks. These include basic JavaScript functionality that we can reuse throughout our own components. They provide hooks for state management, UI, Utilities, and more. An interesting hook is the 'use-click-outside' hook, which you can call a specific action when the user clicks outside your component. This can be useful for components like dropdown menus, modals, and other types of popups you want to close by clicking outside the component. Mantine also includes useful React components for managing layouts. You can force aspect ratios on components, center elements, space components, and more. These components can provide you with basic styling that you would otherwise need to know by hand.

Other frameworks in this category include **Bulma** [6], **MUI** [32], **AntDesign** [2], and more.

2.5.4 CSS In JS

Styled Components [47] uses JavaScript template literals (formatted strings) and the power of CSS to allow you to write CSS for a given component. Because we write CSS that is linked to that given component. The component construct now has the styling already backed in. This means that where ever you use the Styled Component, the styles you provided when creating the component are used when rendering the component. We can create dynamically styled components because Styled Components use JavaScript template literals. We can add a function that takes props and returns custom styling into our template-literal to make the styling dynamic.

CSS Modules [9] are CSS files where the class and animation names are scoped locally by default. They thus will have no impact if they are used globally. We can only use CSS modules by specifically importing the class names and using them from the import directly. This way, we can reuse the same class names as long as they are in a different CSS module. This ensures that class names will not clash with other class names in other CSS files. CSS modules also allow you to compose a class name based on other class names. We can also import these other class names from different files as dependencies.

Styled JSX [57], is a library that allows developers to use the style tag within JSX code. This has the benefit that we can scope the style tag to the scope where we need to use the provided styles. This can be useful if we want to inline our styles with the ability to isolate our styling from the rest of the application. Styled JSX also adds some basic functionality to vendor prefix CSS and optimizes performance.

Emotion [15], is a library for easily writing styling with JavaScript. They provide a function, 'CSS', that allows you to write and compose styling. Emotion also supports features such as styled components, composition, nested selectors, media queries, and more. Nested selectors are important; for example, when the user wants to add specific styling on hover, we can add styling to '&:hover'.

Other frameworks in this category include **JSS** [27], **stitches** [46], **CVA** [10] and more.

2.5.5 CSS Utility frameworks

TailwindCSS [50], is a CSS utility framework. In the case of tailwind CSS, this means that tailwind provides a lot of shorthand CSS classes that we can use to compose the style we want for our component out of these building blocks. TailwindCSS is built to provide a styling system to you with various options to choose from. This limits the use of arbitrary values in your code but, on the other hand, provides enough options to create something unique to your liking. TailwindCSS provides consistent colors, spacing, typography, shadows, and more. However, TailwindCSS is still low-level enough to create a completely different design easily. Another benefit of tailwind is that you will never ship unused CSS. This is because only the styles found in the project are shipped with the final CSS bundle when building for production. Besides atomic utility classes, tailwind supports features like responsive design, hover states, focus states, dark mode states, customizability, and more. Tailwind also provides modern and state of the are shortcut utilities that use vendor prefixes or combine multiple CSS rules into a singular CSS utility. And if we want even more control, we can add our custom utilities or even compose multiple utilities into other utilities or general class names.

Other libraries include **MasterCSS** [31], **Tachyons** [49], **Open Props** [35] and more.

2.6 Back-end Frameworks

In this section, we will discuss back-end frameworks. Similar to a front-end framework, a back-end framework is a set of pre-defined code that can be used to build back-end applications efficiently. Back-end frameworks usually allow the developer to build an API, or more specifically, a REST API. The client application can use this API to communicate with the server and request all the information that the client needs. Common tasks the back-end server performs are authentication, authorization, and a form of persistent storage. This section will discuss two back-end frameworks in the JavaScript ecosystem.

2.6.1 Express

Express.js, or Express [16], is a lightweight and flexible web application framework created to run on Node.js. Express is a software library that provides code to handle HTTP requests and responses in a simplified manner. Express follows a middleware-based [54] architecture at its core. This means that we can chain together multiple middleware functions to handle incoming HTTP requests to perform tasks such as parsing incoming data, authentication, managing sessions, handling errors, and more.

However, Express does not impose a strict pattern or conventions, providing developers considerable freedom and flexibility. Developers can extend their back-end application by installing third-party packages to enhance the functionality provided by Express itself. This allows for rapid development and simplified integration of additional features not directly supported by Express.

2.6.2 Nest

NestJs or Nest [33], is a back-end framework created to build efficient and scalable server-side applications and APIs. It utilizes TypeScript by default to enhance code quality and maintainability. In contrast to Express, Nest provides a more structured and organized development environment with more feature-rich documentation to help the developer integrate with other back-end libraries, such as popular databases or authentication libraries.

Because Nest is built on top of Express, it also embraces the concept of middleware. However, to provide the developer with more structure and consistency, Nest implements concepts such as Dependency Injection, which allows modular and interchangeable development of different modules in your back-end code. Decorators, which allow the developer to annotate classes and methods to provide them with useful information. And more. Nest also encourages different object-oriented principles to create scalable and testable applications. In conclusion, Nest is a feature-rich back-end framework for building enterprise-level web applications.

2.7 Back-end Libraries

In this section, we will discuss common back-end libraries used when developing applications for the web. The most common types of libraries are database libraries and authentication libraries. These play a crucial role in modern web development. This is because security and large data are important topics of modern development. These back-end libraries are relevant because they provide the essential building blocks of pre-built and reusable components to handle essential tasks for your back-end.

2.7.1 Database Libraries

Databases are an essential part of modern web applications. They provide a method of storing data so that this data can persist over different sessions with our web application. However, when integrating databases with our web applications, developers often want a scalable and maintainable method to interact with their database.

A technique often used in modern web development is using an Object Relational Mapper (ORM). This allows a developer to access the database as if a database API was already created for the developer to use. This allows the developer to simply call the correct functions on the database object accessible from your back-end code.

A popular ORM is Prisma [39]. Prisma allows the developer to define a schema in their custom schema language. This schema will then be interpreted to generate the correct SQL or database syntax. Another benefit of using Prisma is that by creating the Prisma schema file, we can generate TypeScript definitions. This allows the developer to interact with the database in a type-safe manner.

A recent alternative to Prisma is Drizzle [13]. The Drizzle schema is defined in TypeScript itself. This is useful because we do not need to generate the TypeScript every time we update the database models.

2.7.2 Authentication Libraries

In modern web applications, ensuring the security of user data has become more critical than ever. One of the fundamental elements of security is authentication, the process of verifying the user's identity before granting access to sensitive information. As these systems become more complex, developers face the challenge of implementing secure solutions that adhere to security best practices.

To address this challenge, authentication libraries have emerged. These libraries provide the developer with standardized methods of implementing a secure authentication process. The most well-known library to handle authentication is Passport.js [36]. Passport.js is an authentication middleware library. This means that Passport.js intercepts the incoming authentication request and uses the correct developer-defined authentication strategy to authenticate the user. The developer can define authentication strategies such as username/password, social logins, and more. This makes Passport.js a flexible and modular framework for handling all the different types of authentication.

A more recent trend is battery-included authentication libraries. An experimental library that does this is Auth.js [3]. Auth.js supports over 60 popular social logins like Google, Facebook, and more. As well as standard authentication methods, such as email, password-less, magic link, and more. Another benefit of this library type is that it supports adaptors for modern frameworks. This means that even more boilerplate code can be omitted. Besides the library being battery-included, Auth.js provides flexibility so developers can use their own database. Auth.js strives to be secure by default by supporting features like signed prefixed server-only cookies, built-in Cross-Site Request Forgery (CSRF) protection, and advanced JWT.

2.8 Client Server Code Sharing

An important limitation in modern web development is sharing code between the client and the server. This section will discuss some common techniques that help developers enquire end-to-end type safety. Because we can both use TypeScript on the client and the Server, it would be helpful if we could use the benefits of the TypeScript ecosystem. This is not always straightforward because client and server JavaScript provide different built-in libraries and features, so server TypeScript code cannot run in the browser. This section will look at what is possible when sharing client and server code.

2.8.1 Monorepo

A common problem before monolithic repositories was that maintaining and integrating the code written in different repositories was a lot of work. Whenever we would change the back-end code, we needed to add these changes to git, create a commit and push the code before integrating it with our client application. The client then needed to pull the server source code, and using this useful server information in the client application was still difficult. We could create another repository that contains only the shared code between the client and the server, but every change needed to be in sync with both the client and the server, and managing this could end up really affecting the DX when sharing code between these different repositories.

A solution to this problem is the monorepo. A monorepo is a single repository that contains the different packages, building up the full application. This has the benefit of keeping all the separate packages in sync with each other. We could also use a shared TypeScript file to benefit from type safety as if the different packages were one coherent package. This would mean that changes in the shared code would warn the developer in the other client and server packages if something has changed.

A modern monorepo is Turborepo [52]. Turborepo re-imagines existing build system techniques to remove maintenance burden and overhead. Turborepo is built on features such as incremental builds, content-aware hashing, parallel execution, advanced caching mechanisms, including remote caching, zero runtime overhead, and more. These features allow for fast building and rebuilding of your application, especially when a part of the source code has already been built in the past or by somebody else using remote caching.

2.8.2 tRPC

tRPC [51], not to confuse with gRPC [20], is a TypeScript library that allows developers to build end-to-end typesafe APIs. Because the implementation of the API is defined in TypeScript, and

TypeScript supports advanced type inference systems. tRPC can use the defined API endpoint to infer a correct type for its tRPC client. Because of this, no code generation needs to be done. And because the client and server are fully connected in TypeScript, tRPC does not need any build or compile steps to benefit from type safety. So when the developer changes the server source code, TypeScript will give type errors wherever the endpoint is used in the tRPC client code.

However, a benefit of code generation is that you can generate an intermediate format that allows the developer to use that to generate, for example, a Postman JSON file. This could be useful if you want to test tRPC outside your application.

Another problem with tRPC is that we cannot fully integrate it with other back-end frameworks. We could define a single route in our back-end application that handles all tRPC requests, but this does not always allow seamless integration with all the features more advanced frameworks provide. This can be a disadvantage when building complex, scalable apps that are built with fully-fledged back-end frameworks.

2.8.3 Open API

Open API [22] is a specification that allows different software applications to communicate and interact with each other seamlessly. It provides a set of rules and guidelines that define how developers should design and document their APIs. By adhering to these standards, developers ensure that their APIs are easily accessible to other developers. This not only improves the accessibility of your external APIs but also has advantages when using the specification internally.

Open API offers a developer experience that is incredibly empowering and efficient. By using the Open API specification as the single source of truth, developers gain access to well-defined documentation for their APIs that could be implemented correctly across their entire application. It provides a readable format for the available endpoints, request parameters, response structures, and other requirements.

Another benefit of this standardized format is that developers can create code generators that result in the Open API specification. However, there must be an implementation for the given framework you are using, and even then, we need to write additional code to fully allow the code generator to generate a complete Open API specification.

An interesting client library, however, is feTS [23]. This library allows developers to create a fully typesafe client API based on the Open API spec without code generation. This is possible because we use the JSON format for Open API and define it as a TypeScript object with the "as const" keyword. This allows TypeScript to infer the full type of the Open API specification and use that type to provide type safety over the JavaScript fetch API.

2.8.4 TypeScript Compiler API

The TypeScript compiler API [56] is a powerful and versatile tool that extends the capabilities of TypeScript. The API is directly accessible from TypeScript and provides developers programmatic access to the TypeScript compiler. This allows developers to interact with TypeScript source code in a structured and efficient manner.

Through using the TypeScript compiler API, developers can programmatically analyze, transform, or even generate new TypeScript code. The API can be used for tasks such as refactoring code, static analysis, custom type checking, and code generation. The part we are most interested in is code introspection. This means that we use the parsed TypeScript code to interpret the meaning of a TypeScript program and use this information to generate new TypeScript code or types. The API can be used to generate types where we cannot share code between the client and the server because of runtime conflicts.

2.8.5 Docker

Docker [11] is a popular and powerful platform that simplifies the process of creating, deploying, and running applications in isolated containers. These containers are lightweight, portable, and, most importantly, consistent across different environments. This means we can easily distribute our code to different developer machines and deployment systems regardless of the underlying infrastructure. Docker has revolutionized software development and deployment by providing a standardized way to manage, build, and test applications. Docker enables seamless collaboration and continuous integration for your development environment.

Chapter 3

Tooling

In this chapter, we will explore the lifecycle that unfolds during the development of web applications. We will dive into various tools that align with specific elements within this development cycle. Each tool is designed to tackle the challenges associated with a particular phase of the development process. Our discussion will begin by outlining the general web application development lifecycle, followed by a section discussing a set of diverse tools capable of enhancing the developer experience for these processes within this lifecycle.

3.1 Lifecycle

When building applications for the web, many of us lay out a preliminary plan before we start setting up and building our web application. This is because we need to define what technologies we will use before we can initialize our project with these technologies. This can be done by manually selecting a set of technologies or using a predefined set of technologies. In web development, this is called a technology stack. However, a common problem with these stacks is that they only focus on the software frameworks and libraries used within your project, not the tooling used to help you provide a better experience building applications. For example, we could decide to use the `MongoDB`, `Express`, `React`, and `Node.js` (`MERN`) stack. This means that we will develop our application with `MongoDB`, `Express`, `React`, and `Node.js`. As we can see, this stack does not define anything about the tooling you should use. For example, we could use the `MERN` stack with or without `Docker`. However, this results in a completely different DX. We also must not use a predefined stack; we could build one ourselves.

