Bachelor's thesis

Information and Communications Technology

2025

Trung Nguyen

Development of Full-Stack Web Application with Bun and TypeScript

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2025 | 83 pages

Author: Trung Nguyen

Development of Full-Stack Web Application with Bun and TypeScript

The first objective of this thesis is to design and deploy a full-stack web application with a secure, scalable back-end and a user-friendly interface. While modern web applications can be built using diverse technologies, TypeScript provides a unified ecosystem that enables solo developers and small teams to remain competitive. Despite Node.js and React being the dominant frameworks in the field, emerging alternatives such as Bun and Svelte challenge their performance benchmarks. Understanding these new technologies is crucial for developers seeking optimal solutions for full-stack projects.

Another objective is to evaluate Bun and Svelte’s production readiness, a job-search web application was designed and deployed as a real-world case study. The application followed MVC architecture, leveraging Bun and Hono for the back-end, and Svelte, Axios, and Vite for the front-end. PostgreSQL served as the database, with Prisma facilitating data modeling and queries.

The findings of this study demonstrate that Bun delivers notable performance improvement in specific scenarios, though its ecosystem maturity remains a limitation. Svelte proves highly effective for front-end development, offering a streamlined approach to building reactive interfaces. These results serve as a valuable reference for developers in deciding whether to use Bun and Svelte.

Keywords:

Full-stack Application, TypeScript, BunJS, HTML, CSS, Svelte.

Contents

List of abbreviations **7**

1 Introduction **9**

2 Objectives, Process, Expectation **11**

2.1 Back-end Development 12

2.2 Database 13

2.3 Front-end Development 13

2.4 MVC Architecture 14

2.5 Testing and Security 15

2.6 Expectations 16

3 Programming Language **17**

3.1 Introduction 17

3.2 The advantages of TypeScript 18

3.3 The disadvantages of TypeScript 22

3.4 Software Development Principles 23

3.4.1 SOLID 24

3.4.2 DRY 25

4 Back-end Development **28**

4.1 BunJS 28

4.2 Hono 33

4.3 Linting and Formatting 35

4.4 Authorization and Authentication. 36

4.5 REST API 38

4.6 Project Structure 42

4.7 Validation 43

4.8 Unit testing and Integration testing 44

4.9 Security 47

5 Data management **49**

5.1 In-Memory (RAM) temporary database. 50
5.2 PostgreSQL 51
5.3 Docker 52
5.4 CRUD Structured Query Language (SQL) operations 55
5.5 Object-relational Mappings (ORM) 57

6 Front-end Development 63
6.1 Svelte 63
6.2 Vite 65
6.3 Axios 66
6.4 Authentication Component 68
6.5 Chat Component 72

7 Conclusion 75

References 77
Declaration of AI usage in Thesis Preparation 82

Codes

Code 1. Implementation of type and interface, and Pick, Omit in TS Development. 19
Code 2. An implementation of memory_app with TypeScript. 21
Code 3. Example of Reusable Function in TS. 26
Code 4. Starting a Bun server on localhost:3000. (Source: Bun.sh) 32
Code 5. How hot reload is set in Bun. 32
Code 6. Basics of starting a server with Hono which includes logger and timing. 34
Code 7. Working server with timing and logger. 34
Code 8. Install hono/eslint. 35
Code 9. Install Prettier. 35

Code 10. Implementation of Authentication and Authorization with JWT 37
Code 11. A functional MVP working chat app that follows RESTful API Design. 40
Code 12. Using bun to add 'zod' package into the project. 44
Code 13. Authentication test cases on Bun. 46
Code 14. Chat and Message test with Bun. 47
Code 15. Implementation of In-memory methods. 50
Code 16. SQL database schema for the application. 52
Code 17. Implementation of CRUD for users in SQL. 55
Code 18. Implementation of CRUD for Chat in SQL 56
Code 19. Implementation of CRUD for message in SQL. 57
Code 20. class User. 60
Code 21. class Chat 61
Code 22. class Message 62
Code 23. Register Component (HTML) 68
Code 24. Register function. 68
Code 25. Login Component 69
Code 26. Login function. 69
Code 27. Authentication store. 71
Code 28. create Chat function. 72
Code 29. Show Chat List function. 73
Code 30. get Chat Details function. 74

Figures

Figure 1. Benchmark result of express server. (Source: Betterstack.com) 31
Figure 2. Project Structure on Back-end. 42
Figure 3. Prisma Schema for ORM. 59
Figure 4. Svelte’s transpiler generates an AST from a component (Source: Miikka, O. (2021). 64

Pictures

Picture 1. MVC Architecture Diagram. (Source: Mozilla.org) 15
Picture 2. Bun official logo. (Source: Bun.sh) 28
Picture 3. Result of API testing with Bun. 45
Picture 4. A running PostgreSQL database with Docker Compose. 54

List of abbreviations

AI	Artificial Intelligence
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AST	Abstract Syntax Tree
AWS	Amazon Web Services
CSS	Cascading Style Sheets
CRUD	Create, Read, Update, Delete
DRY	Don't Repeat Yourself
DOM	Document Object Model
EC2	Amazon Elastic Compute Cloud
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
HMR	Hot Module Replacement
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
LLM	Large Language Models
NPX	Node Package Execute
NPM	Node Package Manager

MVP	Minimum Viable Product
MVCC	Multi-Version Concurrency Control
OOP	Object-Oriented Programming
OS	Operating System
REST	Representational State Transfer
RDBMS	Relational Database Management System
SDLC	Software Development Life Cycle
SE	Software Engineering
SOLID	Software Development Principle
SPA	Single Page Application
SQL	Structured Query Language
SSR	Server-Side Rendering
TDD	Test Driven Development
TS	TypeScript
UI/UX	User interface/User experiences
VAR	Variable
WAPP	Web Application
XML	Extensible Markup Language
YAML	Yet another markup language

1 Introduction

The pursuit of stable and suitable employment is a fundamental activity of human life in the modern world since it is one of the main means humans can gather resources and open various possibilities, especially within the fast-moving and competitive technical field. There are numerous platforms that allow job seekers and recruiters to interact and achieve their goals. In terms of means, there are several different media to post and look for jobs, for example, in person, via events, in newspapers, on social media and on job sites. According to ahilsenbeck (2024) from INSPYR Solutions, with the rise of Large Language Models, one could not ignore the coverage of online tools such as LLM API and their searching power. There is an urgent need for efficient, effective, and innovative job application platforms that combine the simplicity of an interface.

To fulfill this need and given the strategy and situation of the author of the thesis, a Web Application (WAPP) is the most suitable strategy for this project due to its low cost in development and fast deployment. This is the best method for testing the market before investing too much in the services. Since the project will be carried out by one person, using one programming language for many purposes will be the best choice. For web front-end technology, according to Watson, D. (2024), nothing could beat JavaScript/TypeScript (JS/TS) in terms of coverage, support and performance. With this thought in mind, it is strategic to utilize JS back-end technology to streamline the development process. While established technologies such as Node.js have long dominated back-end development, emerging alternatives such as BunJS offer the potential for enhanced performance and streamlined workflows. BunJS promises and efficiency, is lightweight and has no hidden flow controls. Specifically, this thesis examines whether BunJS can provide a viable and superior alternative for building a modern, high-performance, modular and scalable chat app for job searching solutions.

This thesis presents a technical solution to the problem above, structured across seven chapters that guide the reader through the project's conception,

development, and future directions. This consists of 7 chapters with Chapter 1 introducing the core problem addressed in this work and outlining the proposed technical solution, establishing the motivation and significance of the research. This is followed by Chapter 2, which defines the project's objectives, work plan, and thought process, detailing the approach taken to achieve the intended outcomes along with key expectations. Next, Chapter 3 examines the primary programming language chosen for the project, discussing its strengths, limitations, and the rationale behind its selection. Chapter 4 presents the proposed back-end architecture, covering framework selection, development stages, and testing strategies to ensure robustness and scalability. Chapter 5 describes the Database Design and Implementation from an in-app prototype to a production-ready independent database, highlighting key design decisions and optimizations. Additionally, Chapter 6 outlines the proposed work on front-end development, focusing on usability, design choices, and integration with the back-end system. Finally, Chapter 7 summarizes achievements, key findings, and lessons learned, while also proposing next steps for further development and potential enhancements.

2 Objectives, Process, Expectation

This thesis aims to study the development of a basic chat app for job searching purposes website using BunJS and Svelte, emphasizing back-end development with minimal front-end side development, and incorporating a lightweight but performance database system. The goal of this thesis is to provide a guide for building secure, user-friendly, and high-performance platforms, addressing all stages of development and the reasoning behind them. In addition, this thesis also aims to demonstrate the MVC development pattern to the readers. By examining the practical application in this context, the second objective is to assert technical conclusions about new experimental technologies as BunJS and Svelte and evaluate their potential to be sufficient as a main development for the long term. Furthermore, the development strategy can be tested to see if it is versatile enough to be implemented on a variety of platforms of the author's choice. The structure of this thesis will begin with a detailed exploration of the design and implementation of the BunJS and Svelte-based platform, followed by the reasons behind the technology choice and the way of implementation. The author then evaluates its performance and usability, concluding with an analysis of the findings and potential future directions.

The development of a full-stack job searching web application involves a few steps, each step addresses different moving parts of the application and requires specific technical considerations. In the scope of this thesis, the author focuses more on implementing a safe and robust back-end system to see if the BunJS ecosystem is production-ready or not. For the front-end, Svelte is utilized with minimal functions.

To effectively evaluate the readiness of new technology and provide the most practical development process, the author wants to approach the problem uniquely. The author was living in Austria at the time of writing this thesis, joining several career fairs, realizing that most people complain about filling to many forms, which makes the quality of applications lower. According to the European Labour Authority (2021), the job application process, especially with enormous

job postings online, requires the person who is applying for the job must go through many steps via third-party job agencies to apply for a job. The author wants to approach this problem from a unique perspective, which treats the job application process as a normal chatting conversation, where job seekers can chat with an API that connects to LLM to see if they have any interesting open positions. If they see an interesting position, they can start a new chat with the hiring manager and open with a motivational letter and with an attachment as a CV. The hiring manager will have all AI agents start filtering the potential employees. When the hiring manager is interested in the applicant enough, they can start a conversation. On the other hand, all hiring managers can go through all interesting candidates, and if they see a good candidate, they can start a conversation right away. By treating it as a conversation, both can drop the formality in the communication to focus more on exchanging information without being flooded by all types of jargon. This approach makes the process more semantic for both parties.

2.1 Back-end Development

The next phase, back-end development, centers on creating robust server-side applications and well-structured APIs to manage the core functionalities of the chat app, including creating new chats, browsing all chats, conversation tracking, messaging, and user authentication. According to the Amazon Web Services Guide (2024), security and scalability are two primary considerations when constructing the back-end server, as the platform must accommodate potentially high volumes of concurrent users and handle sensitive information. Developing efficient APIs facilitates seamless communication between the front-end and back-end, ensuring a reliable and swift response to user actions while preserving data integrity and secure access.

2.2 Database

Database integration is the next phase that demands detailed planning to establish a reliable data model. According to Raymond Blum and Rhandeev Singh from SRE Google, a reliable data model must support the storage, retrieval, and organization of potentially vast amounts of data over time, including job conversations, user profiles, message records and application records. A concrete database design is important not only for the development process but also for the application's efficiency, as it directly impacts the speed and accuracy of searches, filtering, and data retrieval processes. Changing the data model in the middle of the development process has proven costly and could potentially reset the whole development phase. This phase also involves ensuring that data relationships are structured to minimize redundancy and optimize data handling, which becomes vital as the job board grows in user base and content. Finally, the database should be an independent entity, so it allows the connection of different servers.