After settling on a stack, we can use a tool, such as `create-mern-app`, or install all the technologies independently. This can sometimes be difficult because of the complicated configurations of bundlers, compilers, etc. So in this phase, it is important to install the technologies correctly, update the configuration so that the technologies work together, and optionally install other technologies or tooling to complete the initial setup of your development environment. Another problem with some of the tooling for initializing your project is that they come with a lot of bloatware that you sometimes do not need at all. However, it can be helpful for beginners sometimes. It can also be annoying when you always have to repeat the process of cleaning up your default project configuration.

Even after establishing a foundational starting point for our web application, we sometimes still need different libraries. That helps us with styling, databases, etc. This phase of the development lifecycle can be done right after initializing the project and during development when new technologies are needed. This is because we don't always know everything we need when developing our application. This phase can become more difficult because when the codebase grows, we need to ensure all these technologies are still working and, most of all, working together. It is thus important to configure these projects correctly and ensure we don't interfere with other libraries.

We may start developing our web application once the libraries are installed and properly configured. But when we look deeper at this process, we see that we, as developers, still repeat much of the work we are already familiar with or know how to do. This can possibly have an impact on our productivity and our developer experience. A possible solution here would be to use a library, but even some libraries make you repeat at least some of the work. We refer to the code that represents something we already know how to do but still repeat throughout our code a boilerplate code. Because this code is so predictable, we can look at different techniques to generate this code for us. We will look at this in later sections.

Another similar problem is that we sometimes must repeat the same code twice. The difference is that the concept is not similar to the boilerplate code. But that the code is duplicated. We can not always do something about this without the right tools. The best example of this is the sharing of type-safe REST APIs. Here we have already built the API by implementing the server code. However, we need to access the server using our client code; most of the time, these are different code bases. We can not access our server code from inside our client, at least in most cases. This is because a client does not have access to, for example, the file system and the code can thus not compile for the client. So without changes to the compiler or dependencies, we can not import the server-side code into our client application. There are simple solutions to some of these problems, like creating a shared client and server accessible type safe construct that both packages need to implement. However, the developer experience here is still not always ideal. An example of why not is because we can have, for example, a type-safe database ORM that provides types we do not want to duplicate in the shared library. And we can't always pick the libraries we use without affecting the overall developer experience.

As the code becomes increasingly complex, we sometimes want to get an overview of the codebase or quickly test out different features that are, for example, only implemented on the server yet. We can do this by manually looking at the server implementation and creating, for example, a Postman API request for that given server request. However, it is difficult to maintain if you want to ensure every API request you create in Postman still works as intended. Besides, it can sometimes be useful to see a visual overview of the code you have already written or have a visual source of truth you can access for your current database. This can also be helpful when you have not worked on a project for a long time or for new developers to understand better what is happening.

3.2 Lifecycle Tools

3.2.1 Initialization Tools

The first tool we will describe is the initialization tool. This tool helps you to initialize your project. We divide the initialization tools into three categories. A tool for initializing the general project structure. A tool for initializing the different frameworks we want to use in our project. Lastly, a tool that helps you initialize a library for one of the frameworks you are using.

The **project initialization tool** can, in some cases, even be left out. This is because the tool only initializes things, like the *packages* directory and a *.gitignore* file. We can, however, expand the functionality to create different types of mono-repositories. But in the most basic version, this tool sets up the project structure so that the individual frameworks for the project can be installed consistently and structured.

The **framework initialization tool** is more complicated than the project initialization tool. This is because this tool does not only download the correct templates for a given framework. The tool also ensures you have a minimal and barebones version of the framework template. The tool's last thing is ensuring the framework is configured correctly. An example of this can be to configure the client or server ports. These types of configurations provide consistency between different frameworks so that we end up with a consistent developer experience. This means that if tools like to use different package.json scripts or use different ports for hosting their client app, consistent scripts and ports will be automatically configured so the developer can have a consistent experience across projects with different frameworks and libraries.

The **library initialization tool** is even a step more complicated. This is because one library can be used with a set of different frameworks. However, different instructions might be needed for each framework to install and configure the library or tool correctly. A good example of how instructions can differ per framework is the `Tailwind` getting started frameworks guide [17]. We can see here that `Tailwind` provides the user with different installation guides depending on what frameworks the developer is using. However, if we can detect the framework we use, we can also derive which installation guide to follow. We can thus see that this process of following instructions can be automated for all the supported frameworks. The library initialization tool does exactly that. It first detects the framework currently being used - by looking at the directory where the command is called from. Once we know what framework is currently being used, we can request the set of instructions for the library we want to install, given that framework. And lastly, we can execute these actions to install the library and configure it correctly with the framework.

An important thing to notice is that we not only generate template code for initializing the library, but we also insert code into existing files if needed for the library to work. It is important that this is done correctly because we don't want to affect the code that the developer has already written. However, this is not always easy, especially when we are initializing multiple libraries that affect the same file or that affect files that the developer heavily edits. For more information on safe ways to implement this, take a look at the implementation section.

3.2.2 Generate Tools

The next set of tooling is generate tooling. These tools allow developers to generate code for a common problem repeated multiple times during development. Sometimes it just is one simple boilerplate code. We also need a boilerplate file and some code insertions into other files. In some cases, we need to generate multiple files of boilerplate code in combination with the correct insertions into other files. An example of an insertion is an added import statement in a given file, or even adding the imported class into a dependency array, or adding configuration to an existing configuration file.

Some commonly occurring actions are creating a new page with optional parameters in a client web framework. Another action can be creating a boilerplate component for a given client framework. More interestingly, we can create the boilerplate code for a resource in a server framework. We can view a resource as a controller and service with implemented Create Read Update Deleted (CRUD) operations.

As with library initialization tools, the generate tools consider the frameworks used to decide how boilerplate code should be generated. This is because a component would look different depending on whether you are using `React` or `Vue`. To provide this functionally, we must be able to detect the framework we are using. However, sometimes this is not enough.

Take, for example, a `React` application. Say we want to create a new page `/item/:id`. We could do this manually, but we may like to automate this process to save time. But a problem occurs when we run the command. This is because, by default, `React` does not support routing. We need to install a third-party library to solve this issue. This means that we should generate boilerplate code independent of the framework we are using and based on the libraries that help provide this functionality. To allow for this functionality, it would be helpful to run the instructions provided by the framework you are using, followed by the instructions provided by the libraries that help solve these problems. This way, a library can build further on the framework's principles and expand the code generated by inserting its own boilerplate after the default framework code is generated.

This becomes more clear when we look at resource generation. Let's say we are using a framework that supports the generation of resources. Note that we don't have any libraries installed at this moment, only the framework itself. When we run the code to generate the boilerplate code, a service and controller file are generated. However, because the tool cannot know what database or storage we are using, we can not generate the implementation boilerplate code for the CRUD methods in the service. They are left empty, so the user only needs to implement them before being able to use the server API. However, if we have initialized a database library, the code generator for that library will extend the code previously generated by implementing these service methods. Should we later add a library for authentication that could then maybe add guards to the controller file of the framework to block unwanted requests.

When the number of libraries grows, we should ensure that the generated code works and does not generate conflicts. This could be a limitation if not implemented correctly.

3.2.3 Introspection Tools

Another interesting set of tools is introspection tools. These tools look at the code you have already written and use that as the source of truth to generate more boilerplate code. This can be useful when the framework does not support the action you want to perform with the code you have already written.

The best example is generating a client adaptor for your backend API. Let's say you have written your backend in TypeScript, and you want your client to be able to access this API in a typesafe manner. This is difficult if you don't want to duplicate your code or the types you created in your backend. You can try to share the types with the client and the server using a monorepo, but this is not always convenient. This is because, even though the types are shared, you may still need to generate the correct files to use the shared packages in both your client and server. This can be annoying if you simply want to change and test one type because compiling the types and generating the declaration files takes time. This can impact the developer experience. Another problem that can occur is that you want to use a database type in your shared type definitions; however, the client does not support the database library, which can further impact the developer experience.

Lately, there has been work to solve this problem. Our solution is to introspect the server code and generate the client adaptor. This has some interesting benefits over other methods.

- An interesting benefit is type inference; because we introspect the code, we can infer and extract the typescript types from the code written by the developer and the libraries the developer uses. This means that we don't include the server libraries themselves but just the extracted types, which resolves the problem of not being able to import server code into the client.
- Another benefit is that you can use any of the supported frameworks, even if they themselves don't support a typesafe way to share your API endpoints. We do not need to configure anything. We can generate the client definitions from the source code you have already written.
- The code you generate or the output format is customizable. You can customize the code to support code generation compatible with your goals or generate adaptors for anything you want. You could, for example, also generate a Postman-compatible JSON that you can simply import instead of manually managing these API definitions in your server.

Other introspect tools are tools that use existing code or configuration. To perform certain actions on them or generate other code that you may need. Another useful example is that when using a special database schema language, you can automatically generate the correct types based on your database models. The generate tools could then use these to ensure the correct types are used in your service when you access your database. This is also interesting for generate tools because we can simply define our database model and generate the resource. This will result in boilerplate code that already contains the correct types that were generated from the database model. This will result in fewer errors and type safety from the beginning of the process.

3.2.4 Interaction Tools

The last set of tools are interaction tools. Like introspect tools, interaction tools introspect the source code to provide insights into the codebase or even interact with the codebase using a visual user interface.

A common interaction tool would be an alternative interface to Postman. However, we see these types of tools mostly with Graphql libraries. We mostly don't see them with most REST API libraries. However, similar tooling could also be useful with REST APIs. This is why interaction tools are important. They provide a simple and quick way to interact with the REST API you created without implementing a client application.

Other interaction tools are similar but provide an interface for other parts of your application. Good examples are database and storage managers. The information needed can be introspected from the source code itself. For example, we can detect that we are using a certain database library. We can then use the correct instructions to retrieve the database URL. We can then use that database URL in the interaction tool to set up a connection with the database and provide an interface with basic functionality such as viewing tables, viewing rows, basic sorting or searching, etc.

Chapter 4

Implementation

This section will discuss how we implemented the toolset discussed in Tooling. We will also discuss the limitations and benefits of the tools themselves. We will then look deeper into the exact implementation and also take a look at how we can extend the `flaze` toolset.

4.1 Flaze Toolset

We created a Command Line Interface (CLI) called `flaze` to implement the toolset described in Tooling. Flaze is created with `Node.js` and `TypeScript`.

We can quickly test the application by running `npm run dev` or `flaze update` followed by the `flaze` command we want to access. The dev command uses `ts-node` to build the application using `Node.js` and `TypeScript`. We can also globally install `flaze` using the build command followed by a global `npm install`. We are using `tsc` for the build, and we can install npm packages globally using `npm install -g .`. To make sure we get a clean install every time, we remove the transpiled `lib` folder created by `tsc` and remove the previous global install of `flaze`. The `flaze update` simply then implements `npm run install:bin` to work from anywhere the CLI is called. This helps us develop our application because we do not need to check if we are in the correct directory to install the latest version of `flaze`.

4.2 Commands

4.2.1 Help Command

Because `flaze` is a toolset that helps you improve the developer experience when building web applications, it is very important to create a command line interface (CLI) that is easy to understand and easy to use, even when the user does not know the commands he/she can use. This is why the help command is so important. This helps us learn and understand the toolset without ever leaving the CLI. However, a basic understanding of CLI tools is required.

When we run `flaze` or `flaze help`, we get all the information to get started with the toolset. This command shows the different options like `--version` and `--help`, but also gives us a list of the available commands. Each command also has an extra description to help you understand what the command does. All this information gives clear information on using the `flaze` toolset and how to run commands.

If we still don't understand a given command or want more information on what the command does, we even go deeper and ask `flaze <command> --help` or `flaze help <command>`. This will show you the arguments and options the command expects and show you additional examples of how the command could be used. This could be useful to get an understanding of what the arguments and options could be.

4.2.2 Init Command

The first initialization tool in the toolset is `flaze init`. This command will simply create the initial structure for your project. Because all the following commands will scaffold their packages in the packages folder, the `flaze init` creates that folder. Another thing this `flaze init` command does is create a `.gitignore` file with some default ignored directories. The directories included are `node_modules` and common IDE/editor configurations.

By ensuring the project structure, the commands we will use later will be able to easily infer where the packages are located and where the project's root is.

4.2.3 Create Command

The second initialization tool in the toolset is `flaze create`. This tool will initialize a selected framework into the correct `packages` folder. Although this seems simple, we must ensure that the framework we install is as minimal as possible and configured correctly.

When we call `flaze create <package-name> <framework-name>` we perform a couple of steps. We could use existing tools like `create-vite` or `create-next-app`, or we could define our own templates and pull them from their own source.

A common problem with tools like `create-vite` is that they come with some amount of bloatware. This can mean they come with boilerplate code in the `src` directory, sometimes even assets you will never use. This can also mean that they come (optionally) preinstalled with tooling like `es-lint` or even styling libraries like Tailwind.

Another problem is inconsistencies between these different frameworks. This means that when we create a `react` app using `create-react-app`, we get port 3000 as the default port, but when we create the same `React` app with another tool like `create-vite`, we get port 5173. Another problem could be that when we need to change the port quickly. This could be dependent on which framework we are using. We could also generalize this so that, for example, the port is always defined in a local `.env` file. This could further standardize the differences between different frameworks.

Limitations

One of the limitations of this approach is that these `create-app` tooling and the **frameworks themselves are constantly changing**. This also means that we should update our tooling whenever the framework or the basic structure of the framework changes.

We could use custom templates to minimize the impact because, most of the time, we could simply update the dependencies, and the template still works. However, this is not always possible, so in those cases, we still need to update our template to support the newer versions of the framework or tooling.

Another problem could be that a **user wants to use a specific version of a framework** or tooling. This could mean we need to manage versions in the `flaze` toolset, which is currently not supported.

Benefits

By having a **clean and minimal install**, we have the benefit of simplicity and the option to **install only the libraries or tooling that we need** ourselves. This means that we do not depend on what the `create-app` supports and can extend this minimal version of the framework however we want. However, to simplify the initialization of libraries and tooling, we will provide another tool in the `flaze` toolset.

4.2.4 Use Command

The third initialization tool in the toolset is `flaze use`. This tool allows users to initialize libraries or tools for a given framework. As discussed above, this is useful because we can extend the `flaze create` command with different libraries without executing all the independent instructions yourself.

When we call `flaze use <library-name>` we perform a couple of steps. We first detect all the packages we currently have installed using `flaze create` or those with similar structures to those installed using `flaze`. When we run the command in the root directory, all the packages for the project are detected. However, only that package is detected if we run the command at the project root.

Once we have detected the framework(s), we can check if the library can be installed for the given framework. If not, we provide a clear error to the user. For example, we can not install the `Prisma` database Object Relational Mapper (ORM) in a client-only framework.

The last thing the `flaze use` command does is perform the correct instructions to install and configure the library for the framework. It is important to note that this not only performs the `npm install` command but also updates the existing codebase to integrate the library correctly. For example, when using `react-router`, we install `react-router` and initialize the router config in the `main.tsx` so that the user simply needs to add a route to the router array and get up and running with `react-router`.

Limitations

The first limitation is that the user may have changed the codebase so much that we could not inject the correct code at the correct location. This could result in incorrect library behavior or even result in a broken application that could not run anymore. This could also mean important user-written code is overwritten and lost in the worst case. Luckily, we could provide a way to cache the current project before we perform an action so that the user could revert the install if needed and then manually configure the selected library.

An almost similar limitation to the abovementioned limitation is that libraries could conflict. What if we initialize two routers? What if we initialize two database libraries? These can, in some cases, conflict with each other. This could be a problem when we have complex applications with different complex configurations for multiple similar libraries or tooling. We could maybe solve this by parameterizing the use command to, more specifically, provide instructions on how and where to install the library. However, this can become hard to implement.

Another limitation is that there is not always one way to install a certain library. We could use a router as a memory router for a tabs component on our web page, but we could use that same library to handle the browser routing. These are very different use cases. However, how do we decide what the use case is? We could again look at a way of parameterizing the command so that a user can provide clearer instructions. However, this may not result in a bad developer experience. Otherwise, this can result in the user not using the command. A possible way to solve this is by providing prompts to the user. This way, the user does not need to remember what parameters there are to configure a library but can simply answer the questions provided by the interface. However, this could hinder the benefit of quickly installing a library. We could solve this by only showing the prompts under a universal flag like `--prompts` or `--customize`.

Another related problem might be that the user does not know beforehand what a command will do. However, we could solve this by providing feedforward information using the command of the `flaze feedforward`. This could explain what will happen when a user is just learning to use the `flaze` toolset.

Benefits

The benefits are that you can quickly configure your framework with the libraries you want to use for your project. This makes the `flaze` toolset modular, which is a benefit of the standard `create-app` tools, where you need to know all your project requirements before you start building it. So when you, later on, need a certain library, tool, or configuration, you need to configure that yourself.

Another benefit is that you do not need to know how to configure a certain library with a different framework because `flaze use` detects the framework you are using. The instructions for that specific framework are used so that you do not have to think about how you need to configure a library for a certain framework but rather simply could start using the library.

By providing the most common way to use a certain library in a single command, we can quickly build complex apps without the hassle of knowing the exact details of how to set up and configure the library correctly, given the framework you are using.

Another benefit is that you can learn how to configure certain libraries yourself by looking at what changes in the source code. You can always quickly set up a project with `flaze` and try out a library or look at what steps are required to use a certain library with your framework. For example, if you are mainly an express backend developer, and you want to learn how to set up a Prisma database, you can simply call `flaze use prisma`, and you have an example of how `Express` would work with `Prisma`. However, if you want to know how to do the same thing in another framework, for example, `NestJs`, you can reuse the same command `flaze use prisma` and look at how `Prisma` would be configured using `NestJs`.

4.2.5 Detect Command

The `flaze detect` command can be used to detect the framework and libraries in a given project. This can be helpful to make sure you can use the `flaze` toolset. It can give you an overview of what frameworks are supported and what libraries are already in the project that are supported by `flaze`. This tool is developed as an extra tool that could be useful but is not developed to improve the developer experience massively. Suppose a framework or tool is not supported. It will simply not be listed or be listed as undefined or unsupported.

A helpful way to use the detect tool is by running detect on a project that was not created by `flaze`. If the tool detects the frameworks and libraries correctly, we could probably use other `flaze` commands that use the frameworks and libraries already installed. Initializing all the libraries using `flaze` is always better. This ensures there will be no unexpected behavior if the user creates no other conflicts.

Limitations

We can only detect the frameworks and libraries that are supported by the `flaze` toolset. For every new framework or library, we want to support, we need to extend our toolset with the correct instructions for that framework. More on this later.

Benefits

We can quickly get an overview of our project's packages, the frameworks they are using, and the libraries they have installed that are supported by `flaze`.

4.2.6 List Command

The `flaze list` command lists the frameworks or libraries that are supported by the `flaze` toolset. This can be useful if the developer quickly wants to get a list of the supported libraries so that he/she gets an overview of the possibilities with `flaze`.

Another method to use a `flaze` list is to learn what libraries or frameworks you can use to solve a given problem. We can use the `--tag` option. This allows us only to list the libraries with a given tag. We can then use tags like 'routing', 'database', and more. The list command will then return a list of, for example, routing or database libraries you can use. This can be useful if you want to test a new library or want to learn about what libraries are available.

Limitations

A limitation, for now, is that we can not support more complex search queries. Because for this thesis, we only support a small set of frameworks and libraries, this is not the biggest problem. However, the list could be difficult to use if we have a large list of supported frameworks and libraries. So it would be useful to be able to narrow it down by methods other than tags.

Another problem is that it is difficult to know what tags are available. If we try the tag 'db', it may not return the results we expect because it does not know that 'db' stands for 'database'. This could result in the user thinking there are no libraries for a given problem. It is important to be consistent or provide the user with a list of possible tags. Alternatively, we could extend this with an advanced natural language search.

Benefits

We can quickly get an overview of the available frameworks and libraries without leaving the terminal. We can also quickly list libraries by tag to see all the libraries with tags such as routing, database, etc.

4.2.7 Generate Command

The `flaze generate` command provides the option to the user to generate boilerplate code for a certain problem. This could be as simple as generating the boilerplate code for a component. For example, a react component. However, we can also generate more complex things like pages and resources. We could implement more complex boilerplate code generators later, but we will start with these three possible solutions.

As with the `flaze use` command, the generate command first detects the package you are working from. This is useful to generate different boilerplate-code based on your working context. For example, if we want to generate in `React`, this component will look different if we run the generate component command in a `Vue` app. This is the power of having a consistent API that can generate code depending on your existing technologies.

The most simple version of code generation is used when working with components. We create a file in the components folder, in the correct casing, and generate the code, with the name depending on the name provided by the generate command. We can compare this type of code generation to context-aware snippets.

A more complex example is the code generation for a page component. This code is more complex because we may need path or query parameters for which we want type safety. So when calling the `flaze generate page` command, we need to check whether or not the page component we generate needs some boilerplate code for retrieving the path parameters or query parameters. This way, we do not need to worry about these basic pieces of boilerplate code. Another interesting thing with page generation is that some front-end libraries do not support pages by default. This means libraries, like `react-router`, can also support generators. In the case of `react-router`, we need to worry about more than just generating generic code. We must also insert the new pages into the router array in the `main.tsx` file.

Lastly, we look at the resource generator. This best shows the capabilities of what we can do with `flaze generate`. In its simplest form, the resource generator generates one or multiple files to define all the CRUD operations for a REST endpoint. This can get more complex depending on your packages. For example, if we add a database, we could also generate code for accessing the database. We want to make sure that everything we generate has the best possible type safety. For example, when using `Prisma`, we already have the typesafe `prisma-client`. When we abstract that database call into another function, we want to ensure we can still access the type the abstracted database call expects. A method of doing this involves parsing the `schema.prisma` file and generating the correct types from that `Prisma` file. This allows us to insert these generated types into our resources to have full type safety.

A database is not the only thing that impacts the default CRUD source code when creating a resource. We could also integrate, for example, authentication when we add support for libraries like `passport`, `auth.js`, etc. This way, we generate boilerplate code for the framework you are working in and the libraries you are using with the framework.

Limitations

A common problem with generating code that is injected into an existing codebase is that there could be conflicts. This is because we can not predict what code the user is going to write or how to user is going to make changes to the code. This can result in code that is generated based on assumptions that the user has not edited a certain piece of code. When such conflicts arise, there is the possibility that the code is not working. A possible technique to improve this type of problem could be to inject code using better heuristics. For example, we could inject code after a certain statement or even after a certain comment. However, the user may remove this information. We could also use a different technique, where we first search for an instance of that type of statement or try to detect where previous code was generated, even if this was refactored. This could be done by using the TypeScript API to look for references. We could then apply more generic methods to inject code. The only problem is that this could become very complex.

Another problem is that when we try to generate code where code is already written, we may not be able to detect this correctly. The code that is already written by the user could be overwritten. This could be the result the user was hoping for. However, it may not be the case. We could generate only the code that does not affect user-written code or try to update existing code. This could be safer than overwriting everything.

We discussed earlier that libraries could be used to overwrite or extend code that has been generated. This can be problematic when a user uses multiple libraries with the same end goal. For example, two databases. To make sure no conflicts are created, we should be able to make sure that these libraries do not conflict with each other or that we could select the library we want to use. If done incorrectly, the generated code could be wrong or not usable. Related to this, it might also be the case that we don't want to generate code based on the database we have installed. Maybe we want to use another method. The user should be able to pick for which libraries the code is generated or be able to undo the action or switch to code that does not include certain libraries.

We also noticed that the generate tool is not always used for smaller types of code generation. For example, we did not use it much to generate components. This is probably because the code generated is only five lines long. This means that the command is not always faster. It could also be possible that the user does not exactly know where or when to use the command or that the user does not know that the command exists. A possible solution is the `watch` tool. This tool will be discussed below.

Benefits

A benefit of using the `generate` tool is that you need to write less code and that you can generate boilerplate code so that you don't have to spend too much time doing the same thing over and over again. The tool also allows you to generate complex code based on multiple different libraries that the user has installed. This can reduce the complexity of integrating all the tools yourself with the current framework you are working with.

Because the tool generates code, you can simply edit and extend the code that is generated for you. This means that you are not limited by the things you can generate when a small part of the generated code is not needed. It can simply be removed or updated. This allows us to write custom code with the benefit of not having to do all of the work setting up the boilerplate.

4.2.8 Watch Command

The `flaze watch` command was created because of the limitations of using `flaze generate` for simple things like generating a component. Because we are so used to creating files the normal way, especially components, and quickly typing out the boilerplate code, it could sometimes be annoying to remember the command to generate a component rather than just creating the component yourself. Even if it takes more time, we sometimes do not like to use a command to create a simple file.

However, the watch tool tries to omit some of the use of generate tools by watching the filesystem for changes. By watching the filesystem, we know when a new file is created, with what extensions the file is created, and where the file is created. This information can then be used to fill the file with code directly after creation. This can allow for context-aware boilerplate code generation.

The most simple example is when you create a component file. Let's say you are using `React`. The convention there is that when you create a `.tsx` file, you would probably want to create a component. However, we can take an extra step of precaution and verify if the component file is created in the `./src/components/**` folder. We are even more sure that the user is creating a component in `React`. As soon as the user creates a component file, for example, `Button.tsx`, we can inject boilerplate code into `Button.tsx` that creates an empty "Button" component in `React`. So that when the user does not want to think about the generate commands, the code is automatically generated on the fly.

This can be useful for simple things, such as components. However, more interesting things can be done with pages, for example. In "next.js", to create the page /item/:id, we need to create that file at `./src/pages/item/[id]/index.tsx`. So if a new file is created in the filesystem under `./src/pages/**`, we can assume the user wants to create an empty page component. However, we do not only generate a generic page, we see that the path the `index.tsx` file includes a path parameter. We can incorporate this information when generating the page code.