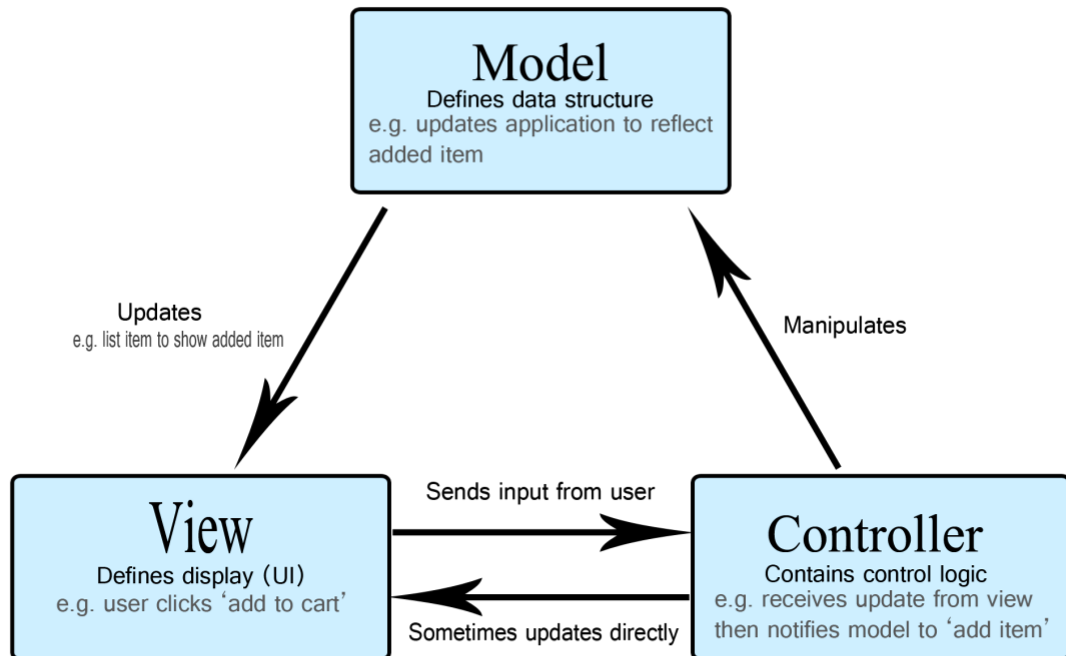
2.3 Front-end Development

Front-end development is crucial because it is responsible for creating an intuitive, responsive, and visually appealing user interface. This step includes designing features that define user interaction, such as easy navigation between components, dynamic job conversation views, and clear prompts for both job seekers and employers. Emphasis is placed on delivering a responsive design that ensures a consistent experience across various devices, allowing users to interact seamlessly whether on desktop or mobile. As the developers often ignore the importance of the user experience of the website, it could make the difference that decides whether the website is successful or not, according to the research from (Jongmans et al., 2022).

2.4 MVC Architecture

According to the findings of (MDN Web Docs, 2023), the Model-View-Controller (MVC) pattern is a widely used software design approach for structuring user interfaces, data management, and application logic. By separating an application into three interconnected components: Model, View, and Controller, it enforces a clear division between business logic and presentation, promoting modularity and ease of maintenance. This principle of separation of concerns allows for more efficient collaboration among developers and simplifies long-term code upkeep. Over time, several variations of MVC have emerged, including Model-View-View-Model (MVVM), Model-View-Presenter (MVP), and Model-View-Whatever (MVW), each adapting the core concepts to different architectural needs.

In the MVC pattern (Picture 1), the Model represents the application's data and business rules, handling tasks such as database interactions and computations. The View is responsible for rendering the user interface, displaying data to the user, and receiving input without processing it directly. The Controller acts as an intermediary, managing user requests, updating the Model, and coordinating changes in the View. This structured separation ensures that modifications to one component, such as the user interface, have minimal impact on the underlying logic, enhancing scalability and maintainability.



Picture 1. MVC Architecture Diagram. (Source: Mozilla.org)

2.5 Testing and Security

This thesis explores the fundamental principles, tools, and technologies required for developing a full-stack job chat application. A thorough review of academic literature in Bun.sh Testing Practice (2025), including case studies on successful back-end platforms, the author can determine the most effective strategies for achieving a reliable, high-performance, and secure application. Additionally, the thesis provides a basic review on applying commonly adopted web development frameworks and tools, such as security frameworks, database management systems, to build a scalable job board platform.

2.6 Expectations

The outcomes of this research will benefit individuals engaged in software development, businesses, and those interested in creating full-stack applications utilizing new technologies such as Bun and Svelte. This will also provide valuable insights into various techniques for brainstorming, planning, developing, implementing and debugging. In addition, this study aims to outline a new approach in building back-end systems and details each phase of full-stack application development, documenting the detailed decision-making process, code snippets, and visual aids to support co-workers or people with an interest in the field of software development. Additionally, the outcomes of this thesis will assist companies in optimizing their recruitment processes, refining candidate outreach strategies, and delivering a more accessible and engaging application for users.

Finally, this thesis seeks to bridge the gap between the new experimental tech-stack with already well-presented practical application in web development. The practical examples and guidance provided in this thesis can serve as an interesting resource, an intriguing reference for developers, established software development professionals, and companies who wish to maintain a competitive edge in the digital recruitment landscape.

3 Programming Language

Following the reasoning in Part 1: Introduction, the programming language of choice is TypeScript (TS). The author will explain in briefly about the background of the programming language, the main strengths of TS, as well as the drawbacks. And lastly, how the choice is beneficial for this project.

3.1 Introduction

Web development is always on the move of innovation with new tech, new libraries, and new frameworks that appear in a matter of days, yet JavaScript is the primary language of the web and will most probably hold the position for a long time. JavaScript itself has both strengths and weaknesses. There is a set of tools that was developed to offset JavaScript's weaknesses. One of those tools is TypeScript, which that developed by Microsoft and is a statically typed, object-oriented programming language that builds on JavaScript by adding type definitions, interfaces, classes, and modular structure. As the TypeScript team from Microsoft states, TypeScript which is a typed superset of JavaScript retains JavaScript's syntax and characteristics while improving it with features that support large-scale application development, especially in cases requiring maintainable and scalable code. These additions make TypeScript an ideal choice for big, complex, team projects where strict data-typed code, clear code structure and early error detection are essential.

For the back-end, this thesis leverages BunJS, a runtime built with a TypeScript-first approach, enabling robust TypeScript support from the ground up. BunJS combines TypeScript's strong typing and Bun's performance-focused runtime to create a development environment that is both verbose and highly optimized for server-side applications. (Source: Bun.sh)

TypeScript's enhanced features, such as decorators and modules, provide additional structure and organization options for developers. This allows for modular and manageable code that can be scaled effectively as the codebase

grows. Programming TypeScript by Boris Cherny (2019) emphasizes the language's flexibility and its role in facilitating the development of complex applications, thanks to its type-safe structure and compatibility with existing JavaScript codebases. By combining ease of use with rigorous type checking, TypeScript bridges the gap between developer productivity and application robustness, making it an indispensable tool for modern web development.

3.2 The advantages of TypeScript

One of the core advantages of TypeScript is its static typing, which provides developers with a safety net by catching potential errors at compile time rather than at runtime. This feature not only improves code reliability but also facilitates a smoother development experience. According to Pro TypeScript by Steve Fenton (2017), TypeScript's strong typing system and compiler enable developers to identify issues before deployment, significantly reducing debugging time and enhancing overall code quality. Moreover, TypeScript's close alignment with modern JavaScript standards ensures compatibility with popular frameworks like React, Angular, and Vue, making it a valuable tool for both front-end and back-end development. Additionally, TypeScript offers features like decorators, enabling programmers to embed metadata within their code.

```typescript
export type Email = `${string}@${string}.${string}`;

export interface DBEntity {
  id: string;
  createdAt: Date;
  updatedAt: Date;
}

export interface DBUser extends DBEntity {
  name: string;
  email: Email;
  password: string;
}

export interface DBChat extends DBEntity {
  ownerId: DBUser["id"];
  name: string;
}

export type MessageType = "assistant" | "user";

export interface DBMessage extends DBEntity {
  chatId: DBChat["id"];
  type: MessageType;
  message: string;
}

export type DBCreateUser = Pick<DBUser, "email" | "password" | "name">;
export type DBCreateChat = Pick<DBChat, "name" | "ownerId">;
export type DBCreateMessage = Pick<DBMessage, "chatId" | "message" | "type">;
```

Code 1. Implementation of type and interface, and Pick, Omit in TS Development.

Another advantage of TypeScript is that it enhances productivity and collaboration, particularly in larger teams. The strong typing system enforces consistent data types and predictable code behavior, reducing misunderstandings and errors among team members. Furthermore, Type Safety also helps to improve tooling with modern IDEs, provides developers with accurate code completion, real-time error detection, and robust refactoring tools, pivots the development workflow, improves the communication between teams, and makes complex applications easier to manage.

Refactoring is another crucial activity in software development because it allows developers to have room to improve features and functionality without messing

up the existing code's behavior. TypeScript adds some advanced features such as Type Annotations, Interfaces, Access Modifiers, Enums, Namespace, Modules, and Advanced Types, making it easier to write code while minimizing the risk of unintended impacts on other parts of the codebase. Additionally, according to Le, D.A. (2023) in his thesis, TypeScript's support for gradual adoption enables teams to introduce typed code into existing JavaScript projects, making it suitable for both new and legacy codebases.

Finally, understanding and reading code, especially code written by others, is an important part of the development process, and that can be challenging. TypeScript simplifies this process by enforcing strict syntax rules, clear variable declarations, and a more organized code structure. Additionally, with type validation and built-in documentation features, TypeScript enhances overall code readability and maintainability. This, combined with TypeScript's compatibility with ECMAScript features, allows developers to future-proof their code, ensuring it remains relevant and compatible with future JavaScript updates. In addition, the added feature from TypeScript helps developers produce better documentation.

```
export class SimpleInMemoryResource<T extends S & DBEntity, S>
  implements IDatabaseResource<T, S>
{
  private data = new Map<string, T>();

  async create(data: S): Promise<T> {
    const entity = {
      ...data,
      id: uuidv4(),
      version: 1,
      createdAt: new Date(),
      updatedAt: new Date(),
    } as T;

    this.data.set(entity.id, entity);
    return entity;
  }

  async update(id: string, data: Partial<S>): Promise<T | null> {
    const entity = this.data.get(id);
    if (!entity) return null;

    const updated = {
      ...entity,
      ...data,
      version: entity.version + 1,
      updatedAt: new Date(),
    } as T;

    this.data.set(id, updated);
    return updated;
  }

  async delete(id: string): Promise<T | null> {
    const entity = this.data.get(id);
    if (entity) {
      this.data.delete(id);
      return entity;
    }
    return null;
  }

  async get(id: string): Promise<T | null> {
    return this.data.get(id) || null;
  }
```

Code 2. An implementation of memory_app with TypeScript.

3.3 The disadvantages of TypeScript

As a superset of JavaScript, that enhances the development of complex web applications. Its strong typing system, improved code organization, and better developer tools contribute to a more structured and maintainable codebase. However, there are several drawbacks that can negatively impact the development process. This section examines key disadvantages of using TypeScript.

One of the main drawbacks of TypeScript is that it can prolong the development cycles/sprints. TypeScript's strict typing system requires developers to explicitly define data types for variables, which can be time-consuming, especially for complex applications. This added step can slow down development and can be especially challenging for developers who are new to static typing. Beginners or those unfamiliar with typed languages may struggle with TypeScript's concepts, such as interfaces, generics, and type definitions, making the learning curve steeper compared to JavaScript. Complex type definitions can make the code harder to maintain or read, especially for developers unfamiliar with TypeScript. Type annotations and interfaces can make TypeScript code more verbose than JavaScript. This can lead to longer, more complex code, which might be seen as a downside for simple projects. "Cherny, B. (2019)"

TypeScript code must be compiled into JavaScript before execution. Since most web browsers do not natively support TypeScript, this additional compilation step is necessary. It increases iteration times and complicates the build pipeline, making the deployment process more complex. From the author's experience working with this technology, TypeScript requires additional setup compared to JavaScript, including installing Node.js, npm, or Bun and the TypeScript compiler. Developers often need to configure extra tools, such as Webpack, BunJS to handle TypeScript, which can be time-consuming and confusing for those new to the language. TypeScript code must be compiled to JavaScript before it can be executed. This adds an extra step to the development process and can complicate building and deployment pipelines. The configuration can become

even more complex than plain JavaScript, and with any additional step in the configuration, it is more likely to break.

TypeScript’s static typing system can create compatibility issues with some JavaScript libraries. Not all JavaScript libraries are written with TypeScript in mind, and integrating such libraries often requires writing or searching for custom type definitions. This adds an extra step that JavaScript developers do not face, and it can limit the flexibility of TypeScript when working with third-party libraries, especially older ones not updated to support TypeScript.

For small or quick projects, the benefits of TypeScript may not justify the overhead of setting up types and compiling the code. In such cases, plain JavaScript may be a more efficient choice, as TypeScript’s additional features may not offer enough advantages to outweigh the extra setup and maintenance. While TypeScript is widely adopted, some libraries and third-party tools may still have better support for JavaScript. This means developers might need to invest additional effort to find or create TypeScript type definitions for existing JavaScript libraries.

3.4 Software Development Principles

TypeScript enables programmers to write cleaner and more concise code by supplying type annotations. Using that advantage point, developers can readily detect errors early during the development phase, therefore reducing errors that occur at runtime. In addition, TypeScript provides improved code navigation and completion and enables developers to write more robust code by leveraging interfaces, classes, and inheritance. TypeScript also offers enhanced tooling support and compiler optimizations to ensure the code is quick and efficient. Overall, TypeScript provides numerous benefits for developers, and it can considerably enhance the quality of code by providing improved type checking and a more solid code structure. Overall, TypeScript isn’t flawless, but the type safety it offers leads to more reliable and error-free code. This benefit is often more crucial for larger projects than the drawbacks that TypeScript might present.