The last example shows how we can use this `flaze` command to generate resource boilerplate on the fly. For example, when using `NestJs`, when a folder is created in "./src/modules/**". It is assumed to be a resource. However, in `NestJs`, we need three different files, as do we need to import the module file into the "./src/app.module.ts" file. However, when we detect a new folder is created, for example, "./src/modules/user", we could automatically create the needed files to create a module, service, and controller and insert the module into the main app module. This way, by tracking how we normally create files, we could assume what files need to be generated without the user having to remember what types of code generation are supported.

Limitations

The `watch` tool can be very unpredictable. If the user does not know what can be generated, he/she can be confused if the watch tool injects code that the user did not want to create. This can lead to confusing the user or generating unwanted code. This is because we may assume incorrectly that the user wanted to generate code based on creating a certain file. It is important that we can undo these changes or that we have a way to verify if we wanted to create a generated code. This will allow the developer to benefit from context-aware snippets while also being able to ignore or undo them if this was not the intention.

Because there are also cases where multiple files are generated, how do we know when to generate multiple files or when to generate only one file? This can either mean that the user is overwhelmed by multiple files being created or that he/she thought more files were going to be created. We can try to provide the user with the option to define what watch generators are run when. However, this could lead to a lot of configuration, which could have an impact on the usability of the tool.

Benefits

A benefit, however, is that we do not need to remember the commands to generate code for us automatically. This can be useful when you are focused on programming and forget to run a command. However, by using the watch tool, you could benefit from code generation without the need to run a command in your terminal. This could result in a seamless workflow when programming.

It is also useful that the code generated is context-aware. For example, if we create a `.tsx` file in the `components` folder, the code for a component would be generated. However, if we create that file in the `pages` folder, code could be generated for a page. This results in the correct code being generated depending on the context the user finds himself/herself. But we can also generate different code based on the framework or libraries we are using without the need to use to select the template you want to use manually.

4.2.9 Introspect Command

The last CLI tool will be the introspect tool. This tool allows you to introspect, most of the time, parse the existing codebase to reuse information retrieved from the codebase to generate new code. This can be useful when we have written a lot of code and want to retrieve an easy-to-use format from that code that we can use to generate other code.

Examples of code introspection are, introspecting the server API, introspecting database tables, introspecting the .env files in your project, introspecting client routing code, etc.

A common thing we do when developing web applications is implement the server API into our client. Or even just type all the information into Postman. However, if you do not have a scalable way to do this, it can be difficult to maintain a consistent version of your API throughout the server, Postman, and the client application consuming your API.

To solve this problem, we need to be able to create a single source of truth. The best option for this is the server implementation. We pick the server because this already contains the most information about the API itself because it literally contains the code that builds up the API.

However, we now need to look at how we can consume this API using Postman and our client. This can be done by introspecting the server code to generate an intermediate format from the server API. This intermediate format can then be used to generate both the Postman format and a generated client API.

NestJS API introspection

Let's take a `NestJs` server. To define a new resource, we need to create a controller file. This controller file can be used to introspect most of the API information. To introspect the source, we make use of the `TypeScript` Compiler API.

The controller class contains a controller decorator which we can extract the base path from. We can also read the name of the class and use that as the name of the controller or resource.

To parse out the endpoints, we need to look at the different methods defined in the controller file. When a method in the controller class contains decorators with the request method, we assume the method contains the code for handling a request to the endpoint defined in the decorator. From this method, we can then extract the following information.

We get the request method from the decorator, we get the path from a parameter in the method decorator. We can use the method name as the name of our request. To handle (query) parameters and the request body, we provide parameters to the method prefixed with a decorator of type ”Param”, ”Query” or ”Body”. We can then extract for each method the name of the parameter, and its type, the name of the query parameter, and its type. And for the body, we could use the type that is provided for the body parameter.

Lastly, we want to retrieve information about the response. An important thing to extract is the return type of the method. This can be useful to generate a typesafe API for the client. However, sometimes the type is inferred, which means that `TypeScript` knows the type because somewhere in the implementation, the correct type is used, and that value is passed to be returned from the function. This thus makes it more difficult to retrieve the type. This can also be the case at other locations where we extract inferred types.

The problem here is that the `TypeScript` Compiler API does not completely support inference. So we can either work with the type we receive from the `TypeScript` Compiler API or try to write our own method to infer types. For now, we try to resolve the type as well as possible and extract the needed type declarations from the source code. However, it would also be interesting if we could fully infer a type so that we can represent that type as a simple resolved JSON file. This way, we could generate a type in `TypeScript` but also use this information for other purposes.

Prisma Schema introspection

Another example of introspection is introspecting the database. We will look at `Prisma` to give an example of how the introspect could work with database models.

In the most simple case, we may want to generate types based on the database models. This way, we can use the types throughout our application and thus also in our client application without the need for the `prisma-client` auto-generated types. This is beneficial because the types generated by `Prisma` are not that easy to access and not usable in a client environment.

However, the types of database models/tables could be useful to generate return types for database calls or as input types if the database call is nested in abstraction layers. As we can see, we could thus use Prisma introspect tools to generate types for a given database model when we are generating a resource using `flaze generate resource` to generate fully typesafe code instead of using any as the input type. And later, when we introspect the API, the types will be easier to carry over to the front-end code.

Limitations

It is not always possible to generate code. Sometimes the developer makes use of runtime features that we can not extract. For example, in NestJS, we define the path of our endpoint as a string. However, we could also, for some reason, define that string as a variable or even use different functions to return a string. This could be a problem when we want to extract the path as a string because we may not be sure what the output will be without running the code. However, most of the time, this is not a problem.

Sometimes, we need to assume things about the code to introspect it correctly. Maybe the user is using a different library that affects the assumed code, which could result in incorrect inference. For example, we may be able to pass a validation object to a custom decorator into the source code. This code then uses the decorators used by us to introspect the information. However, we may not know that the developer has created an abstraction on top of a certain framework or library that we want to introspect. This could result in incorrect introspection and could break the whole introspection process.

When using introspection to extract types, it could be difficult to cover all edge cases to correctly extract a full type with different dependencies using the standard `TypeScript Compiler API`. This could lead to incomplete or incorrect types. And because these types are inserted into other code, it could result in an unusable `TypeScript` file because that file will always result in type-checking errors. We do need to make sure that the types we extract are correct and complete. This means that we also need to export other reference types referenced in the main type and make sure these do not conflict with types that have the same name and more.

Because the code the user writes can be unpredictable, it is not always possible to implement introspection for every edge case. This can be frustrating whenever the user needs a feature that interferes with the introspection. We could allow the user to implement their own strategies, but not every developer wants to do this, can do this, or has the time to do this.

Benefits

When introspection is used with large codebases, the time that you can save automatically introspecting code and using that to generate typesafe code could have a big impact on the developer experience. You can use a single source of truth to generate all the code you need and consume it in other parts of the web application. This can save a lot of time. We also do not repeat ourselves. This would be a waste of time in most cases, so we should strive to solve these types of problems.

When changes are made to a codebase, it can be difficult to update all the other sources of truth manually. However, when we can simply regenerate the code based on what we introspected to save us time. This means that types and code are kept up to date and that we do not need to worry about forgetting to update something. If we are still not sure, we can just regenerate.

4.2.10 Studio Command

The last command, `flaze studio`, starts up a dashboard to get an overview of the applications in the current directory. Note that we need to run `flaze studio` from the project root to get access to all the packages.

Once we open the dashboard in the browser, we get an overview of all the packages in the project's root directory. For example, 'nest-app (nest)', 'react-app (react)', etc. We can then select the package we want to interact with. By clicking on the package, the left navigation menu will update with the available tools for the selected package.

The configuration tool is the first tool available via `flaze studio`. This tool introspects the .env files in your project and visualizes them. Here we can quickly get an overview of provided env variables and update them if we want to. This tool can give you basic insights into the configuration of your application.

The next tool is even more interesting. This is the API tool. This tool will provide a `Postman`-like interface allowing you to interact with the server API defined by the codebase. To make this possible, we will use the introspect tool for the APIs to generate an intermediate format to represent the API. We can then use that data to provide the user with the list of controllers and their endpoints so that they can quickly test out the API without having to do anything other than run the `flaze studio` command.

This tool helps the developer interact with their server, even when they have not implemented the client application yet. Or even when they have implemented the client, it could always be useful to test the API or quickly make an API call with the latest required parameters, body, etc. We can also use this to quickly check the request response to ensure we are retrieving the correct data.

The next tool is similar to the API tool. It is a database tool. Instead of introspecting the server implementation to retrieve the API, the codebase will be introspected to retrieve the correct information to make a database connection. This tool will then allow you to list the tables found in the database and view the row data when you select a database table.

This can give the user insights into the data when developing the application. This also verifies to the user that the database works and that the correct data is stored there.

The pages tool allows you to introspect the different pages that are defined in a front-end application. We can then quickly visit the page to test out the page itself. This can give you a quick overview of what pages are available for a given package and their required parameters.

As you can see, whatever we can think of, we can build a tool for it. This means that we could implement even more custom tools to provide the user with information from the codebase or interact with that codebase that would otherwise not always be straightforward.

Limitations

Because the flaze studio relies on introspection, the same limitations apply here. If the introspection is incorrect, some features might not work. However, there are no other disadvantages. This is because you do not need to use this tool. It is just there to help you provide a better experience when building applications for the web.

Benefits

The user can get insights into the codebase, the API, the page routes, the configuration, and the data stored for a given package directly in the browser. This can be very helpful to make sure everything is working correctly and to perform actions quickly.

4.3 Extending Flaze Toolset

An important part of the `flaze` toolset is that we have a good developer experience when using the toolset and a good developer experience when extending the toolset. This way, we can implement new frameworks, libraries, generators, introspect modules, or even studio extensions to use the toolset for any software stack we like.

4.3.1 The provided API

To allow developers to extend the flaze toolset rapidly, we provide a set of APIs that allow the developer to do common tasks when generating or injecting code into an existing codebase. It is important that this API is not too complicated. Otherwise, the process of adding support for a given feature could be too complicated. This would make it more difficult to extend the toolset, which could result in a negligible number of supported frameworks and libraries.

When building the different tools, a set of commonly recurring actions are performed. This first thing is the execution of existing commands. It should be easy to perform bash commands without needing external libraries. Luckily this is already supported by `Node.js`. Yet, we still provide a wrapper around this API to allow the developer to add a loading message while the command runs. The wrapper is called the `ExecUtil`. This allows us to execute commands using the `exec` method or a batch of actions using the `batch` method. This method also has the possibility to add a loading message to provide the user with information about what is happening.

Besides executing commands, it is also common to perform certain actions on files. This can be done using the `FsUtil`. The `FsUtil` provides basic functionality to perform actions such as file creation, file reading, file appending, and file removal. It also supports recursive directory creation. However, this is also supported for files. This means that if a file does not exist, it also automatically creates the necessary folders for you. Besides the basic functionality, the util also provides methods of replacing, removing, appending, and prepending based on certain patterns. For example, we can append a line of code after the last, starting with an import statement. Because there are a lot of different string utilities, we created the util so that you can pick whatever type of matching you want.

Other useful utilities are language-specific utilities. These include utilities such as `TypeScriptUtil`, `DockerUtil`, `EnvUtil`, `PackageJsonUtil`, and more. The goals of these utilities are that you can use them to read or alter certain files written in a specific language. For example, if we would like to retrieve the value of a certain environment variable, we could use the `EnvUtil`. If we wanted to read the name of a method in `TypeScript`, we could use the `TypeScriptUtil`. These utilities can be useful if certain tasks for a specific language need to be easily accessible.

4.3.2 Add framework support

To support a framework in `flaze`, we create a new class for the framework and extend it from the `AbstractFramework` class. This class will allow the developer to provide a standard port, the name of the framework, a set of aliases that could also be interpreted as the framework, a set of generators, and a method to create or initialize the framework as well as a method to detect the framework after it has been created.

When creating the framework the command `flaze create <package-name> <framework-name>` is called. This information is used by the create command to initialize the package correctly. Once the command creates the root directory of the framework, information, such as the directory and name, is passed to the framework create method.

The create method is supposed to do the following. It must initialize the framework by downloading the correct files or scaffolding them by manually implementing this within the `create` method. It is important to provide a clean installation or clean-up files that are not needed to start the application. Another thing that needs to be done is to implement the default configuration. This includes setting the correct port for the framework. By following these guidelines, we provide a consistent way of extending the supported frameworks.

Once we have implemented all the necessary methods, we can simply call the `create` command and test out the framework initialization process.

4.3.3 Add library support

A similar process is needed when we want to implement support for a library. However, implementing a library can be done for different types of frameworks. Because of this, we need to create a library class that extends from `AbstractLibrary` and a class for each framework we want to support. This can be done by extending the `AbstractLibraryExtension` class. The benefit of a central library class is that we can implement core functionality here so the different library extensions can reuse it. The abstract library requires information such as a name, a set of aliases for the library name, a set of tags used to search for the library, and the set of extensions that the library implements. A library extension requires you to implement a framework or library name you want to create the extension for and a set of generators for the library extension. As with the `AbstractFramework`, the `AbstractLibraryExtension` provides a method `use` to initialize the library as well as a method to detect if the library is already in use.

When creating the library, the command `flaze use <library-name>` is called. But, before the `use` method of the specific library is called, the framework that currently is being used, as well as other information, is passed to the `options` parameter of the `use` method. We could access the framework's name, package information, the current and root directory, or even information like the port number of the framework. This allows good integration with the already installed framework.

4.3.4 Add generator support

We could also generate code for a framework or depend on different libraries. To allow this type of functionality, both a library and a framework have the ability to have a set of generators. The framework generators will be called first to provide a bassline when the generate command is called. Afterward, the library generators will be called. To create a generator class for a framework or library, you need to extend from the `AbstractGeneratorExtension`. This will require you to provide the framework and/or library name and the type of generator you want to create. This can be a `component`, `page`, or a `resource`. Lastly, there is a generate method you need to implement. This will provide information about what the user wants to generate based on the `generate` command specified by the user and also will include information about the current package, framework, directory, and libraries available.

To generate code, the command `flaze generate <type> <...parameters>` is called. The parameters will already be parsed so that the implementation of the generator can already use this useful information. For example, for the `page` generate command, the provided URL path is parsed into parts that either a parameter's name. This type of functionality can be reused by every generator that tries to implement page generation. This allows for seamless and fluent integration of different generate tools.

4.3.5 Add introspect support

The last command we could extend is the introspect command. This can be implemented for different frameworks and library combinations as well. We could create a class that extends `AbstractIntrospectorExtension` to create a custom introspection tool. This will require you to provide a framework and/or library name. Besides this, we also need to implement a `introspect` and `generate` method. These allow us to first introspect into an intermediate format before generating code. We could also provide additional generators. For example, we could create an API generator for a specific framework or to generate an alternative format, for example, an Open API specification. And because we use an intermediate format, every framework or library that supports an introspect could now generate all the other implemented generators. This allows us to expand the possibilities of what is possible with introspection.

Chapter 5

Evaluation

In this section, we will discuss how we evaluated the toolset that we built for the thesis. The section on evaluation methods will discuss the possible methods we could use to evaluate the toolset. The next section will discuss the evaluation methods we selected for this thesis. The final section will discuss the results of the selected evaluation methods.

5.1 Evaluation Methods

5.1.1 Internal testing

Internal testing is a testing technique where the developed software, in our case, the `flaze` toolset, is tested by the developers themselves. Internal testing can involve different methods to test all aspects of the toolset. We will test real-world scenarios, testing its functionality, performance, and usability. However, we will not focus on performance that much for this method unless it becomes a noticeable problem. This method can be useful because it can test our toolset quickly and could result in a user-friendly and feature-complete software tool.

A possible technique to test the toolset is to use it to create a simple project. If the toolset allows us to create quality applications efficiently, it is useful in helping us achieve that goal. If we still struggle to build these simple projects, we need to improve the tooling until we can efficiently create these projects.

We can iteratively improve the tool by repeating the same process repeatedly. The first step in this process is using the tool to build a simple application. If the tool already struggles with this simple project, we need to look at why that is. When we have found a solution for these problems, we need to verify that we can confidently use to tool to help us build the simple project. Once we have verified that, we can try to build a more complicated project. We must pick a more challenging project if the toolset can handle it. If not, we can repeat the cycle and improve the toolset to be able to handle the more complicated project as well. It is important to note that we should still be able to use the tool for the simple project. We should also make sure that the toolset does not become too complex. This could result in a worse developer experience than before.

Another important aspect of the toolset is that we can easily extend it. So as developers of the toolset, we need to make sure the tool not only provides a good developer experience for the developers using the toolset and a straightforward system for extending the toolset. We can also test this internally. However, as long as the internal code of the toolset is scalable, there should not be that many problems. But if we notice that adding support for other frameworks or libraries becomes more complex, we should definitely search for better ways to support this functionality.

5.1.2 Beta testing

Beta testing is a method that uses a small group of external test users or, in our case, developers that are willing to test the software or, in our case, our toolset. This could be useful because we can only access a small group of developers. We could split the beta tests into small weekly tests to use these results to improve the toolset iteratively.

For a beta test, we could ask the participating users to follow an instruction document to create a simple conceptual web application. This could give us insights into what the main bottlenecks are when using the provided toolset. This process is similar to iteratively improving the software using the internal testing method. However, by using external test users, we expand the scope of test users and thus get better insights into the toolset in the form of feedback, bottlenecks, improvements, pain points, and possibly other metrics.

A problem, however, is that when setting up the instructions to build a conceptual web application, we need to make sure that we do not limit the scope of the project to only what the tool is capable of. Because this can result in positive results for the given use case but can give different results in real-life use cases, this is not always easy because the tool is developed based on the common problems in web development detected by myself. This could mean that the problems that are common for me when developing web applications do not always are applicable to other developers. It is thus important that during testing, we retrieve further information about the tool and its capabilities using other methods.

5.1.3 User surveys

A user survey is a method of retrieving feedback about the system you are testing by getting in contact with beta testers. What is their experience with the toolset? What are some of the problems they have encountered? What are suggestions for improvement?

The last problem we discussed above can possibly be solved by this method. By doing user surveys, we can make sure that the problems we are solving are globally accepted as problems with the developer's experience when building web applications. This can be done by not asking questions about the toolset itself. But first, create a survey that gathers information about problems with the developer experience itself.

A problem with this approach for our main use case is that the user first needs to have used the toolset we created before being able to answer the questions asked by the survey. If done incorrectly, this could lead to incorrect data collection and affect the survey results. This also takes away the benefit of doing online surveys. If we don't have control over our environment, the survey could lead to incorrect results.

However, we could conduct small surveys for the users that have already participated in the beta test. But because we are testing iteratively, this could not be the most practical method for doing this. This is because there are better methods to ask these questions in person. So we may not benefit as much from a user survey as from other evaluation methods.

5.1.4 User interviews

User interviews are a method of gathering feedback from the pilot testers by interviewing them one-on-one. These interviews give us in-depth insights into the experiences of the test users, their pain points, and other useful information when using the toolset.

A good way to do this is by using semi-structured interviews. In this type of interview, the test users are questioned based on a predefined set of questions. However, when interesting feedback is provided, or follow-up questions come to mind, the interviewer can ask these questions to gather more information besides the predefined questions. This allows for the combination of both a strict question set and the benefit of more feedback with the questions that are asked as follow-up questions.

5.1.5 Feedback Sessions

Feedback sessions are used to directly interact with pilot testers to address their questions and concerns. Feedback sessions can have benefits over predefined questions because we can gather information out of the scope that we are focussed on, which could lead to additional ideas and inspiration for developing more qualitative software.

For the study we conducted, it would be helpful to get user feedback when they have tested out the toolset. This could be done by using feedback sessions. However, this could possibly be combined with a semi-structured interview. This is because we can extend the predefined questions with on-the-spot questions for feedback and possibly even further brainstorming.

5.1.6 Comparative study

A comparative study would involve comparing the usage of the toolset to develop an application in comparison to building an application without the toolset we created. This allows us to see the impact of using our toolset.

However, this may not be ideal for this thesis. An important consideration is the learning effect. This is when the test user is already familiar with what he/she needs to do because the first half of the comparison gave an in-depth overview of how to build the demo application. This would have an impact on the second half of the comparison because we would build the same project with or without the toolset, depending on what we used for the first user. Even if this could result in subjective feedback from the test user, we think that this would not be the best method for getting that type of feedback from the test user.

Another disadvantage of this method is that it will probably take much longer to create the applications without the toolset to generate boilerplate code. This could result in long sessions where the user needs to stay focused. This is also a common problem with this type of evaluation if the thing we want to test has a large scope.

The last problem with this technique that we will discuss is that we need a big enough group of developers to make sure the results we conduct from this test are significantly relevant. Because of time limitations and the number of developers that are intermediate web developers that would be able to take part in this test, this could lead to limitations in the results we would normally get.

5.1.7 Documentation review

An important aspect of creating tools for developers is the documentation. It is thus important that the documentation for the toolset is evaluated so that everything in the documentation is clear and easy to understand. Reviewing the documentation can thus be an important aspect to help improve the learning rate of using your developer tools.

Because documentation consists of a lot of written English text, we could benefit here from the fact that we don't always need domain experts to help improve the documentation here. Of course, we should not only focus on clear text, but the correct technical information should also be clear. However, we could benefit by using both parties to improve the documentation and thus provide the developers with a better understanding of the capabilities of the tools we provide.

5.2 Evaluation

Because of the limited timeline, we were only able to evaluate the toolset using a small selection of the methods discussed above. The main method of evaluation is internal testing. Because of the limitations of this type of test, we extended this evaluation method by conducting an additional beta test. In the preceding sections, we will discuss the decisions behind these choices.

5.2.1 Internal Testing

Because we are building a tool for developers, internal testing seemed an interesting option. This is because we, web developers, are users of the evaluated domain ourselves. We could use this to our advantage to quickly iterate when developing the toolset. However, we need to make sure that we aim to improve the developer experience not only for ourselves but for all other web developers that would use such a tool. This is why we also decided to conduct a beta test with a small group of external developers. The results of the beta test could then be used in addition to the feedback from internal testing to improve the developer toolset iteratively.

The benefit of internal testing, in contrast to other methods, is that we could iterate more quickly. This eliminates the need for communication with external domain users. Whenever we build a part of the toolset, we can evaluate it ourselves and implement possible improvements.

5.2.2 Beta Testing

The main problem with other methods of evaluation was the large scope of the toolset, in combination with the fact that the domain users are web developers. An important aspect is that the test users are not impacted by fatigue or similar factors. This could have a negative impact on the end result. To ensure the user says focussed, we can split the beta test into smaller iterative beta tests. This is more difficult when using other methods where more users are needed to draw objective conclusions.

Let us say that we decided to use a user survey as the main evaluation method. A problem with this is that the user must have used the tool before being able to give useful answers to questions in the questionnaire. This could lead to fatigue, and not a lot of testers like long, time-consuming sessions. And because a beta test can be done iteratively. We decided to perform multiple sessions where we could question the user. This allowed us to divide the large scope of the toolset into smaller sections.

5.3 Results

In the first section about Internal Testing, we will discuss the different iterations and conclusions we conducted at the end of each iteration. In the next section, we will more closely look at the conducted Beta Tests.

5.3.1 Internal Testing

Because the toolset consists of tools that are somewhat dependent on each other, we will start with building the Initialization Tools, followed by the Scaffolding tools, and so on. With each tool, we will also discuss how we evaluated that specific tool of the toolset.

The first command we implemented was the `flaze init` command. Because this tool only creates a `.gitignore` file and a `packages` folder, we could run `flaze init` and validate this. Because we did not need more at this moment in time, we were able to start developing the next command.

The `flaze create` command allows the developer to initialize a framework of choice as a package in the `packages` folder. When implementing this, we first build the command to only work for one framework. This allowed us to focus on the idea itself instead of writing modular code from the start. When we had a working version of a single framework, we identified the different steps needed to initialize a framework and strip it down to a bare minimum installation of the framework.

When we knew the steps needed to build this command, we added support for a second framework. In this initial phase of adding the second framework, we used an if-statement to decide which framework we should initialize. By doing this, we could quickly test the different frameworks to get a better understanding of how the command would work. Now that the code to initialize both frameworks had been written, we could identify the shared code and abstract that code into an abstract class. This class is called `AbstractFramework`. When we now want to implement a new framework, we could simply create a `TypeScript` class for the framework and extend from the abstract class. This allowed us to implement the required methods in the abstract class consistently.

To evaluate if the `flaze create` command was working, we would first call the command to initialize the framework. Secondly, we would start the framework by running the framework-specific `npm` script. This would serve the framework on a random port.

The problem with this was that the different frameworks contained inconsistencies in their configuration. This was frustrating from a consistency point of view. This allowed us to come up with a solution. The solution we selected would include configuring a standardized `npm` script and a standardized port. For now, the `flaze create` command was complex enough to move on to the implementation of the next command.

The `flaze use` command allows developers to initialize different libraries into the package the developer is currently using. As with the `flaze create` command, we would start off by creating a minimal version of the command that initialized a library for a hard-coded framework. This allowed us to test out the concept and get a better understanding of the steps needed to integrate a library into an existing package. To evaluate if this was working, we ran the command after we created a framework with the `flaze create` command. We could then serve the framework to test out whether the library was working correctly. This was not always the case, sometimes, multiple iterations were needed to write the correct code.

This was mostly because of the fact that we did not have access to linting in the string templates used to scaffold the code for the library. Another problem was that some libraries required injecting code into the existing codebase. In the early stages of development, this was done using a naive implementation. We would, for example, remove certain lines of a given file. This was, of course, not scalable. Because of this limitation, we would later build more complex APIs to allow the injection of code into the existing codebase.

Another benefit at this stage was that we could start building simple applications using the provided toolset. This allowed us to evaluate the toolset from a different perspective. Whenever we would run into problems, we would be able to reevaluate the previously created tools.

To make sure the toolset provides a good developer experience, we would like the developer to use the toolset to have to perform a minimal amount of steps to configure a certain library for the current framework. Otherwise, this could result in the developer having a more complicated quest to find what when wrong and give preference to performing the instructions to install a library manually instead of the automated approach. For example, when we install the styling library `Tailwind` into our package using `flaze use tailwind`. We expect that the `Tailwind CSS` file `index.css` is already imported correctly into the framework. Otherwise, the developer could start a quest why `Tailwind` is not working or even decide to manually search for instructions to configure the library for the given framework correctly. This could then result in a worse developer experience. However, these kinds of problems can be fixed by iteratively testing out if the application is working correctly without any additional steps unless specified otherwise.