To create more reliable and error-free code, the developers should employ some software development principles such as SOLID or DRY. Which the inspiration by Levy, J.-J. (2023), the author presents his understanding of SOLID and DRY below.

3.4.1 SOLID

The SOLID principles are the fundamental concepts in software development, which are one of the most common practices that any developer should know. These principles promote the design of robust, scalable, and maintainable code. SOLID is an acronym representing five key design principles: Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP).

The Single Responsibility Principle (SRP) asserts that a class should have only one well-defined responsibility, meaning it should handle a single task or aspect of the system. This approach enhances code clarity, simplifies maintenance, and improves reusability. For example, instead of combining user authentication and notification logic in one class, these responsibilities should be separated into distinct classes. Applying SRP leads to more modular code, making future modifications easier while isolating issues to specific components. Additionally, it encourages reusability, as specialized classes can be utilized across different parts of the system.

The Open/Closed Principle (OCP) emphasizes that software entities should be open for extension but closed for modification, meaning new functionality should be added through extension rather than altering existing code. Techniques such as inheritance, polymorphism, and dependency injection help achieve this. By adhering to OCP, developers can introduce new features without disrupting existing behavior, improving flexibility and reducing regression risks during testing.

The Liskov Substitution Principle (LSP) states that derived classes should be fully substitutable for their base classes without affecting system correctness. If class B inherits from class A, it should seamlessly replace A without introducing unexpected behavior. Following LSP ensures consistency in object-oriented design, allowing new subclasses to be integrated smoothly without breaking existing functionality. This principle enhances modularity and reusability, as subclasses maintain the expected contract of their parent classes.

The Interface Segregation Principle (ISP) advocates for designing small, client-specific interfaces rather than large, general-purpose ones. Clients should not be forced to depend on methods they do not use. By adhering to ISP, interfaces become more focused and easier to understand, while modifications to an interface only impact the clients that rely on the relevant methods. This results in cleaner, more maintainable code.

The Dependency Inversion Principle (DIP) encourages relying on abstractions (interfaces or abstract classes) rather than concrete implementations. High-level modules should depend on these abstractions rather than low-level details, reducing tight coupling between components. Applying DIP improves modularity, as implementations can be swapped without affecting dependent modules. It also simplifies testing, as dependencies can be easily mocked, and promotes a more flexible, reusable architecture.

3.4.2 DRY

The DRY (Don’t Repeat Yourself) principle emphasizes avoiding unnecessary code duplication in a software development prototype. According to this principle, each simple task or logic should have a single representation within the system, and any similar code or logic can be prototyped and called up on request.

First, it reduces code complexity by avoiding unnecessary repetitions. This makes the code more readable, clear, easy to follow a code standard by an individual or a team. As a result, the code maintenance process can be simplified and sped

up as any fixes or updates only need to be made in one place rather than in multiple parts of the code.

```
// Bad - repetitive code
function calculateAreaOfCircle(radius: number): number {
  return Math.PI * radius * radius;
}

function calculateCircumferenceOfCircle(radius: number): number {
  return 2 * Math.PI * radius;
}

// Good - reusable function for circle calculations
class CircleCalculator {
  static area(radius: number): number {
    return Math.PI * radius * radius;
  }

  static circumference(radius: number): number {
    return 2 * Math.PI * radius;
  }
}

// Usage
const area = CircleCalculator.area(5);
const circumference = CircleCalculator.circumference(5);
```

Code 3. Example of Reusable Function in TS.

Second, it promotes code reuse, as generic utility functions or logics can be encapsulated into broader functions, classes, or interfaces that can be used in multiple places within the system. This way, the same code can be called in multiple places without needing to rewrite it.

On the other hand, the use of types and interfaces, thanks to TypeScript, can encapsulate common methods such as CRUD (Create, Read, Update, Delete) and reuse them in specific situations. Common methods can be defined once and then called with different parameters.

Finally, the use of libraries, modules, or frameworks can help in reinventing the wheel, which leads to avoiding a lot of reusing code in the code base. Also, with reputable libraries, frameworks that have already been tested by thousands of people could provide the project a great security if used in the right way. In this thesis, some notable libraries and frameworks are used Hono, Axios, …

4 Back-end Development

> "The back-end is all the technology required to process the incoming request and generate and send the response to the client. This typically includes three major parts: The server, the app, and the database." – CodeAcademy.

According to CodeAcademy, the back-end system is all the infrastructure that is used to create communication between the client and the server, and all the jobs related to extracting, storing, etc., data. In this thesis, the back-end part is the logic engine, which is written by TS, BunJS as the runtime/server, and using Docker to host a PostgreSQL database.

4.1 BunJS

Picture 2. Bun official logo. (Source: Bun.sh)

BunJS is a modern JavaScript runtime designed to address the evolving demands of web development. Developed with a focus on performance, simplicity, and developer efficiency, BunJS serves as a streamlined alternative to traditional runtimes like Node.js and newer entrants like Deno. At its core, BunJS

combines a runtime, a package manager, and a bundler into a single tool, aiming to simplify the development workflow and enhance performance.

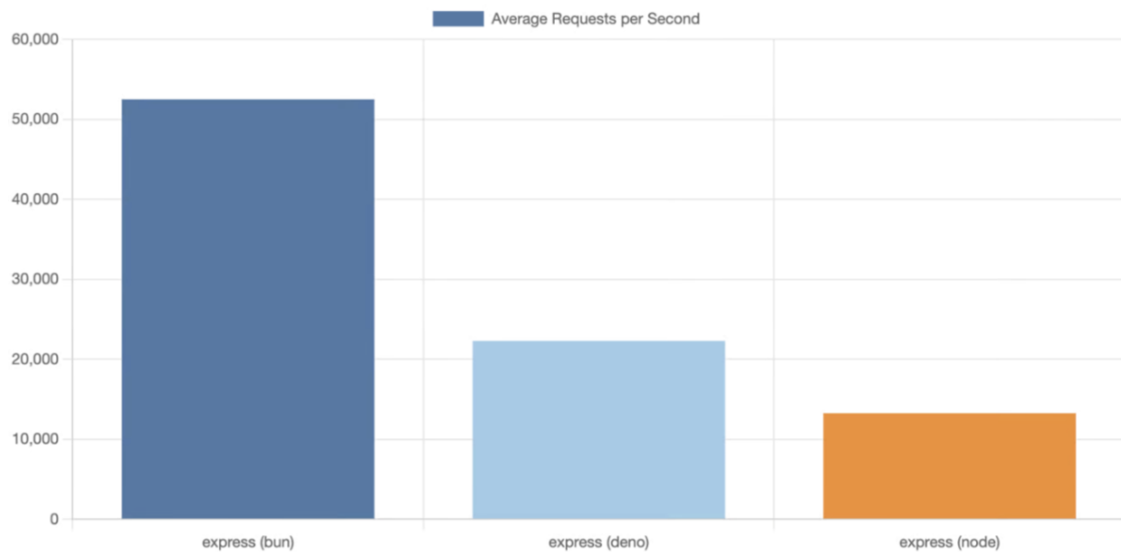
Bun is constructed upon the JavaScriptCore engine utilized in Safari, which is recognized for its exceptional speed, as stated by Sumner, J. (2025). Moreover, the employment of the low-level programming language Zig renders Bun an extraordinarily efficient runtime, providing a performance enhancement that can accommodate 5 to 10 times the load that Node.js can manage. According to Ahmod, Md Feroj (2023), Bun is one of the fastest back-end runtimes in terms of request performance and one of the best back-end JavaScript runtimes in terms of utilizing the hardware resources (CPU and memory).

The first primary goal of Bun is to increase the speed of application. According to the research of (Ahmod, Md Feroj, 2023), the author can determine that it remarkably rapid initialization time and enhanced operational efficiency. BunJS endeavors to minimize latency within both developmental and production contexts. Besides that, through the integration of essential tools such as a bundler, transpiler, and task runner, BunJS significantly diminishes the necessity for external dependencies and configurations, thereby optimizing the development workflow. Developers require solely Bun to interpret and transpile their code, obviating the need for configuration of tools like webpack, esbuild, or rollup. Which leads to a better development experience, BunJS allows developers to concentrate on code creation rather than the management of configurations or supplementary tools.

Furthermore, declared in the BunJS documentation, with the first-class support for TypeScript and NPM modules, BunJS can be seamlessly implemented and integrated into the modern JavaScript ecosystem without sacrificing the efficiency of developers. BunJS was created to address some of the pain points commonly encountered in JavaScript development, such as performance bottlenecks, fragmented tooling, and lengthy build times. Bun offers a selection of essential tools required for the development of most real-world projects out of the box, including test runners, environment variable managers, and password generators. This provision is likely to conserve substantial time, as otherwise, the

introduction of external libraries for these functionalities would be necessary. The package manager of Bun is meticulously designed to surpass the speed and efficiency of traditional package managers such as npm or yarn, exhibiting performance improvements ranging from 2 to 10 times in various scenarios (Source: betterstack.com). In addition, BunJS offers a selection of essential tools that are required for the development of most projects out of the box, including test runners, environment variable managers, and password generators. This provision is likely to conserve substantial time, as otherwise, the introduction of external libraries for these functionalities would be necessary.

Another big advantage of Bun is crafted to serve as a seamless substitute for Node.js. It natively encompasses most of the fundamental Node.js and Web APIs, thus enabling developers to utilize packages originally designed for Node.js within Bun, and even to replace Node.js with Bun in existing Node.js projects. At the time of writing this thesis, BunJS states on its official webpage that it does not guarantee 100% compatibility in every instance; its compatibility is sufficiently above 95%, allowing most projects to operate with minimal modifications.



Framework	Runtime	Average	Ping	Query
express	bun	52,479.34	58,955.77	50,583.29
express	deno	22,286.51	23,318.99	22,414.20
express	node	13,254.55	16,821.80	16,250.99

Figure 1. Benchmark result of express server. (Source: Betterstack.com)

Comparison with existing JS/TS Framework:

- Node.js is the most established JavaScript runtime, leveraging the V8 engine. Although it boasts a vast ecosystem and mature tooling, performance can be limited by the need for external tools like Webpack or Babel for tasks like bundling and transpilation. BunJS addresses these limitations by offering integrated tools, which lead to faster and more streamlined development.
- Deno is another modern runtime that emphasizes security and improved developer ergonomics, also built-in support for TypeScript and ES modules. However, Deno lacks some of the extensive ecosystem compatibility that Node.js provides. Compared to Deno, BunJS retains

compatibility with NPM, giving it a significant advantage in adopting existing projects while keeping performance benefits.

Code 4 is a quick demonstration of how fast and easy a BunJS server can start and listen on the port of choice.

index.tsx

```
import { sql, serve } from "bun";

const server = serve({
  port: 3000,
  routes: {
    "/": () => new Response("Welcome to Bun!"),
    "/api/users": async (req) => {
      const users = await sql`SELECT * FROM users LIMIT 10`;
      return Response.json({ users });
    },
  },
});

console.log(`Listening on localhost:${server.port}`);
```

Code 4. Starting a Bun server on localhost:3000. (Source: Bun.sh)

Also, it is imperative for developers to establish the configuration of the service to operate utilizing Hot Module Replacement (HMR) to observe real-time updates both on the server and within the Document Object Model (DOM) of selections. Engaging in this practice will contribute to a more coherent and effective development process.

```
▷Debug
"scripts": {
  "dev": "bun run --hot src/index.ts"
},
```

Code 5. How hot reload is set in Bun.

4.2 Hono

Hono is a modern, ultra-fast, and lightweight web framework built on Web Standards, designed specifically for developing web applications and APIs. Developed in TypeScript, it emphasizes speed, ease of use, and developer productivity, making it an attractive choice for full-stack web development. Hono is inspired by the popular Express.js framework but focuses on offering superior performance while maintaining a minimal resource footprint. As a TypeScript-based framework, Hono aims to provide a better developer experience by leveraging the strong typing features of TypeScript, ensuring code quality and reducing errors during development. Although relatively new, Hono is gaining attention as a streamlined alternative to heavier frameworks, offering a straightforward API that can be quickly picked up by developers familiar with Express.js or similar frameworks. (Source: Hono Documentation)

Hono boasts several key features that make it ideal for building fast and scalable web applications. Its core is highly optimized for speed, ensuring lightning-fast request handling, which is particularly beneficial for applications requiring high-performance APIs. The Express-like syntax makes it easy for developers to quickly adapt to the framework, lowering the learning curve, even for those with limited experience. Hono is also highly scalable, making it suitable for both small applications and high-traffic APIs, a feature that is critical for modern web development. Moreover, Hono is designed to be flexible, offering compatibility with serverless platforms such as AWS Lambda and Cloudflare Workers, allowing developers to deploy applications efficiently across different environments. However, there are some drawbacks. The ecosystem around Hono is still relatively small compared to more established frameworks like Express.js or Next.js, which means fewer available resources, such as plugins, tutorials, and community support, potentially making it challenging for beginners. Additionally, since it is a newer framework, it may lack the maturity and battle-tested stability of older frameworks, which could pose challenges in mission-critical production environments. Despite these limitations, Hono remains a compelling option for

developers seeking a high-performance, TypeScript-centric framework for modern web applications. (Source: Hono Documentation)

```
import Bun from "bun";
import { Hono } from "hono";
import { logger } from "hono/logger";
import { timing } from "hono/timing";

const app = new Hono();
app.use("*", timing());
app.use("*", logger());

app.get("/", (c) => {
  return c.json({ message: "Hello Hono!" });
});
console.log(Bun.env.TEST);
console.log(Bun.env.TEST2);
console.log(Bun.env.TEST3);
export default app;
```

Code 6. Basics of starting a server with Hono which includes logger and timing.