At this stage, we decided to implement a set of common libraries that could be useful for developing applications. To decide on what libraries to implement, we could also try to build a simple application and see what types of libraries we are missing to complete the development of the simple application. For example, when working with client-side frameworks, we often want to create different pages for our application. Another common thing is that we create a lot of components when developing web applications. And lastly, we would often use styling libraries to handle styling properly. On the other hand, when developing server-side applications, we would probably need other libraries that handle things such as databases, authentication, and more. However, the types of libraries required for a given project can differ. We decided to implement these types of libraries into `flaze` because they are well-known basic building blocks for any modern web application.

To implement the different libraries, we would first need to streamline the process of building libraries for the `flaze` toolset. For this, we use an abstract class called `AbstractLibraryExtension`. When extending from this abstract class, we are able to provide the name of the library and framework. This allows us to select the correct library initializer for the correct framework automatically. However, how do we know what framework we are currently using? This is possible by extending the `AbstractFramework` with the possibility to detect itself. When we implement this, we can write a utility that is able to detect the active framework. After we implemented this, we could try to build a simple application and see where we are held back.

At this stage, the next problem occurred. When developing the actual application, we noticed that the developer was writing a lot of duplicate code that could easily be generalized. The first example we briefly mentioned above is that the user builds a lot of components when developing client-side applications. We could abstract this into a tool that can be used to generate common recurring pieces of code. We described this tool as the generate tool. A tool that generates code templates based on the data the user provides when they are created.

The `flaze generate` command does exactly that. It generates code given some additional user input. The first generator we built was a generator that generates code for a component in a front-end framework. For example, when the developer is using the `React` framework, the correct `React` component code is generated in the corresponding `components` folder. We generate the component in the `components` folder because it is a convention used by a large group of developers. If we were to generate a component when we are currently in another type of front-end framework, different code would be generated. To support this functionality, we use a similar system as the abstract classes we explained above. In the case of generators, we extend the abstract class called `AbstractGeneratorExtension`. By extending this class, we can correctly implement the properties and methods so that the generator can be automatically selected.

Similar to components, we could also create pages. This is because pages are commonly needed in these applications, as discussed above. As we think about adding this functionality, we can see that there is a possible problem here. What if the framework does not support pages by default? We then need to install a library first. This means that we should be able to create generators for not only frameworks but also possibly framework library combinations. By implementing this, there are a few side effects. What if both the framework and the library generate different code? How should we solve this? We can solve this by overwriting or extending upon the code generated by the generator that is called first. In most cases, this will be the framework followed by the framework library combinations. This will not only be useful for generating pages, but we will learn more benefits of this mechanism in the following section. Having solved the problem of generating different code if a certain library is installed, we could move on to the following needs of our toolset.

Because we have two generators for generating client-side code, we would also like to create a generator for the server. We will again look at common recurring code and find that resources or REST CRUD operations are often recurring. To solve this problem, we created a generator for resources. However, if we generate a CRUD resource without knowing if the developer wants to use a database or other form of storage, we can only generate a limited amount of code. We can implement the API but can not generate the service that stores and reads the data. We can solve this using the technique learned above. We can write a database generator so that when a database is installed, the resource generator will also be called for the selected database library if installed. This will result in code generation for storing the resource in a database. This allows us to scaffold a simple server with basic CRUD operations quickly. We can combine this with a simple client application with some components and pages.

This allows us to test out the limitations of the current possibilities of the toolset. We can build simple CRUD applications without writing much code. However, it still takes a lot of time if we want to manually implement the server API into the client we have selected. However, we can notice that we already have implemented the server, which contains the necessary information needed to implement the client API. If we could somehow reuse this information to automatically generate a client adaptor so that we could utilize the work we have already put into creating the server API. A possible method would be to generate code for the client every time we generate a resource in the server. However, this is not scalable. What if we update one of the CRUD operations for a given resource or decide to manually create a resource without utilizing the generator? A solution would be to introspect or parse and understand the code that is already written to translate this server code to generate code for a client adaptor. The details of how this is implemented can be found in the section about tooling.

The `flaze introspect` command can be used to solve this problem. Similar to the previous commands, the introspect command is shared between different frameworks by making use of abstract classes. We can run the introspect command and select a location where the types should be generated. Every time we now update the server, we can update the client API adaptor automatically with a single `flaze introspect` command.

We repeat the iterative process and try to build a simple application again using the toolset. This results in the problem that we still need to manage the client-side state in some front-end frameworks. We can solve this problem by generating the state management and the base adaptor. This solves the problem. So now we can build quickly build CRUD applications with full end-to-end typesafe connections.

The API is not the only thing we can introspect. We can also introspect things like the database types to access these types in the server and the client. This could be useful if you want access to the types but do not want to manually maintain the database schema and the `TypeScript` types.

When we ran this version, other problems occurred. This was because we did not enable Cross-Origin Resource Sharing (CORS) on the server. However, we did only notice once we implemented a full application with client-to-server communication. This shows the importance of understanding the end goals of building applications for the web because only at this stage could we again improve the `flaze create` tool to support enabling CORS. CORS could be enabled for all sources. However, for security reasons, we could detect what client applications are present and make sure that only these hosts and ports are allowed to make requests to the server.

This clearly shows one of the disadvantages of developing a tool such as this. Integrating different frameworks, libraries, and more complex applications could require manual implementation by the developer. We try to provide tools as far into the development process as possible.

When the application grows larger, we may want to be able to quickly test out API routes without building a client application. Or maybe you are just building a server application. This is where we created `flaze studio`. This command starts up a web server with a dashboard with different features we, as developers, could use. The dashboard uses introspect tools to introspect the source code. It provides tools such as a visual editor for `.env` files, a Postman-like interface for testing your server API, a similar interface for connecting to your database, and more. These tools extend the developer experience further by allowing developers to easily interact with code that is written but not directly accessible because we need to create a request to an API, for example.

We last noticed that the generate command was commonly used for resources but less for generating components. This led us to think about possible solutions for this problem. A solution we came up with is the `flaze watch` tool. This tool watches the filesystem. And by following conventions such as that components are always generated in the `components` folder. We could use this information to insert boilerplate code into the file when the developer creates a file at a location such as the `components` folder. This also shows the benefit of creating tools that may not seem complex at first but could result in interesting results down the line.

Because of time constraints, this is about as deep as we got into iterating to develop a useful tool for developers that provides a basic infrastructure to build simple applications for the web, allowing you to write minimal boilerplate code yourself. As we can learn from this section, we can see that the process of iterating while using a simple application as a goal could result in useful tooling to create more complex applications. And improve the overall developer experience when building these applications. However, to validate the tooling we have built, we will use beta testing, discussed in the following section.

5.3.2 Beta Testing

Because we did not want to be limited by our own experience of using the toolset, we decided to test out the toolset by performing an additional beta test with intermediate web developers. Because of the limited timeframe, we were able to test two iterations with developer one and did an additional validation beta test to confirm the overall feedback of the first developer. In the following paragraphs, we will first discuss the two iterations performed by the first developer, followed by the validation test.

In the first iteration of the beta test, the goal was to help the developer understand the concept of the toolset and test out the first stable version of the tool by building a simple application. At the end of the session, we also asked the developer to answer a set of questions to give feedback on the toolset and a subjective opinion of the developer's experience of the toolset. The instructions were provided to the developer using a Google document. The results of the questions and follow-up questions were also captured in that document. The source code written by the developer is stored in a zip file and, together with the Google document, will be included as separate documents in the appendix.

To start the experiment, the developer was first instructed to read the instructions for the experiment. If certain aspects of the instructions were unclear, they would be clarified before the experiment would start. The first instruction in the first iteration of the test was to take a look at the documentation. This will help the tester understand what the tool does and what the available commands are. In case of unclarity, the developer could always ask for help, but this would then be captured so that we could learn more about why the developer was stuck at a certain point in time. The last step performed before the actual development of a demo application was a questionnaire to make sure that the developer had a good understanding of the toolset. This would also provide us with information on what we could improve for the application's documentation, as well as the toolset itself.

The next phase of the experiment instructed the developer to create a simple web application that is a subset of a social media application. This application requires to be able to create posts, view a list of posts, and view individual posts. If the developer had extra time and was not exhausted, they could also implement comments for the posts. However, the first developer was not able to complete this because of exhaustion. Before we discuss the problems and feedback captured by this experiment, we will discuss the more technical requirements hidden from the tester. The developer would first need to create two packages, a client and a server, this would be followed by installing the necessary libraries needed for the project. Once we have the libraries we need, we will generate the code for the server that generates an API, provided that we have installed a database. This will generate a fully working CRUD API. We could then test the API, but because flaze studio was not implemented, this would require a lot of extra time. We would then need to introspect the API from the server so that we could access the server in a type safe manner from our client application. Lastly, we would generate pages and components to consume the API and build a functional subset of the social media application.

In the following paragraphs, we will discuss the results obtained by the beta test. When we asked the developer questions about the commands listed in the documentation, the developer did remember most of them; however, because that documentation was not thoroughly read, some commands were still vaguely explained by the developer. However, the commands that were described correctly were the init, create and use commands. The commands that were still unclear were the flaze generate command. When we asked more specific questions like what does the command `flaze generate page /video/:id` do, the user could form better answers because the commands contained more context about what could be generated, created, or used. An important note the developer made was that either the documentation should explain every single thing that happens when a command is run or that we should be able to receive feed forward. This was because some commands do way more than the developer anticipated. The following questions asked about the clarity of the command names. This resulted in feedback that the word `introspect` was not well known to the developer and that the command `new`, now known as `generate`, would be a better name. Another thing we discussed was natural language transformed into commands. However, this is out of the scope of this thesis. The last question asked about how the developer knew the available commands. This resulted in feedback that you needed to be able to list the frameworks and libraries that are supported by the toolset. Or even better, search for frameworks and libraries based on related keywords and the exact name.

Next, the developer was instructed to build the test application. This resulted in useful feedback that allowed us to refine the user experience of using the tool. The developer suggested a better way to get help when he/she got stuck. This could be done by adding a `--help` flag. However, it would be important for the developer to be able to view usage examples when the `--help` flag is provided. Related feedback again suggested a way to not only get examples of what happens when a command is used but also provide feedforward so that when learning to use the tool, the developer can anticipate what is going to happen. A thing that the tester liked was that when a command was run, and additional options were available, a simple usage text was provided to help the user get started with certain libraries. This test also allowed us to detect a bad user experience when using the toolset. These included things such as that no user feedback was provided when no code was generated, incorrect commands would not give information that something clearly was wrong, and when your code was initialized multiple times, no errors were shown, but this resulted in code that would not run. Other small bugs were detected and fixed for the next iteration. A last interesting opinion of the developer was that the tool could have helped him/her a lot more if he/she had known the order in which commands could be run to maximize the code that would be generated. For example, if you generated a resource and you didn't have a database initialized, this would result in code that was incomplete because the resource generator did not know that it should also generate a connection with the database. If he/she had known this in the beginning, the developer would have thought about using the same commands in a different order.

The last set of questions of the first iteration allowed us to get additional feedback on the developer experience of the toolset as well as possible limitations. The main problem with the toolset was that it was difficult to anticipate the order in which different commands could or should be used. This required a basic understanding of the toolset and basic knowledge of software development as well. This could be improved by using feedforward to estimate better what each command exactly does. The tools that were described as helpful were the `create` and `use` tools. This was because the tester did not have to know for given frameworks and libraries how to set them up. The developer could directly start using these tools. The tester would use the tool for quickly setting up a project but not that much to generate code. Note that at this stage, there were only init, create, use, and generate tools. The toolset is also described as versatile and flexible. This would mean that it could be used for all kinds of projects. According to the tester, the different commands of the tool are abstracted in a good way. Because of this, the tool is described as modular and could be easily used to build scripts that setup up a codebase or just in a modular way.

The last question asked about the impact on the developer experience. The test user said that the tool could be useful for people exploring new technologies, this is because the tool takes away the pain of misconfiguring frameworks and libraries. For developers with more experience, the tool could speed up the process of generating code for your application. This could result in a better developer experience from the aspect of efficiency.

Before we discuss the second iteration, we will shortly discuss the course of the evaluation. Because the first iteration required knowledge of the tool, using the tool to build a simple application, and a feedback questionnaire, it took a long time to complete this process, about an hour and a half. This could have had an impact on the overall quality of the experiment. We still learned much by letting an external user test our toolset. Note that because of this, the tester could not completely finish the demo application. Luckily, we could do a second iteration, where the tester could implement the same project with different frameworks and libraries. This allowed us to improve the toolset, fix bugs and reiterate the experiment.

The second iteration was similar to the first iteration. The core differences lie in that the questions about the documentation were replaced with questions to identify if what the test user had remembered from the previous session. Note that the second session was conducted a week and a half after the first session. Another change is that the last questionnaire is updated with new questions that are updated based on the questions of the first iteration.

We will now discuss the second iteration. First, we looked at what the user remembered from the different `flaze` commands. The user correctly remembered the following tools, init, create, use, introspect, help, and the new command list. However, he/she also remembered the other tools partly. The tester remembered only the resource generator and not the page and component generators. This could be because they are not as spectacular as the resource generator. The tester was also able to guess what the flaze studio command did but not what the flaze watch tool did. To make sure the user had all knowledge needed, the parts that he forgot were shortly revisited.