When this simple code is executed, the results can be observed via the terminal:

```
cozygarage@MacBookAir chat_backend % bun run dev
$ bun run --hot src/index.ts
test value;
test value 2;
test value 3;
Started server http://localhost:3000
  <-- GET /
  --> GET / 200 0ms
  <-- GET /favicon.ico
  --> GET /favicon.ico 404 0ms
```

Code 7. Working server with timing and logger.

Bun also automatically scans for all environmental variables and returns the correct value.

4.3 Linting and Formatting

Using linters helps developers detect and catch potential bugs and problems even before the codebase grows bigger, which might turn them to be bigger problems. Early detection could help developers deal with the issue beforehand, even before running it. For example, linter tools are useful in catching unused VAR, …On the other hand, the formatting tools help developers maintain a readable code and enforce a consistent style across the codebase, saving time and reducing the occurrence of small mistakes like syntax errors. The value of formatting is multiplied by the number of people in the project. Zakas, N.C. (2024) stated that in his official ESLint Documentation.

In this thesis scope, the ESLint package will be used as the linter tool. This is the most common linter in JavaScript's ecosystem, according to Colandrea, D. (2023), which includes TypeScript. There is a special technical detail that must be noticed in this project. Both the normal ESLint package and the recommendation from Hono must be installed.

```
bun install --dev @hono/eslint-config
```

Code 8. Install hono/eslint.

Followed by installing ESLint as a linter tool, the formatter can be easily installed. The formatter of choice here is Prettier, also one of the most popular formatters in the JS ecosystem. Throughout the project, the author can use Bun as a package manager, reducing the need for different sets.

```
% bun install --dev eslint-config-prettier eslint-plugin-prettier
```

Code 9. Install Prettier.

4.4 Authorization and Authentication.

In Hono's architecture, middleware functions act as interceptors that process HTTP requests before they reach their intended route handlers. These functions can also modify outgoing responses before their delivery to clients. Their functionality includes request modifications (such as body parsing or header injection) and response adjustments (like cookie setting or header manipulation).

The execution sequence of middleware is determined by their declaration order in the codebase. Each middleware component possesses the autonomy to either propagate the request to subsequent handlers or terminate the response pipeline. Typical applications include request logging, user authentication, error management, and input data processing. Source: Chowdhury, M.S.N. (2024).

For this research implementation, the primary middleware deployed handles authentication using JSON Web Tokens (JWT). Following the RFC 7519 standard, JWTs provide a compact, self-contained method for secure information exchange between systems through JSON-formatted tokens. While all JWTs are tokens, it's important to note that not all tokens conform to the JWT specification. Their compact nature enables transmission via URLs, POST parameters, or HTTP headers with minimal overhead.

A key advantage of JWT implementation lies in its self-contained design - each token carries complete entity authentication data, eliminating repetitive database queries for authorization checks. This design also removes the need for server-side token validation, as the token itself contains all necessary verification information. “Auth0 (2024)”

```
export const API_PREFIX = "/api/v1";

export async function checkJWTAuth(
  c: Context,
  next: () => Promise<void>,
): Promise<Response | void> {
  const publicRoutes = [
    API_PREFIX + JOBS_PREFIX, // GET /api/v1/jobs/
    API_PREFIX + JOBS_PREFIX + ":id/", // GET /api/v1/jobs/:id/
  ];
  if (
    publicRoutes.some((route) => c.req.path.match(new RegExp(route))) ||
    c.req.path === API_PREFIX + AUTH_PREFIX + LOGIN_ROUTE ||
    c.req.path === API_PREFIX + AUTH_PREFIX + REGISTER_ROUTE
  ) {
    return await next();
  }
  const { JWT_SECRET } = env<{ JWT_SECRET: string }, typeof c>(c);
  // Add validation for JWT_SECRET
  if (!JWT_SECRET || JWT_SECRET.length < 32) {
    throw new HTTPException(500, {
      message: "JWT_SECRET not configured properly",
    });
  }

  const authMiddleware = jwt({
    secret: JWT_SECRET,
  });
  return authMiddleware(c, next);
}

export async function attachUserId(
  c: Context,
  next: () => Promise<void>,
): Promise<Response | void> {
  const payload = c.get("jwtPayload") as { userId: string };
  if (payload?.userId) {
    const id = payload.userId;
    c.set("userId", id);
  }
  await next();
}
```

Code 10. Implementation of Authentication and Authorization with JWT

4.5 REST API

According to MDN Web Docs. (2023) REST (Representational State Transfer) APIs work as a standardized approach for client-server communication over HTTP/HTTPS, enabling seamless data exchange and interaction between server resources. In a full-stack chat application, these APIs act as the critical link between front-end interfaces and back-end systems, facilitating operations on various resources like messages, user profiles, and chat conversations. The architecture follows a straightforward yet powerful principle: using standard HTTP methods to perform CRUD operations on resources. The fundamental methods include GET for retrieving data, POST for creating new records, PUT/PATCH for updating existing ones, and DELETE for removing resources.

Central to REST API design is the concept of resource representation - the snapshot of a resource's state at any given moment, typically formatted for client interpretation. JSON has emerged as the predominant format for this purpose due to its optimal balance of human-readability, processing efficiency, and platform independence. While alternatives like XML and plain text exist, JSON's versatility makes it particularly suitable for client-server data exchange in modern web applications.

Beyond basic HTTP methods, RESTful implementations extensively utilize request headers and parameters to extend functionality. Headers carry crucial metadata, including authentication details (like API tokens), content specifications (such as application/json), and caching directives. Parameters provide supplementary information for operations, enabling features like filtered queries (e.g., retrieving messages by sender or date range). Similarly, response headers convey important feedback through status codes (200 for success, 404 for missing resources, etc.) and operational metadata like rate limits or cache controls.

The strategic use of HTTP status codes forms another vital component of REST API architecture, offering immediate insight into request outcomes. Common indicators range from 200-series codes for successful operations (200 OK, 201

Created) to 400-series codes for client errors (400 Bad Request, 401 Unauthorized) and 500-series codes for server-side issues. Proper implementation of these elements ensures robust, secure, and user-friendly API interactions.

In practical application, particularly in systems like a job chat platform, effective REST API integration enables smooth user experiences by seamlessly connecting front-end interfaces with back-end services and databases. This architectural approach mirrors web navigation simplicity, where HTTP methods provide built-in actions for data interaction. Clients initiate operations through HTTP requests, with servers responding accordingly through structured data exchanges. The thesis leverages this REST API paradigm to create a cohesive, secure, and efficient system that meets modern web application requirements while maintaining data integrity and user satisfaction.

```
export const CHAT_PREFIX = "/chat/";
const CHAT_ROUTE = "";
const CHAT_DETAIL_ROUTE = ":id/";
const CHAT_MESSAGE_ROUTE = ":id/message/";
export function createChatApp(
  chatResource: IDatabaseResource<DBChat, DBCreateChat>,
  messageResource: IDatabaseResource<DBMessage, DBCreateMessage>,
) {
  const chatApp = new Hono<ContextVariables>();

  chatApp.post(CHAT_ROUTE, zValidator("json", chatSchema), async (c) => {
    const userId = c.get("userId");
    const { name } = c.req.valid("json");
    const data = await chatResource.create({ name, ownerId: userId });
    console.log(c.req.path);
    c.get("cache").clearPath(c.req.path);
    return c.json({ data });
  });

  chatApp.get(CHAT_ROUTE, async (c) => {
    const userId = c.get("userId");
    const data = await chatResource.findAll({ ownerId: userId });
    const res = { data };
    c.get("cache").cache(res);
    return c.json({ data });
  });

  chatApp.get(CHAT_DETAIL_ROUTE, zValidator("param", idSchema), async (c) => {
    const { id } = c.req.valid("param");
    const userId = c.get("userId");
    const data = await chatResource.find({ id, ownerId: userId });
    const res = { data };
    c.get("cache").cache(res);
    return c.json({ data });
  });

  chatApp.get(CHAT_MESSAGE_ROUTE, zValidator("param", idSchema), async (c) => {
    const { id: chatId } = c.req.valid("param");
    const data = await messageResource.findAll({ chatId });
    const res = { data };
    c.get("cache").cache(res);
    return c.json(res);
  });

  chatApp.post(…
  );
  return chatApp;
}
```

Code 11. A functional MVP working chat app that follows RESTful API Design.

In this thesis, due to the limitations in time and resources, the author has implemented the two most important features for a Minimum Viable Product (MVP): Authentication APIs and Chat APIs, which are described below:

- POST /auth/register: Registers a new user by validating the provided email, password, and name, and stores the user in the database after hashing the password.
- POST /auth/login: Authenticates a user by verifying their email and password and returns a JWT token upon successful authentication.
- POST /chat/: Creates a new chat room with a specified name, associated with the authenticated user.
- GET /chat/: Retrieves a list of all chat conversations owned by the authenticated user.
- GET /chat/:id/: Retrieves detailed information about a specific chat room, identified by its ID, provided the authenticated user is the owner.
- GET /chat/:id/message/: Retrieves all messages associated with a specific chat room, identified by its ID.
- POST /chat/:id/message/: Allows the authenticated user to send a message to a specific chat room, identified by its ID, and automatically generates a dummy response from the assistant.

4.6 Project Structure

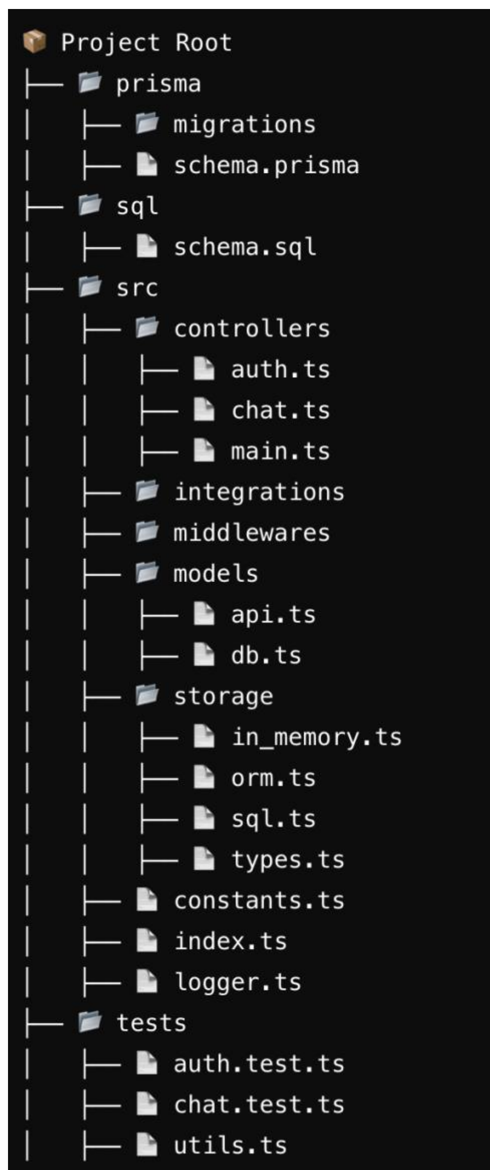


Figure 2. Project Structure on Back-end.

When structuring code for a project, it’s essential to have a clear separation of concerns. This means organizing files and folders in a way that each component has a distinct responsibility, contributing to overall project clarity and maintainability. For this relatively small project, a simple structure is used. The structure and the rationale behind it are shown below:

- src/controllers: This folder will contain specific REST endpoint handlers. Each controller deals with incoming requests and generates appropriate responses. By isolating endpoint logic in controllers, it is easier to update or extend API functionalities.
- src/middlewares: Here, the author stores additional middleware functions for Hono. Middleware is crucial for processing requests and responses, offering functionality such as authentication, logging, or data parsing. Keeping them in a dedicated folder allows for easy reuse and management.
- src/models: This directory is designated for type definitions of the objects used in the code. It ensures a centralized location for data structure definitions, enhancing code consistency and reducing the likelihood of type-related errors.
- src/storage: A folder to manage code that interacts with various storage solutions, such as in-memory databases, SQL databases, or ORMs. This separation ensures that changes in storage can be unified.
- “src/constants.ts: This is a file to hold project-wide constants. These centralizing constants ensure uniformity and prevent discrepancies that can arise from hardcoding values in multiple locations.
- src/index.ts: This is the entry point of the application. These are various components of the application that are tied together, setting up the server, middleware, routes, and any initial configurations.
- tests: This is a dedicated folder for storing test files.

4.7 Validation

Validating incoming data to the endpoints of the application is crucial for maintaining the integrity and security of the application. Validating data can be done in many different stages of the application, such as front-end, database, and API. However, the main rule of development, in the author’s opinion, is that the back-end has the final responsibility to maintain the integrity of the incoming data. That’s why proper validation acts as a first line of defense, filtering out malformed, corrupt, or malicious data before it can interact with the back-end

system. It ensures that the data aligns with the expectations and requirements, safeguarding the application from unexpected behavior, crashes, or security vulnerabilities. The knowledge is from the author's experience and the research from Full-Stack Web Development with TypeScript 5.

By rigorously checking incoming data, the author not only protects the back-end processes and databases but also provides a more reliable and user-friendly experience, especially in cases when the API is used by third parties. In the scope of this project, the author will use Zod to handle all the validation on the back-end side.

```
$ bun add zod @hono/zod-validator -d
```

Code 12. Using bun to add 'zod' package into the project.

4.8 Unit testing and Integration testing

According to Katalon's blog, developing tests is not just nice to have; it is essential for building reliable and maintainable applications. Tests act as a safety net, an early detector catching issues before they make it to the final stage, where they can be costly, problematic, and time-consuming to fix. One of the most common types of tests that developers of products should employ all the time is unit tests. Unit tests make sure each part of the application works as expected and continues to do so even when the developers add new features or refactor existing ones. Automating tests allows the team to focus on writing functional code instead of fixing bugs and issues that should not have happened in the first place. Test automation is also one of the main pillars in CI/CD development philosophy. In addition, thorough testing and good coverage increase efficiency, promote independence and build trust between teams by ensuring the code is reliable.

One of the most common developing paradigms is known as Test-Driven Development (TDD). It forces developers to think about the structure of the code before starting to write, which helps develop and avoid the impulsiveness of writing code without thoroughly planning, because the frustration of non-working code after a hundred lines can kill productivity fast. In addition, it encourages developers to write testable code, which usually leads to more modular code with a clear separation of concerns.

```
✓ chat tests > POST /chat/:id/message - incorrect body [60.82ms]
---------------------------------------------------|---------|---------|-------------------
File                                               | % Funcs | % Lines | Uncovered Line #s
---------------------------------------------------|---------|---------|-------------------
All files                                          |   93.33 |   91.18 |
 chat_backend/src/constants.ts                     |  100.00 |  100.00 |
 chat_backend/src/controllers/auth.ts              |  100.00 |  100.00 |
 chat_backend/src/controllers/chat.ts              |   90.00 |   91.30 | 50-55
 chat_backend/src/controllers/main.ts              |  100.00 |  100.00 |
 chat_backend/src/logger.ts                        |  100.00 |  100.00 |
 chat_backend/src/middlewares/auth.ts              |  100.00 |  100.00 |
 chat_backend/src/middlewares/cacheMiddleware.ts   |   83.33 |   86.36 | 28-29,52-55
 chat_backend/src/middlewares/rateLimiting.ts      |  100.00 |   89.29 | 25-26
 chat_backend/src/storage/in_memory.ts             |   66.67 |   53.70 | 17-30,50-60
---------------------------------------------------|---------|---------|-------------------

 11 pass
 0 fail
 32 expect() calls
```

Picture 3. Result of API testing with Bun.

The test process is automated with Bun. By using the coverage mode, the developer can request the coverage of the test cases. Generally, any test that could cover 90% of the code could be considered a good test.

```
describe("auth tests", () => {
  const app = createORMApp();
  const prisma = new PrismaClient();

  beforeEach(async () => {
    await resetORMDB(prisma);
  });

  test("POST /register - normal case", async () => {…
  });

  test("POST /register - user already exists", async () => {…
  });

  test("POST /login - success", async () => {…
  });

  test("POST /login - non-existing user", async () => {…
  });
  test("POST /register - incorrect body", async () => {…
  });
  test("POST /login - incorrect body", async () => {…
  });
});
```

Code 13. Authentication test cases on Bun.

For authentication, the test is mostly used to send a POST request to the server, either creating a new row in the user database or logging in an existing user. There are 6 test cases:

- Register – normal, user already exists, bad request.
- Login – normal, non-existing user, bad request.

```
async function getToken(email = "test@test.com"): Promise<string> {⋯
}

async function createChat(token: string) {⋯
}

test("GET /chat/ - get user chats", async () => {⋯
});
test("GET /chat/ - get user chats when multiple chat and users are available", async ()
});
test("POST, GET /chat/:id/message/ - create and get chat messages", async () => {⋯
});
test("POST /chat - incorrect body", async () => {⋯
});
test("POST /chat/:id/message - incorrect body", async () => {⋯
});
});
```

Code 14. Chat and Message test with Bun.

4.9 Security

Security is one of the most important aspects of a reliable and solid web application. Not only does the application have to resolve requests fast and accurately, but it also must be safe and reliable. The idea of a good web application is that it functions as expected, even under the threat of cyberattacks. According to Andreeva, O. (2024) and Blackduck, there are some of the most common security threats that a web and software must deal with. Implementing security measures is crucial for the application, specifically against SQL injection, XSS attacks, DDoS attacks, unauthorized domain requests, and man-in-the-middle attacks. In this thesis, the author will implement strict input validation, HTTPS encryption, and request throttling, so the application can mitigate these risks and ensure a secure back-end infrastructure.

CORS restricts API access to trusted domains, preventing unauthorized front-end requests. Throttling (rate limiting) blocks excessive requests from a single user, protecting against DoS attacks and API abuse. Caching reduces database load by storing frequently accessed data (e.g., chat messages), improving response times. Logging (via Pino) tracks system behavior, helping debug issues

and monitor suspicious activity. Together, these techniques ensure reliability, scalability, and security in production environments.

A well-designed back-end must balance performance and security. While caching speeds up responses, it requires careful invalidation to avoid serving stale data. Throttling prevents abuse but must be fine-tuned to avoid blocking legitimate users. CORS enhances security but must be configured to allow necessary cross-origin requests (e.g., front-end-back-end communication). Logging provides transparency but should avoid excessive data storage to maintain efficiency. The thesis demonstrates how Bun’s lightweight architecture, combined with these optimizations, achieves this balance, delivering fast, secure, and maintainable APIs.

Compared to traditional Node.js back-ends, Bun provides built-in tools that enhance security and optimize performance. For instance, Bun’s native Bun.password.hash securely manages password encryption without the need for external libraries, while its high-performance runtime minimizes the necessity for additional caching layers. In contrast to Node.js, which depends on middleware like express-rate-limit or helmet, Bun’s streamlined design incorporates features such as CORS and logging more efficiently. This positions Bun as an excellent option for full-stack applications, especially when combined with Svelte, by reducing dependencies and boosting execution speed while upholding robust security standards.

5 Data management

According to Google Cloud (2024a), databases are an essential component of modern web applications, providing a systematic way to store, manage, and retrieve data. They allow for an efficient organization of large volumes of information, ensuring its integrity and accessibility over time. In the context of web applications, databases serve as the backbone for data persistence, enabling users to interact with dynamic, ever-changing data, such as user profiles, messages, or transactions. By using databases, developers can structure data in ways that support quick searches, updates, and the maintenance of relationships between different data entities. Databases also play a crucial role in maintaining consistency, ensuring that data remains accurate even during system failures or crashes through various mechanisms like ACID compliance (Atomicity, Consistency, Isolation, Durability). ACID is a recommended compliance as stated in the MariaDB official Documentation.

For this application, the author chose PostgreSQL as the database management system (DBMS) due to its robust features, reliability, and scalability. As an open-source relational database, PostgreSQL is known for its strong adherence to SQL standards and its ability to handle complex queries and large datasets efficiently. It supports a range of advanced features, such as ACID compliance, foreign keys, joins, and triggers, which make it an ideal choice for applications that require data integrity and consistency. Additionally, PostgreSQL is highly extensible, offering a wide array of tools and extensions for customization. Its strong community support and widespread adoption across industries further solidify its position as a top choice for developing scalable, data-driven applications. These advantages make PostgreSQL particularly well-suited for a chat application, where managing user data and interactions requires reliability and performance.

5.1 In-Memory (RAM) temporary database.

```
export interface IDatabaseResource<T, S> {
  create(data: S): Promise<T>;
  update(id: string, data: Partial<S>): Promise<T | null>;
  get(id: string): Promise<T | null>;
  find(data: Partial<T>): Promise<T | null>;
  findAll(data: Partial<T>): Promise<T[]>;
  delete(id: string): Promise<T | null>;
}
```

Code 15. Implementation of In-memory methods.

In the development of the back-end system, the author employed a non-persistent implementation of a database resource using an in-memory storage mechanism. It is designed for simplicity and quick prototyping, allowing data to be stored and managed entirely in memory without the need for a persistent database.

The SimpleInMemoryResource implements the IDatabaseResource interface, providing basic CRUD operations. It stores data in an array, where each entry is an object containing an id, `createdAt`, and `updatedAt` timestamp, along with the provided data.

The in-memory app is used in the `createInMemoryApp` function in main.ts, where it serves as the back-end storage for both authentication and chat functionalities. This approach is ideal for scenarios requiring rapid development, testing, or temporary data storage, as it avoids the overhead of setting up and maintaining a persistent database. However, it is not suitable for production environments due to its lack of data persistence and scalability. A persistent database is needed.

5.2 PostgreSQL

According to Google Cloud, PostgreSQL is one of the most popular relational databases on the market. In the scope of this project, having a relational database is the most optimal choice. PostgreSQL is frequently compared to MySQL, MariaDB, and all the forks, which are also popular open-source RDBMSs. While MySQL is renowned for its speed and reliability in read-heavy scenarios, PostgreSQL shines with its advanced features, such as complex queries and support for multiple concurrent transactions (through MVCC). PostgreSQL's extensibility and standards compliance, including full ACID compliance for transactions, make it a preferred choice for complex applications. It is also compared with SQLite. SQLite is a lightweight, file-based database. It's designed for simplicity and minimal setup, making it ideal for embedded applications and small projects. PostgreSQL, in contrast, offers more robustness and scalability, supporting large datasets and concurrent users more effectively. PostgreSQL, while primarily a relational database, also incorporates JSON support and NoSQL features, allowing for both structured and unstructured data management in a single system. The conclusion is from the author's own experience and the intensive research PostgreSQL Documentation.

```sql
schema.sql
CREATE TABLE "user"
(
    id          SERIAL PRIMARY KEY,
    "createdAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    "updatedAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    name        VARCHAR(500)                           NOT NULL,
    email       VARCHAR(200)                           NOT NULL,
    password    VARCHAR(500)                           NOT NULL
);

CREATE UNIQUE INDEX "user_email_key"
    ON "user" (email);

CREATE TABLE "chat"
(
    id          SERIAL PRIMARY KEY,
    "createdAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    "updatedAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    "ownerId"   INT                                    NOT NULL REFERENCES "user"
        ON UPDATE CASCADE ON DELETE CASCADE,
    name        VARCHAR(1000)                          NOT NULL
);

CREATE TABLE "message"
(
    id          SERIAL PRIMARY KEY,
    "createdAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    "updatedAt" TIMESTAMP(3) DEFAULT CURRENT_TIMESTAMP NOT NULL,
    "chatId"    INT                                    NOT NULL REFERENCES "chat"
        ON UPDATE CASCADE ON DELETE CASCADE,
    type        VARCHAR(100)                           NOT NULL,
    message     TEXT                                   NOT NULL
);
```

Code 16. SQL database schema for the application.