Without spending too much time revisiting knowledge, the tester could repeat the experiment where he/she needed to build a simple superset of a social media application. The second time we performed this with technologies chosen by the developer, we noticed that the result of using the tool was much more fluent than the first time. However, some interesting feedback was given. The flaze list command should show more information than just the name of the frameworks and libraries. Instead of tags, we could use categories or create a search mechanism to find the correct framework or library quickly. Another problem we saw was that the tester would try to use a command he/she did not fully remember without using the help option. This resulted in feedback that some commands could ask for permission or verification to run a certain command to verify the intention was correct. However, this could lead to an extra step for each command. We also noticed that the developer liked that the installed frameworks were automatically cleaned up so that the tester did not have to do this themselves. However, the tester was confused by some things that the toolset did not specifically handle well. The tester always tried to run commands from the root directory, which sometimes resulted in incorrect behavior. Another related issue was that when a use command is run from the root directory, it will initialize for all the detected packages in that directory. The packages that were not supported gave a message with a red ‘x’ symbol. This confused the user. It meant that the library was not supported by the framework and, thus, not installed. The tester would like a warning here instead. Overall, the toolset did not really fail badly when the user performed this test the second time. This is probably because of the previous improvements. Another interesting thing the developer said is, ‘I still haven’t written any code.’. This also suggests that the tool could be used for many of the steps commonly done manually.

In general, we saw a better end result for the application. Note that many bugs were fixed, and feedback and missing tools were implemented. To view the end result, you can look at the appendix for the source code written for this experiment. In the end, we asked a final set of questions to conclude about the overall experience of using the tools and the advantages and disadvantages are when using this tool. The first observation by the user was that he/she was confused when using the introspect tool. He/she did not know if he/she could introspect the API without any additional steps. It seemed like more needed to be done. When looking at the help command, a lot was clarified. To problem was not looking at the help command. A thing noticed by the tester is that the tester would follow the same conventions. This could be a problem for people with different conventions using the same frameworks and libraries. The tester also said that the create and generate tools were the most useful. However, he/she later said that all tools were really useful. When we discussed possible missing features, the tester again said no confirmation was asked before some commands. Multiple follow-up questions could also be asked to allow for even better end results. This could affect the simplicity of the tool. An important answer provided by the tester is that he/she would think it would take him/her a lot longer if he/she implemented everything himself/herself. Flaze provides a good baseline for setting up your project and configuring your stack more quickly. It was also mentioned again that the toolset could be very useful for learning or working with other languages in which you do not have much experience. Flaze will provide a way to find my way around every supported framework quickly. The tester said that the tool would not be ideal for complete beginners.

When asked about the developer experience, the tester mentioned that flaze provides a better developer experience if we look at efficiency. The same structure is used, and we can quickly extend our application with other generators. However, if a user has different conventions, this could lead to a worse developer experience. The developer would not find where things are generated, which would result in searching the codebase and maybe even code duplication. On the other hand, when used in larger teams, the same structure is consistent across developers, which could be a benefit.

In the last set of questions, we asked about the developer experience of each tool individually. This resulted in a general overview of the impact of the different tools. The init tool did not provide a major benefit. This could even be integrated into the create tool. The create tool was easy to use and allowed the developer to install frameworks without knowledge of their basic configuration. The `use` tool has a similar impact as the create tool. The developer can simply install different libraries without the specific knowledge required to configure these. The `generate` tool was really useful. It could generate components, pages, and resources following consistent conventions. And because for simple applications, these three commands are common, these cover a large part of the code the tester wanted to generate. However, this may change when building different applications. The introspect tool was also very useful. However, this resulted in a lot less repetitive work with recent developments such as Github Copilot. This is not as big a problem as before. Using the tooling provided, the types are extracted using the TypeScript Compiler API, which could result in more correct results than AI. The tester did not answer this question because the watch tool was not used during the experiment. The studio command was also very useful. However, it was limited by what was possible. The Postman-like interface was great. The features it provides could be improved. It was also useful to take a quick look at the database, but this interface also needed additional functionality, such as updating rows, searching, handling large data, and more.

To validate the results of the first test user. We conducted a second validation beta test. We would perform the same experiment with a different developer in this beta test. The results obtained from this test allow us to verify the feedback provided by the first beta test. This validation beta test was conducted to ensure we did not forget important problems with the user experience that could lead to a worse developer experience. In these next few paragraphs, we will discuss the results obtained from the validation test. The validation test starts with an introduction to the tool by reviewing the documentation. This is done to get a general understanding of what the tool is capable of and helps the tester to use the help feature of flaze. The next step is the creation of a simple social media web application. The last step is a set of questions to retrieve final feedback on the overall product that has been created.

We start off by discussing the first part of the experiment. This is where we ask the tester what he/she remembered after reading the documentation. These results were not great. The tester did not remember the exact meaning of most of the commands without the documentation being present. This required us to review the documentation again before the questions could be answered. The results were much better the second time, although still not perfect. The questions about what the commands do were answered correctly for the create, use, and studio tool. The questions that were answered partially correctly were about the generate, introspect, and list tools. The tester could use the build help command and the documentation for the coding task.

During the coding task, we captured feedback from the tester and problems that occurred during the test. We will discuss the results here. The first feedback the tester provided was that he did not know much about most of the provided frameworks. Ultimately, this did not have a noticeable impact on the final source code. When using the tools, the first feedback was that the parameter `name` was unclear when using the `create` command. This became clear when the tester used the `--help` parameter. Overall, the help command provided an easy way to ensure what commands could be used and what required parameters and options. Another thing the developer tried to do was read the `README.md` to get a better understanding of the framework. The default framework `README.md` was provided. The tester suggested that this could be useful if it had more information about the framework and what flaze provided. Another problem the tester mentioned was that `flaze` was difficult to type on the keyboard and that a better name could be provided. The list command was useful. The tester did not know all the frameworks and libraries. The tester also had problems with the distinction we made between frameworks and libraries. This was unclear to the tester. This resulted in he/she being confused about how to initialize libraries correctly. He/she was able to figure it out thanks to the help command and was able to use it correctly afterward. He/she suggested that the `list` command should include (create and use) to clarify what could be initialized by what. Another idea was that because some commands show usage information, this information should be added to a usage markdown file. This would allow the tester to quickly go back and look at the usage of certain libraries. Another unclear thing was that the tester used the resource's name when migrating. However, this is possible; it was unclear whether this would be the name of the migration. A comment about the flaze studio API tool was that it should be made clear that you need to enter JSON or another format if that is being used. Overall, the application's implementation was great, except for minor mistakes and unclarities. The tester completed the project and was a lot more confident using the tool after he/she had used all commands. The next paragraph will discuss more findings and results validating the first beta test.

The first feedback provided by the tester was about the problems with the toolset. He/She said that because of the lack of knowledge about the frameworks, the tester found it more complicated to build the web application. He/She also had difficulty understanding the introspect tool. However, he/she did find the tool useful. It was also mentioned that the other commands were clear and that the help option allowed him/her to clear up uncertainties. An interesting comment was that he/she would not know where to begin without the tool. When asked about what tools were most helpful, he/she said that combining the different tools provided a good experience and that the commands on their own should probably not be that useful. The functionality the tester was missing were better help commands. This flaze run command would run the correct npm script, more safe commands, a history log with more information about what happened, an undo feature, and more insights into what is happening. The tester was impressed by how quickly he/she could set up a working web application without extensive knowledge about the frameworks and libraries. Lastly, we asked about the developer experience of the overall toolset and the individual commands. He/She said that if the tools were completely finished, this toolset could really speed up the development process of building web applications with different frameworks and libraries. Not only would it be faster, but also a lot easier. The biggest struggle would be correctly setting up the frameworks and libraries; writing them could take him/her a lot longer than generating a boilerplate code. Some things happened that he/she couldn't see, like CORS. He/she mentioned that he/she didn't have these problems that are usually there when developing web applications. A problem, however, could be that developers get used to writing less code which could be a problem when changes need to be made to generated code. As mentioned before, the tester did like the toolset as a complete package. However, we also discussed each command individually. For all the initialize tools, he mentioned that they were very useful and that even if they did not do a lot, he/she did not have to worry about correctly setting up a project. The `generate` tool allowed the tester to not think about the framework but about the functionality of the application he/she was using. The introspect was very useful and ensured the tester did not have to copy code from the backend manually. However, at first, he/she found the name and the help information unclear. He/she did also not use the watch command, so that was not discussed. The flaze studio command was very useful. The tester really liked the Postman-like interface that was aware of the parameters and bodies. This resulted in a fluent experience. He/she also liked that he/she did not have to type any endpoints to test them with Postman. The database was said to be not as useful because it seemed like a simple get query. However, if the flaze studio tools had more features, they would be more useful. A missing thing was the set of supported frameworks and libraries. A bigger list to choose from would be very beneficial. However, the frameworks and libraries that are very popular would provide a good starting point.

To conclude, we can say that the overall experience of using the tool had a positive impact on the efficiency of creating basic web applications. The toolset was useful as a whole, and the distinction between the different commands was clear for all commands except the create and use commands. Both testers mentioned that they loved how quickly they could build applications and were impressed by how quickly they could build the demo application. The overall feedback that was provided could result in a more stable tool and mentioned improvements in documentation, ‘help’ information, feedforward, and more clarity about what introspection does. This feedback could help us build an even more stable product that could provide a way of creating applications for the web in a more feature-driven way.

Chapter 6

Conclusion

This chapter will discuss the conclusions found when working on this thesis. First, the main objective of this thesis will be revisited. Using the research questions defined in the introduction, the findings for these questions will be addressed here. Finally, a summary of the findings and conclusions will be presented to the reader.

6.1 Conclusion

This thesis investigates potential methods for enhancing the developer experience in creating web applications. We began by thoroughly exploring the existing literature to identify the contemporary technologies utilized in constructing modern web applications. With a grasp of the available tools, we formulated a comprehensive lifecycle encompassing the necessary steps for developing modern web applications. Subsequently, we worked on devising solutions for the challenges that arise throughout this lifecycle. Our efforts were primarily guided by the research questions outlined in the introduction. This approach allowed us to simultaneously address emerging issues and remain aligned with the research questions. The following section will dive into the research questions and provide a general conclusion.

6.1.1 Research Questions

RQ1: What tools can we create to improve the DX when building web applications?

This question can be answered by looking at the `flaze` toolset. This toolset provides a set of tools that help improve the DX when building applications for the web. We were able to define these tools by looking at the development lifecycle. For every step in the lifecycle, we tried to identify a corresponding tool or tools that could improve the steps needed to be performed by the developer. The tools include initialization, generate, introspect, and interaction tools. What the individual tools to is discussed in the chapter about Tooling.

RQ2: What libraries can we create to improve the DX when building web applications?

Because of time constraints and focus on the `flaze` toolset, we could not answer this question from the research that was conducted. However, we could look further into libraries that solve the need to write boilerplate code or help with code sharing. We tried to solve these aspects with the `flaze` toolset and were confirmed to be bottlenecks of modern web development by the beta testers.

RQ3: How can we reduce the amount of boilerplate code we write?

We tried to reduce the amount of boilerplate the user had to write by generating the code for the user. This resulted in the user only having to run a command instead of writing the code itself. An important part of generating code is context awareness. Generated code can depend on the installed frameworks and libraries or specific configurations. This provided a seamless integration with the existing code. Another aspect we looked into was not having to write boilerplate code if we could learn what code the user would want to write based on existing code. This process is called introspection. These techniques allowed us to automate the process of writing boilerplate code, resulting in abstracting away the redundant process of writing boilerplate code.

RQ4: What types of patterns can we abstract?

The patterns we can abstract are similar to what the generate and introspect tools can do for the developer. These patterns are the generation of components, pages, resources, API connectors, Database connectors, and more. These patterns are commonly recurring and can be abstracted into generic boilerplate code. The abstracted boilerplate code can be used to generate the correct code depending on the developer's needs. However, there are many more patterns that we could not abstract or that would be more difficult to abstract. The process of identifying all of these patterns is difficult and an important next step.

RQ5: Can we generate boilerplate code for these abstracted patterns?

Yes, because we can abstract the boilerplate code into generic templates, we can abstract the complexity away and only require the developer to fill in the missing values of the generic templates. However, not all patterns can be easily abstracted. Sometimes, a lot of initial data is still needed. Not everything can be generated from a single command. Sometimes, follow-up information is needed to create a complete generic boilerplate. Abstraction is great when we have a lot of static boilerplate with a small number of parameters. The problem becomes more visible when the number of parameters grows. Then it becomes more complex to generate a boilerplate, and maybe more easy to write the code yourself. It will already be beneficial to generate a boilerplate for the top 80% common problems. However, this may be different for different developers.

RQ6: What are the problems with generating boilerplate code?

We either need a lot of information or don't have enough information to generate complex boilerplate code. This means that the simpler it is to generate boilerplate code, the simpler the boilerplate code will be. If we want more complex boilerplate code generated, we may need to pass more parameters, and the command to generate boilerplate becomes more complex. We could try to create an adaptive approach where we incrementally increase the generated code. The problem here is the complexity of the implementation.