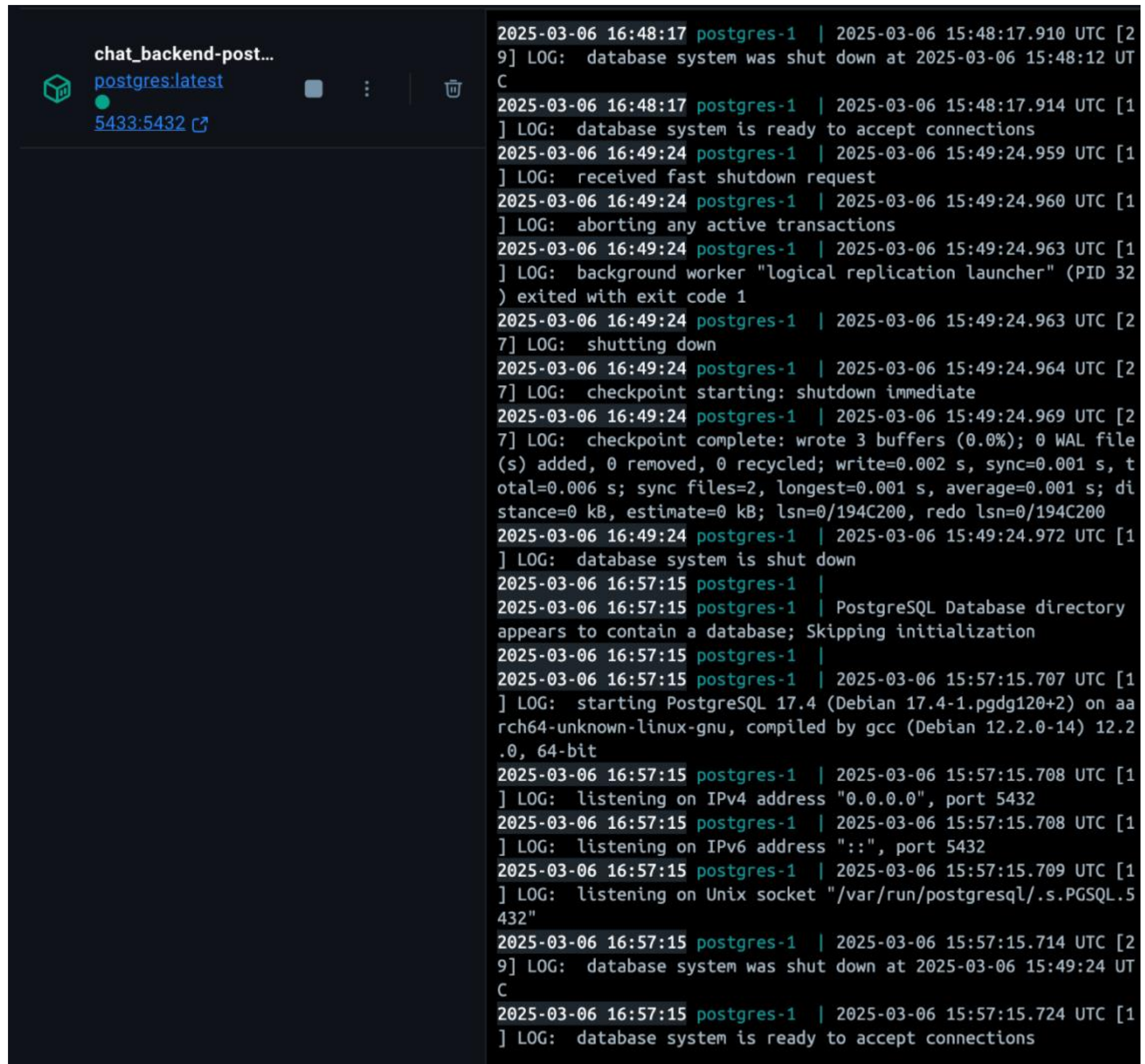
5.3 Docker

Docker provides a powerful platform for packaging applications along with their dependencies into portable containers that can run consistently across various environments and operating systems. To illustrate this concept, consider your application as a fragile item that needs transportation. Docker serves as the ideal protective casing that bundles everything securely. Whether deployed on different machines, operating systems, or cloud platforms, the containerized application remains fully functional without compatibility concerns. This approach significantly simplifies development workflows, particularly for teams working

across diverse systems, while also streamlining deployment since most modern infrastructure platforms natively support Docker containers.

Complementing Docker's core functionality, Docker Compose offers a solution for managing multi-container applications through declarative configuration. Developers can define all application components, including services, networks, and volumes, in a single YAML file, then launch the complete environment with one command. This proves especially valuable when working with complex architectures involving web servers, databases, and caching systems, as it maintains clear relationships between interconnected services while abstracting away deployment complexities. (Sources: Docker Documentation (2020))

The adoption of Docker Compose in this project brings multiple advantages. It guarantees environmental uniformity throughout the development lifecycle, minimizing environment-specific bugs and improving reliability. For microservices-based systems, Docker Compose shines by allowing each service to operate in its container with dedicated dependencies, enabling teams to work on components independently. The container isolation model enhances security by preventing conflicts between services running on the same host. From an operational perspective, Docker simplifies scaling through container replication and eases database management by containerizing data services. This ensures consistent database configurations across environments while simplifying maintenance tasks like backups and horizontal scaling.



Picture 4. A running PostgreSQL database with Docker Compose.

5.4 CRUD Structured Query Language (SQL) operations

```
async create(data: DBCreateUser): Promise<DBUser> {
  const query =
    'INSERT INTO "user" (name, email, password) VALUES ($1, $2, $3) RETURNING *';
  const values = [data.name, data.email, data.password];
  const result = await this.pool.query(query, values);
  return result.rows[0] as DBUser;
}

async delete(id: string): Promise<DBUser | null> {
  const query = 'DELETE FROM "user" WHERE id = $1 RETURNING *';
                                (property) UserSQLResource.pool: Pool
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBUser) : null;
}

async get(id: string): Promise<DBUser | null> {
  const query = 'SELECT * FROM "user" WHERE id = $1';
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBUser) : null;
}

async find(data: Partial<DBUser>): Promise<DBUser | null> {
  return this.findByFields(data, false);
}

async findAll(data: Partial<DBUser>): Promise<DBUser[]> {
  return this.findByFields(data, true);
}
```

Code 17. Implementation of CRUD for users in SQL.

```
async create(data: DBCreateChat): Promise<DBChat> {
  const query =
    'INSERT INTO chat (name, "ownerId") VALUES ($1, $2) RETURNING *';
  const values = [data.name, data.ownerId];
  const result = await this.pool.query(query, values);
  return result.rows[0] as DBChat;
}

async delete(id: string): Promise<DBChat | null> {
  const query = "DELETE FROM chat WHERE id = $1 RETURNING *";
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBChat) : null;
}

async get(id: string): Promise<DBChat | null> {
  const query = "SELECT * FROM chat WHERE id = $1";
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBChat) : null;
}

async find(data: Partial<DBChat>): Promise<DBChat | null> {
  return this.findByFields(data);
}

async findAll(data: Partial<DBChat>): Promise<DBChat[]> {
  return this.findByFields(data, true);
}

private async findByFields<T extends (DBChat | null) | DBChat[]>(
  data: Partial<DBChat>,
  all: boolean = false,
): Promise<T> {
  const fields: string[] = [];
  const values: unknown[] = [];

  Object.keys(data).forEach((key, index) => {
    fields.push(`"${key}" = $${index + 1}`);
    values.push(data[key as keyof DBChat]);
  });

  const whereClause =
    fields.length > 0 ? `WHERE ${fields.join(" AND ")}` : "";
  const query = `SELECT *
                    FROM chat ${whereClause}`;
```

Code 18. Implementation of CRUD for Chat in SQL

```
    'INSERT INTO message ("chatId", type, message) VALUES ($1, $2, $3) RETURNING *';
  const values = [data.chatId, data.type, data.message];
  const result = await this.pool.query(query, values);
  return result.rows[0] as DBMessage;
}

async delete(id: string): Promise<DBMessage | null> {
  const query = "DELETE FROM message WHERE id = $1 RETURNING *";
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBMessage) : null;
}

async get(id: string): Promise<DBMessage | null> {
  const query = "SELECT * FROM message WHERE id = $1";
  const values = [id];
  const result = await this.pool.query(query, values);
  return result.rowCount ?? 0 > 0 ? (result.rows[0] as DBMessage) : null;
}

async find(data: Partial<DBMessage>): Promise<DBMessage | null> {
  return this.findByFields(data);
}

async findAll(data: Partial<DBMessage>): Promise<DBMessage[]> {
  return this.findByFields(data, true);
}

private async findByFields<T extends (DBMessage | null) | DBMessage[]>(
  data: Partial<DBMessage>,
  all: boolean = false,
): Promise<T> {
  const fields: string[] = [];
  const values: unknown[] = [];

  Object.keys(data).forEach((key, index) => {
    fields.push(`"${key}" = $${index + 1}`);
    values.push(data[key as keyof DBMessage]);
  });
```

Code 19. Implementation of CRUD for message in SQL.

5.5 Object-relational Mappings (ORM)

Object-relational mapping (ORM) represents a framework that enables developers to interact with relational databases through an object-oriented approach. Rather than manually constructing SQL queries, ORM instruments correlate database tables to objects within the programming languages of the

application, such as JavaScript, Python, or Java. This abstraction facilitates developers' interactions with database records as though they were standard objects, thereby rendering data manipulation and retrieval processes more intuitive. By bridging the divide between relational databases and object-oriented programming, ORM streamlines database interactions while preserving consistency within the application’s code architecture. (Source: *ReviewNprep.)*

The employment of ORM instead of conventional SQL presents numerous advantages, thereby enhancing the efficiency of database management. It amplifies productivity by automating the generation of queries, diminishing the necessity for repetitive SQL code, and enabling developers to concentrate on the fundamental logic of the application. ORM additionally augments code readability and maintainability by encapsulating database interactions within objects. Furthermore, it bolsters security through the utilization of parameterized queries, thereby mitigating the risks associated with SQL injection. Moreover, numerous ORM frameworks accommodate multiple database systems, simplifying the transition between different databases without necessitating substantial alterations in code. Although ORM may impose certain performance overhead and complexity for straightforward queries, the advantages it provides surpass these limitations for applications necessitating frequent and structured database operations. (Source: ReniewNprep)

Following the research from (TiDB, 2024), the author of this thesis has chosen Prisma as the ORM of choice. Prisma represents a TypeScript-centric ORM framework that presents a sophisticated and effective methodology for database administration, with a pronounced emphasis on type safety and the enhancement of the developer experience. A principal benefit of this framework is its robust type system, which autonomously generates types corresponding to database models, thereby diminishing the probability of runtime errors and ensuring a more predictable interaction with the database. Furthermore, Prisma improves the developer experience through functionalities such as intelligent autocompletion and an all-encompassing query builder, which facilitate database operations and enhance overall productivity. Another significant feature is its capabilities in

migration management, which optimizes the process of implementing and monitoring alterations to the database schema via Prisma Migrate.

```
generator client {
  provider = "prisma-client-js"
}

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

model User {
  id        String   @id @default(uuid())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  name      String   @db.VarChar(500)
  email     String   @unique @db.VarChar(200)
  password  String   @db.VarChar(500)
  chats     Chat[]
}

model Chat {
  id        String     @id @default(uuid())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  ownerId   String
  name      String   @db.VarChar(1000)
  owner     User     @relation(fields: [ownerId], references: [id])
  messages  Message[]
}

model Message {
  id        String     @id @default(uuid())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  chatId    String
  type      String   @db.VarChar(100)
  message   String   @db.Text
  chat      Chat     @relation(fields: [chatId], references: [id])
```

Figure 3. Prisma Schema for ORM.