Another problem with generating boilerplate code is the fact that the boilerplate code needs to be compatible with the code that is already there. This can be unpredictable. Lastly, the number of things you can generate boilerplate code for is probably very large. This results in deciding what problems are more important to solve than others. Where should we provide boilerplate code generation? This can be a difficult question to answer.

RQ7: What techniques can be used to build web applications efficiently?

Our main improvement in this thesis was the ability to generate code. This can be done by generating generic boilerplate code for certain patterns or using introspection to generate more complex boilerplate code. The benefit of introspection is that this could allow for more complex code generation. This is because introspection provides an intermediate format. This format can be seen as a large number of parameters. This allows us to generate more complex boilerplate code because the developer does not need to write all the code himself/herself. This could have a massive impact on the efficiency of building web applications. The beta testers also confirmed this.

RQ8: What techniques can be used to build web applications consistently?

To ensure consistency, we may look at context-aware code generation. This concept allows us to use the same process to generate code even when the underlying technologies are different. This allows the developer to perform a set of consistent steps that result in similar results depending on the framework the user is using. This allows us to transfer our skills using the toolset to learn about new tools and technologies. Another benefit of using the toolset is that we provided both consistency in the commands and consistency in using the correct conventions for a given framework or library.

6.1.2 Discoveries and Observations

The `flaze` toolset helps developers quickly set up a project with the necessary frameworks and libraries. After initialization, we can generate code based on the user's needs or based on already written code. This workflow allows us to set up and build applications for the web seamlessly. The code that is generated is context-aware. This helps you focus on the product you want to create rather than focusing on learning how to install a certain framework or library or write a certain piece of code. The generated code can also be modified by the developer afterward. This results in a system that allows modular iterative development.

We can make the development process more efficient when using the `flaze` toolset. This can be done because we can use a tool provided by the toolset for each step in the process that improves efficiency. This allows the developer to save time setting up, generating, and introspecting code. This can also make sure that the user does write minimal duplicate code. And because the process of creating a basic application is so efficient, we can quickly prototype a simple web application without the need to write most of the code. When combined with flaze studio, we can quickly test our API and inspect our database without writing additional code. This allows us to provide our application with seeded data without having to have a fully working version of the client application.

Overall, the toolset created provides a seamless DX when prototyping and building applications for the web. Although not everything is handled by the toolset, the toolset allows us to quickly develop our application without needing specific knowledge about the underlying technologies.

Chapter 7

Future Work

This final chapter will discuss possible future work. Because of the wide scope of this thesis, we were not able to implement all the possible findings and solve all limitations that were identified in this thesis. This section will provide the reader with an overview of what improvements can be made to ‘Improving the Developer Experience when Building Web Applications’.

7.1 Future Work

An important limitation addressed by us and by the beta testers was that the set of supported frameworks and libraries was limited. To further benefit from the toolset and the concept itself, a larger set of supported frameworks and libraries would vastly improve the overall experience. It remains important that the supported libraries and frameworks are relevant and that they can be implemented in a straightforward and modular manner.

A downside of generating code and mostly injecting code is that it becomes more difficult to update or regenerate this code. For example, when an import is added to an array when we generate the same code again, we need to make sure that the import is not added to the array again but rather is updated. This can be important to maintain a stable and usable codebase. A better mechanism to handle updating generated boilerplate code is needed. This can solve the problem, that code generation can be run not only once but in an iterative manner. Another benefit is that we can update generated code after that the developer has updated the generated code. The flow of generating code, customizing code, and regenerating unmodified code becomes a seamless process that provides stable, customizable, and runnable code. This could also improve conflict resolution.

Conflict resolution is another problem that occurs when different code generators are used together with user-written code. These conflicts can still happen with the current system. Because we have no say in what code the user is going to write, we need a good system to try and find code that is relevant for us to base ourselves on. However, in the end, a system for conflict resolution or a better system to interact with code is needed.

When generating or introspecting code, we sometimes need to modify, add, or interpret existing code. This can be done by using parsing these different types of files and using this knowledge to perform certain actions. We can do some of these things by using a naive parsing approach, but this can sometimes result in problems. A better way to solve this is by implementing useful APIs that help you with parsing common files. These files include TypeScript, TypeScript JSX, Env, Prisma, Vue, Dockerfile, and more. By providing advanced parsing utilities to the developers implementing `flaze` for their framework or library, the end result will be a more advanced and stable toolset. These APIs can also help you with the problem above. Because better parsing, means better insight into what to update in the code, these APIs provide the basis for understanding and working with existing code.

If we look at TypeScript specifically, we need a better method for inferring and resolving types defined by the user. If we are able to do this, we can make sure that every type we introspect is the correct type. This will allow us to ensure that the types are always correct. This allows us to do interesting things, like create a type-aware Postman. For now, when we are not sure about the type. We provide the system with any type. This can work for now, but it leaves open a set of cases, where we don't have full type safety. A better way to 100% correctly infer types or even convert the type of an intermediate notation can be useful to provide the user with an unmatched developer experience.

Because of the large scope of web development code. It is important to look deeper into what can be generated. For this thesis, we mostly looked at generating components, pages, and resources. This does not mean that these are the only things that can be generated. A deeper dive into what can be generated is needed. This information can be used to either improve the amount of boilerplate needed for the library or the framework or to generate these commonly repeated processes. When we better understand the needs of the developer, we better understand what needs to be generated to provide a more efficient workflow. This will allow for better `generate` and `watch` commands to be included in the `flaze` toolset.

The `flaze studio` command provides us with a visual interface for interacting with existing code without the need for the developer to set up anything. In this thesis, we were only able to create visual interfaces for environment variables, for testing the API, and for testing the database. A lot more can be possible using the same techniques. It would be interesting to see what other tools can be created that use the source code as their source of truth. By abstracting these tools from the libraries themselves, we could provide interesting tooling for all libraries that implement a simple adaptor or that use a community adaptor. It would also be a good thing to improve the UI of `flaze studio` itself as a tool. This could refine the overall user experience when using `flaze studio`. Lastly, existing `flaze studio` tools can be improved to provide a wider range of functionality.

A lot more can be done to improve the toolset and 'Improving the Developer Experience when Building Web Applications' in general. However, we are not able to put every possible limitation here. To read more about other limitations, you can read the chapter about the Implementation of the toolset. Here we discuss many of the limitations that we noticed when implementing the toolset. You can also look at the chapter about the Evaluation that we conducted. Here you can find a lot of information about the results of the internal testing and the beta tests.

Bibliography

[1] Angular. https://angular.io/. (Accessed on 03/18/2023).

[2] Ant design - the world's second most popular react ui framework. https://ant.design/. (Accessed on 03/24/2023).

[3] Auth.js. https://authjs.dev/. (Accessed on 08/13/2023).

[4] Blade templates - laravel - the php framework for web artisans. https://laravel.com/docs/9.x/blade. (Accessed on 03/04/2023).

[5] Bootstrap · the most popular html, css, and js library in the world. https://getbootstrap.com/. (Accessed on 03/24/2023).

[6] Bulma: Free, open source, and modern css framework based on flexbox. https://bulma.io/. (Accessed on 03/24/2023).

[7] Canvas api - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. (Accessed on 08/13/2023).

[8] Css: Cascading style sheets — mdn. https://developer.mozilla.org/en-US/docs/Web/CSS. (Accessed on 03/04/2023).

[9] css-modules/css-modules: Documentation about css-modules. https://github.com/css-modules/css-modules. (Accessed on 04/07/2023).

[10] cva. https://cva.style/docs. (Accessed on 04/07/2023).

[11] Docker: Accelerated container application development. https://www.docker.com/. (Accessed on 08/13/2023).

[12] Document object model (dom) - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model. (Accessed on 08/13/2023).

[13] Drizzleorm - next gen typescript orm. https://orm.drizzle.team/. (Accessed on 08/13/2023).

[14] Ecmascript® 2023 language specification. https://tc39.es/ecma262/. (Accessed on 03/04/2023).

[15] Emotion – introduction. https://emotion.sh/docs/introduction. (Accessed on 04/07/2023).

[16] Express - node.js web application framework. https://expressjs.com/. (Accessed on 08/13/2023).

[17] Framework guides - tailwind css. https://tailwindcss.com/docs/installation/framework-guides. (Accessed on 07/24/2023).

[18] Getting started — less.js. https://lesscss.org/. (Accessed on 03/18/2023).

[19] Getting started – pug. https://pugjs.org/api/getting-started.html. (Accessed on 03/04/2023).

[20] grpc. `https://grpc.io/`. (Accessed on 08/13/2023).

[21] Headless ui - unstyled, fully accessible ui components. `https://headlessui.com/`. (Accessed on 04/07/2023).

[22] Home - openapi initiative. `https://www.openapis.org/`. (Accessed on 08/13/2023).

[23] Home – fets. `https://the-guild.dev/openapi/fets`. (Accessed on 08/13/2023).

[24] Html: Hypertext markup language — mdn. `https://developer.mozilla.org/en-US/docs/Web/HTML`. (Accessed on 03/04/2023).

[25] Introducing jsx – react. `https://reactjs.org/docs/introducing-jsx.html`. (Accessed on 03/04/2023).

[26] Javascript — mdn. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. (Accessed on 03/04/2023).

[27] Jss. `https://cssinjs.org/?v=v10.10.0`. (Accessed on 04/07/2023).

[28] Just in time compilation explained. `https://www.freecodecamp.org/news/just-in-time-compilation-explained`. (Accessed on 03/18/2023).

[29] Laravel - the php framework for web artisans. `https://laravel.com/`. (Accessed on 08/13/2023).

[30] Mantine. `https://mantine.dev/`. (Accessed on 03/24/2023).

[31] Master css - a virtual css language with enhanced syntax. `https://css.master.co/`. (Accessed on 04/07/2023).

[32] Mui: The react component library you always wanted. `https://mui.com/`. (Accessed on 03/24/2023).

[33] Nestjs - a progressive node.js framework. `https://nestjs.com/`. (Accessed on 08/13/2023).

[34] Node.js. `https://nodejs.org/en`. (Accessed on 08/13/2023).

[35] Open props: sub-atomic styles. `https://open-props.style/`. (Accessed on 04/07/2023).

[36] Passport.js. `https://www.passportjs.org/`. (Accessed on 08/13/2023).

[37] Postcss - a tool for transforming css with javascript. `https://postcss.org/`. (Accessed on 03/18/2023).

[38] Primitives – radix ui. `https://www.radix-ui.com/`. (Accessed on 04/07/2023).

[39] Prisma — next-generation orm for node.js & typescript. `https://www.prisma.io/`. (Accessed on 08/13/2023).

[40] React aria. `https://react-spectrum.adobe.com/react-aria/`. (Accessed on 04/07/2023).

[41] React – the library for web and native user interfaces. `https://react.dev/`. (Accessed on 03/18/2023).

[42] Sass: Syntactically awesome style sheets. `https://sass-lang.com/`. (Accessed on 03/18/2023).

[43] Solidjs · reactive javascript library. `https://www.solidjs.com/`. (Accessed on 03/18/2023).

[44] Stack overflow developer survey 2023. `https://survey.stackoverflow.co/2023/`. (Accessed on 08/04/2023).

[45] The state of css 2022: Other tools. https://2022.stateofcss.com/en-US/other-tools/. (Accessed on 03/18/2023).

[46] Stitches — css-in-js with near-zero runtime. https://stitches.dev/. (Accessed on 04/07/2023).

[47] styled-components. https://styled-components.com/. (Accessed on 04/07/2023).

[48] Svelte • cybernetically enhanced web apps. https://svelte.dev/. (Accessed on 03/18/2023).

[49] Tachyons - css toolkit. https://tachyons.io/. (Accessed on 04/07/2023).

[50] Tailwind css - rapidly build modern websites without ever leaving your html. https://tailwindcss.com/. (Accessed on 04/07/2023).

[51] trpc - move fast and break nothing. end-to-end typesafe apis made easy. — trpc. https://trpc.io/. (Accessed on 08/13/2023).

[52] Turbo. https://turbo.build/. (Accessed on 08/13/2023).

[53] Typescript: Javascript with syntax for types. https://www.typescriptlang.org/. (Accessed on 03/04/2023).

[54] Using express middleware. https://expressjs.com/en/guide/using-middleware.html. (Accessed on 08/13/2023).

[55] Using template engines with express. https://expressjs.com/en/guide/using-template-engines.html. (Accessed on 03/04/2023).

[56] Using the compiler api · microsoft/typescript wiki. https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API. (Accessed on 08/13/2023).

[57] vercel/styled-jsx: Full css support for jsx without compromises. https://github.com/vercel/styled-jsx. (Accessed on 04/07/2023).

[58] Vue.js - the progressive javascript framework — vue.js. https://vuejs.org/. (Accessed on 03/18/2023).

[59] Web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API. (Accessed on 03/04/2023).

[60] Web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API. (Accessed on 08/22/2023).

[61] Web components — mdn. https://developer.mozilla.org/en-US/docs/Web/Web_Components. (Accessed on 03/04/2023).

[62] Web workers api - web apis — mdn. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. (Accessed on 08/13/2023).

[63] Webassembly — mdn. https://developer.mozilla.org/en-US/docs/WebAssembly. (Accessed on 03/04/2023).

[64] Writing markup with jsx – react. https://react.dev/learn/writing-markup-with-jsx. (Accessed on 08/03/2023).