After the setup, the database is ready for interaction. The next step would be implementing the methods to interact with the database via Prisma.

```
export class UserDBResource implements IDatabaseResource<DBUser, DBCreateUser> {
  prisma: PrismaClient;

  constructor(prisma: PrismaClient) {
    this.prisma = prisma;
  }

  async create(data: DBCreateUser): Promise<DBUser> {
    const user = await this.prisma.user.create({
      data: { ...data },
    });
    return user as DBUser;
  }

  async delete(id: string): Promise<DBUser | null> {
    const user = await this.prisma.user.delete({ where: { id: id } });
    return user as DBUser;
  }

  async get(id: string): Promise<DBUser | null> {
    const user = await this.prisma.user.findFirst({ where: { id: id } });
    return user as DBUser | null;
  }

  async find(data: Partial<DBUser>): Promise<DBUser | null> {
    const user = await this.prisma.user.findFirst({ where: { ...data } });
    return user as DBUser | null;
  }

  async findAll(data: Partial<DBUser>): Promise<DBUser[]> {
    const users = await this.prisma.user.findMany({ where: { ...data } });
    return users as DBUser[];
  }

  async update(id: string, data: Partial<DBUser>): Promise<DBUser | null> {
    const updateUser = await this.prisma.user.update({
      where: {
        id,
      },
      data,
    });
    return updateUser as DBUser | null;
  }
}
```

Code 20. class User.

```
export class ChatDBResource implements IDatabaseResource<DBChat, DBCreateChat> {
  prisma: PrismaClient;

  constructor(prisma: PrismaClient) {
    this.prisma = prisma;
  }

  async create(data: DBCreateChat): Promise<DBChat> {
    const chat = await this.prisma.chat.create({
      data: { ...data },
    });
    return chat;
  }

  async delete(id: string): Promise<DBChat | null> {
    const chat = await this.prisma.chat.delete({ where: { id: id } });
    return chat;
  }

  async get(id: string): Promise<DBChat | null> {
    const chat = await this.prisma.chat.findFirst({ where: { id: id } });
    return chat;
  }

  async find(data: Partial<DBChat>): Promise<DBChat | null> {
    const chat = await this.prisma.chat.findFirst({ where: { ...data } });
    return chat;
  }

  async findAll(data: Partial<DBChat>): Promise<DBChat[]> {
    const chats = await this.prisma.chat.findMany({ where: { ...data } });
    return chats;
  }

  async update(id: string, data: Partial<DBChat>): Promise<DBChat | null> {
    const updatedChat = await this.prisma.chat.update({
      where: {
        id,
      },
      data,
    });
    return updatedChat;
  }
}
```

Code 21. class Chat

```
export class MessageDBResource
  implements IDatabaseResource<DBMessage, DBCreateMessage>
{
  prisma: PrismaClient;

  constructor(prisma: PrismaClient) {
    this.prisma = prisma;
  }

  async create(data: DBCreateMessage): Promise<DBMessage> {
    const message = await this.prisma.message.create({
      data: { ...data },
    });
    return message as DBMessage;
  }

  async delete(id: string): Promise<DBMessage | null> {
    const message = await this.prisma.message.delete({ where: { id: id } });
    return message as DBMessage | null;
  }

  async get(id: string): Promise<DBMessage | null> {
    const message = await this.prisma.message.findFirst({
      where: { id: id },
    });
    return message as DBMessage | null;
  }

  async find(data: Partial<DBMessage>): Promise<DBMessage | null> {
    const message = await this.prisma.message.findFirst({
      where: { ...data },
    });
    return message as DBMessage | null;
  }

  async findAll(data: Partial<DBMessage>): Promise<DBMessage[]> {
    const messages = await this.prisma.message.findMany({
      where: { ...data },
    });
    return messages as DBMessage[];
  }

  async update(…
  }
}
```

Code 22. class Message

6 Front-end Development

After setting up the back-end logic, API, and database, the application can interact with them. It is crucial to have a user interface (UI) to engage with the application. The users of this application are mostly individuals who lack experience (UX) with software development, so an intuitive, simple, and friendly user experience is essential. It can consist of a standard API, a component on the website, or the entire application interface itself. In the scope of this thesis, a simple interface utilizing web technology will be implemented. The most popular method for creating front-end components on the web currently is React, according to Miikka, O. (2021). However, the author wants to try experimentation, Svelte will be used as the front-end library.

6.1 Svelte

Svelte is a modern compiler-based framework for building fast and efficient web applications. Unlike traditional UI frameworks that rely on a virtual DOM, Svelte compiles components into optimized JavaScript at build time, resulting in smaller bundles and better runtime performance. Svelte was created by Rich Harris, with the first stable release in 2016. Harris was a graphics editor at The New York Times, and he designed Svelte to address the complexities and performance bottlenecks inherent in some existing JavaScript frameworks like React. More information about Svelte is shown in the Svelte Documentation by Annon (2024).

Figure 4 shows how Svelte code works under the hood. A single Svelte component will be written in a single file ".svelte" that contains 3 main parts: HTML, CSS, and JavaScript. At compile time, the Svelte compiler analyzes the component above and uses special tools, which are:

- HTML Parser: Extracts markup, directives (e.g., {#if}), and dynamic bindings.

- CSS Parser (css-tree): Processes scoped styles and generates optimized CSS.
- JavaScript Parser (acorn): Parses scripts, detects reactive declarations ($:), and tracks state dependencies.

The parsed input is converted into an intermediate representation (AST) that structures the component as: Root, HTML, CSS, and JS. After that, the compiler removes unused code, creates JS code that updates the DOM directly, and avoids using hooks for reactivity and events.

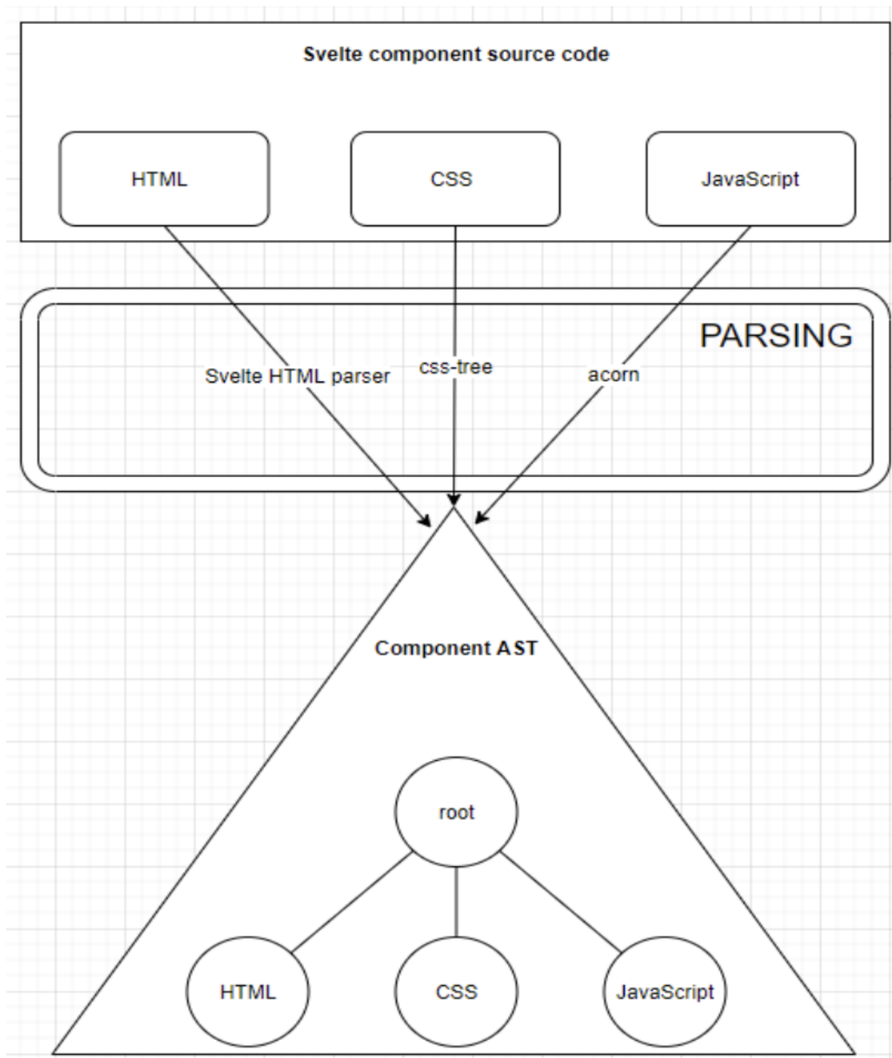


Figure 4. Svelte's transpiler generates an AST from a component (Source: Miikka, O. (2021).

6.2 Vite

Modern front-end development demands tools that balance performance with simplicity, which is precisely why this project adopts Vite as its build tool and development server. Unlike traditional bundlers like Webpack, as stated in the Vite Documentation, Vite leverages native ES modules to deliver near-instantaneous server starts and Hot Module Replacement (HMR). This means developers see changes reflected in the browser the moment they save a file, without full-page reloads or lengthy rebuilds. For Svelte applications, where reactivity and component-driven architecture are central, this rapid feedback loop is invaluable. Vite's plugin-based ecosystem also simplifies integration with Svelte, TypeScript, and CSS preprocessors, reducing configuration overhead while maintaining flexibility.

The decision to pair Vite with Svelte, rather than SvelteKit, comes from the project's focus on building a lightweight Single-Page Application (SPA). While SvelteKit offers powerful features like server-side rendering (SSR) and file-based routing, these add unnecessary complexity for an interactive chat application that operates primarily on the client side. Vite's streamlined approach avoids the boilerplate of SSR hydration or static site generation, instead optimizing for fast production builds through Rollup-powered code splitting and tree-shaking. This aligns perfectly with the project's goals: a performant SPA with minimal tooling friction. According to SvelteJS itself, Vite treats TypeScript code in `.svelte` as JavaScript, so there would be no drawback from fast development and deployment.

The project structure further reflects this philosophy of simplicity. The public folder houses static assets like fonts and favicons, while the src/lib directory organizes reusable Svelte components, promoting modularity. Global styles are centralized in app.css, ensuring visual consistency, and TypeScript support is enhanced via vite-env.d.ts, which defines environment variables. This setup shows how Vite and Svelte together enable a clean, maintainable codebase free from the complexity of meta-frameworks like SvelteKit.

Ultimately, the choice of Vite over SvelteKit lies in use-case alignment. According to Matthias Andrasch (2022), SvelteKit excels for SEO-driven applications requiring SSR or static rendering, but its feature set would introduce unnecessary overhead for this project. Vite's focus on SPAs, combined with Svelte's compiled reactivity, delivers a faster development cycle and leaner production bundle. This synergy exemplifies how modern tooling can be tailored to a project's specific needs, prioritizing developer experience and end-user performance without compromising scalability.

6.3 Axios

Routing and API communication form the backbone of modern web applications, and the Svelte implementation carefully balances simplicity with functionality. For navigation, svelte-routing is adopted, a lightweight solution that provides essential routing capabilities without the overhead of larger frameworks as SvelteKit. The system revolves around a central <Router> component that manages the application's navigation state, with individual <Route> components defining specific paths. This setup allows developers to maintain a clean separation between views while enabling dynamic parameter handling, such as rendering the <Chat> component with either a null chatId at the root path or extracting the ID from URLs matching the /:id pattern.

For back-end communication, the author chooses Axios as the HTTP client, a decision that significantly simplifies the API interaction layer. Unlike the native fetch API, Axios provides a more developer-friendly interface with built-in features that would otherwise require extensive boilerplate. The library's interceptor system allows the developers to centrally manage authentication tokens and error responses, while its cancellation feature helps prevent memory leaks from abandoned requests. The author chose to implement Axios with async/await patterns throughout the application, creating API calls that are both readable and maintainable, such as the chat message fetching endpoint, which cleanly handles various response states.

The integration between the routing system and Axios creates a seamless development experience. When users navigate to different routes, components automatically trigger the appropriate API calls through Axios, with URL parameters seamlessly feeding into request parameters. The author has enhanced this further with response interceptors that handle common scenarios like authentication failures by redirecting to login routes. This tight coupling between navigation and data fetching eliminates common pain points while keeping the code organized and predictable.

The technical choices reflect a deliberate focus on simplicity and maintainability. While alternatives like SvelteKit's built-in routing or the fetch API exist, the author prioritized solutions that offer just enough functionality without introducing unnecessary complexity. The svelte-routing/Axios combination proves particularly effective for the mid-sized SPA, providing all the capabilities the author needs while maintaining excellent performance. This approach also keeps the architecture flexible, and should requirements change, the author can easily swap individual components without major refactoring, thanks to the clean separation of concerns between routing and API communication layers.

6.4 Authentication Component

```
<div class="auth-container">
  <form on:submit|preventDefault={register} class="auth-form">
    <div class="form-header">
      <h2>Create Account</h2>
    </div>
    {#if errorMessage}
      <div class="error">{errorMessage}</div>
    {/if}
    <div class="input-group">
      <input type="text" placeholder="Name" bind:value={name} required />
    </div>
    <div class="input-group">
      <input type="email" placeholder="Email" bind:value={email} required />
    </div>
    <div class="input-group">
      <input
        type="password"
        placeholder="Password"
        bind:value={password}
        required
      />
    </div>
    <div class="action-group">
      <button type="submit" class="auth-btn" disabled={!formValid}
        >Sign Up</button
      >
    </div>
    <div class="switch-auth">
      Already have an account? <a href="/login">Sign in here.</a>
    </div>
  </form>
</div>
```

Code 23. Register Component (HTML)

```
onMount(() => {
  if ($authToken) {
    navigate("/");
  }
});

async function register() {
  try {
    await axios.post(`${API_HOST}/api/v1/auth/register/`, {
      name,
      email,
      password,
    });
    navigate("/login");
  } catch (error) {
    const defaultError = "An unexpected error occurred";
    if (axios.isAxiosError(error) && error.response) {
      const errorSlug = error?.response?.data?.error;
      switch (errorSlug) {
        case "ERROR_USER_ALREADY_EXIST":
          errorMessage = "User already exists, try logging in instead";
          break;
        default:
          errorMessage = defaultError;
      }
    } else {
      errorMessage = defaultError;
    }
  }
}
```

Code 24. Register function.

```
<div class="auth-container">
  <form on:submit|preventDefault={login} class="auth-form">
    <div class="form-header">
      <h2>Login</h2>
    </div>
    {#if errorMessage}
      <div class="error">{errorMessage}</div>
    {/if}
    <div class="input-group">
      <input type="email" placeholder="Email" bind:value={email} required />
    </div>
    <div class="input-group">
      <input
        type="password"
        placeholder="Password"
        bind:value={password}
        required
      />
    </div>
    <div class="action-group">
      <button type="submit" class="auth-btn" disabled={!formValid}
        >Sign in</button
      >
    </div>
    <div class="switch-auth">
      Don't have an account? <a href="/register">Register here</a>.
    </div>
  </form>
</div>
```

Code 25. Login Component

```
async function login() {
  try {
    const response = await axios.post(`${API_HOST}/api/v1/auth/login/`, {
      email,
      password,
    });
    authToken.set(response.data?.token);
    navigate("/");
  } catch (error) {
    const defaultError = "An unexpected error occurred";
    if (axios.isAxiosError(error) && error.response) {
      const errorSlug = error?.response?.data?.error;
      switch (errorSlug) {
        case "INVALID_CREDENTIALS":
          errorMessage = "Invalid email or password";
          break;
        default:
          errorMessage = defaultError;
      }
    } else {
      errorMessage = defaultError;
    }
  }
}
/script>
```

Code 26. Login function.

From code 23 to code 27 is how the Login and Register components are implemented in the codebase. The author uses a Svelte store (`authToken`) to centrally manage JWT tokens, ensuring persistence in `localStorage`, automatic

axios header configuration, and JWT decoding for user data. Both the login and registration flows feature real-time validation, typed axios error handling (with API errors mapped to user-friendly messages), and redirection. Routes are protected via `onMount` checks, forms are reactive (disabled buttons until valid), and navigation links switch seamlessly between auth pages.

This approach achieves clean, maintainable code by separating concerns: the auth store handles token logic, while components focus on UI, adhering to SOLID and DRY principles. Security leverages `localStorage` and axios headers, and the UX is elevated through reactive forms, inline errors, and persistent sessions. By decoupling logic from presentation, the system remains extensible (e.g., for OAuth) without compromising simplicity or user experience.

```
import { writable } from "svelte/store";
import axios from "axios";
import { jwtDecode } from "jwt-decode";

interface TokenPayload {
  name: string;
}

function setAxiosAuth(token: string) {
  axios.defaults.headers.common["Authorization"] = `Bearer ${token}`;
}

function createAuthStore() {
  const token = localStorage.getItem("authToken");
  if (token) {
    setAxiosAuth(token);
  }
  const { subscribe, set } = writable<string | null>(token);

  return {
    subscribe,
    set: (value: string) => {
      localStorage.setItem("authToken", value);
      set(value);
      if (value) {
        setAxiosAuth(value);
      } else {
        delete axios.defaults.headers.common["Authorization"];
      }
    },
    remove: () => {
      localStorage.removeItem("authToken");
      set(null);
      delete axios.defaults.headers.common["Authorization"];
    },
    getPayload: () => {
      const token = localStorage.getItem("authToken");
      if (token) {
        const decoded: TokenPayload = jwtDecode(token);
        return decoded;
      }
      return null;
    },
  };
}

export const authToken = createAuthStore();
```

Code 27. Authentication store.

6.5 Chat Component

```
<script lang="ts">
  import axios from "axios";
  import { API_HOST } from "../constants";
  import "../styles/chatPopup.css";
  export let onCreate: (newChatId: string) => void;
  export let onClose: () => void;

  let chatName = "";
  let errorMessage: string | null = null;

  async function createChat() {
    try {
      const response = await axios.post(`${API_HOST}/api/v1/chat/`, {
        name: chatName,
      });
      onCreate(response.data.data.id);
    } catch (error) {
      console.error("Error creating chat:", error);
      errorMessage = "Failed to create chat. Please try again later.";
    }
  }
</script>
```

Code 28. create Chat function.

Codes 28 to 30 are the implementation of the Chat component. The chat system consists of three key parts: chat creation, list display, and message handling, all implemented with clean separation of concerns. The `createChat` component handles new chat creation through a simple form that posts to the API and invokes a callback to update the chat list. The main chat list view fetches and displays available chats on mount, with navigation to selected chats using Svelte's routing. The chat detail view manages message loading and submission, featuring real-time updates through reactive statements ($:) when the `chatId` changes. State management prevents duplicate submissions and improves perceived performance. The code is easily extensible; for example, WebSocket integration could easily be added for real-time messaging while maintaining the existing component structure.

```
<script lang="ts">
  import { onMount } from "svelte";
  import axios from "axios";

  import { navigate } from "svelte-routing";
  import CreateChatPopup from "./CreateChatPopup.svelte";
  import { API_HOST } from "../constants";
  import "../styles/chatList.css";

  let chats: { id: string; name: string }[] = [];
  let errorMessage: string | null = null;
  export let chatId: string | null;

  async function getData() {
    try {
      const response = await axios.get(`${API_HOST}/api/v1/chat/`);
      chats = response.data.data;
    } catch (error) {
      console.error("Error fetching chats:", error);
      errorMessage = "Failed to fetch chats. Please try again later.";
    }
  }

  onMount(async () => {
    await getData();
  });

  let isCreatingNewChat = false;

  function selectChat(chatId: string) {
    navigate(`/${chatId}`);
  }

  function createNewChat() {
    isCreatingNewChat = true;
  }

  async function onCreate(newChatId: string) {
    onClose();
    navigate(`/${newChatId}`);
    await getData();
  }

  function onClose() {
    isCreatingNewChat = false;
  }
</script>
```

Code 29. Show Chat List function.

```
<script lang="ts">
  import { onMount } from "svelte";
  import axios from "axios";
  import { API_HOST } from "../constants";
  import "../styles/chatDetails.css";
  export let chatId: string;
  let messages: { id: string; message: string; createdAt: number }[] = [];
  let newMessage = "";
  let errorMessage: string | null = null;
  let isLoading = false;

  onMount(async () => {
    await loadMessages();
  });

  async function loadMessages() {
    try {
      const response = await axios.get(
        `${API_HOST}/api/v1/chat/${chatId}/message/`,
      );
      messages = response.data.data;
    } catch (error) {
      errorMessage = "Failed to get chat details. Please try again later.";
      console.error("Error fetching messages:", error);
    }
  }

  async function sendMessage() {
    isLoading = true;
    try {
      const response = await axios.post(
        `${API_HOST}/api/v1/chat/${chatId}/message/`,
        { message: newMessage },
      );
      messages = [
        ...messages,
        { message: newMessage, createdAt: Date.now() },
        response.data.data,
      ];
      newMessage = "";
    } catch (error) {
      errorMessage = "Failed to send message. Please try again later.";
      console.error("Error sending message:", error);
    }
    isLoading = false;
  }
  $: {
    if (chatId) {
      loadMessages();
    }
  }
</script>
```

Code 30. get Chat Details function.

7 Conclusion

This thesis explored the development of a full-stack web application using Bun and TypeScript, focusing on the back-end system with experimental tools including the Bun framework, an isolated PostgreSQL database managed via Docker and user interface handling with Svelte. The objective of this thesis was achieved, which was to evaluate the production readiness of emerging technologies like Bun as a TypeScript-powered back-end runtime and Svelte as a front-end framework, while adhering to modern software development principles such as SOLID and DRY. In theory, those principles look straightforward, but to implement in real code, it is a different story, since there are so many ways to interpret how those principles apply to some specific scenario or whether the code was following all the principles. Sometimes, rigidly following the principles of SOLID and DRY in every part might lead to an inflexible code base. After all, it requires the developer's experience to decide which actions need to be taken. However, the project successfully followed the MVC pattern, utilizing Bun and Hono for the back-end, PostgreSQL for data storage, and Svelte for a reactive user interface. TypeScript was used as the programming language to link and route the model, view and controller together. The author also successfully defined all 'type' in the codebase, without relying on type "any".

In the process of writing this thesis and developing features for this single-page web application, the author has successfully implemented core functionalities, including user authentication, chat creation, and message handling via RESTful APIs. As expected, and explained in Chapter 2, due to time constraints, some features remained incomplete and not fully tested. The back-end API integration is lacking, and it needs to be implemented as soon as possible to ensure a seamless communication experience between users and servers. Due to the unique approach in this project, using an AI API is crucial for the functionality of this web application. Additionally, the front-end user interface lacks advanced features such as responsive design optimizations, accessibility improvements, and robust user interactions. The author of this thesis will continue working on

the lacking features in the future to push the project to a state that will be production-ready.

One key lesson from this project is the importance of strict scoping and prioritizing a Minimum Viable Product (MVP) before expanding features. A concrete planning method should be used throughout the development cycle, as having small milestones will keep the project progressing forward. Without these milestones, the developers might lose focus and wander without making meaningful progress toward their objectives. Testing proved indispensable, as early bug detection significantly accelerated development and improved code reliability. That is why it is worth considering TDD in combination with Scrum, which is a widely used agile framework that helps teams, especially in software development, organize their work and deliver value incrementally. While Bun demonstrated impressive performance benefits, thoughtful utilities, including all necessary tools for developers in the working process, its ecosystem is still maturing. As the author was encountering unresolved bugs, it suggests that it may not yet be fully production-ready. Similarly, Svelte offered a streamlined development experience but required careful consideration for complex state management. The documentation and community support for Svelte is somewhat lacking, as not as comparable to popular libraries such as React.

In conclusion, this thesis can serve as a practical reference for developers considering Bun and Svelte for full-stack projects. Future work will focus on implementing the API connect to LLM, refining front-end accessibility such as a11y, i18n, front-end testing and working with Bun’s team to improve the stability of the runtime and make sure all the Bun update in the future will be backward compatible with the project, because it takes a year to develop a fully worked production ready service. The experience throughout this thesis reinforced that modern web development demands a balancing between innovation and reliability, choosing the right tools while maintaining rigorous testing and incremental development. Despite challenges, both Bun and Svelte show significant potential in the future, in the author’s opinion, and it is beneficial to have experience early with them.

References

ahilsenbeck (2024). *The Impact of Artificial Intelligence on the Job Search - INSPYR Solutions*. [online] INSPYR Solutions. Available at: https://www.inspyrsolutions.com/impact-of-artificial-intelligence-on-job-search/ [Accessed 20 November 2024].

Ahmod, Md F. (2023). Javascript Runtime Performance analysis: Node and Bun. *Master's thesis – Tampere University.* Available at: https://trepo.tuni.fi/handle/10024/149672 [Accessed 21 October 2024].

Amazon Web Services (2024). *What Is an API? - API Beginner's Guide - AWS*. [online] Amazon Web Services, Inc. Available at: https://aws.amazon.com/what-is/api/. [Accessed 21 May 2025].

Andrasch, M. (2022). Rich Harris explains why SvelteKit pushes for Server Side Rendering (and against SPA/CSP). [online] DEV Community. Available at: https://dev.to/mandrasch/rich-harris-explains-why-sveltekit-pushes-for-server-side-rendering-and-against-spa-5flj [Accessed 21 May 2025].

Andreeva, O. (2024). *Top 10 Web Application Vulnerabilities in 2021–2023*. [online] securelist.com. Available at: https://securelist.com/top-10-web-app-vulnerabilities/112144/. [Accessed 08 March 2025].

Annon. (2024). *Welcome to Svelte • Svelte Tutorial*. [online] Svelte.dev. Available at: https://svelte.dev/tutorial/svelte/welcome-to-svelte. [Accessed 21 May 2025].

Auth0 (2024). *JSON Web Tokens*. [online] Auth0 Docs. Available at: https://auth0.com/docs/secure/tokens/json-web-tokens. [Accessed 21 May 2025].

Betterstackhq. (2016). *Node.js Vs Deno Vs Bun: Comparing JavaScript Runtimes | Better Stack Community*. [online] Available at: https://betterstack.com/community/guides/scaling-nodejs/nodejs-vs-deno-vs-bun/. [Accessed 21 May 2025].

Blackduck (2023). *What Is Web Application Security and How Does It Work? | Black Duck*. [online] Blackduck.com. Available at:

https://www.blackduck.com/glossary/what-is-web-application-security.html. [Accessed 08 March 2025].

Blum, R. and Singh, R. (2024). *Google - Site Reliability Engineering*. [online] sre.google. Available at: https://sre.google/sre-book/data-integrity/. [Accessed 16 May 2025].

Brayden G. (2021). *Why SvelteKit Uses Vite Instead of Snowpack*. [online] YouTube. Available at: https://www.youtube.com/watch?v=tUXqcrHrGJk [Accessed 27 April 2025].

Bun.sh. (2025). *Bun Testing Practices*. [online] Available at: https://bun.sh/guides/test/testing-library [Accessed 26 January 2025].

Cherny, B. (2019). *Programming TypeScript*. O'Reilly Media.

Chowdhury, M.S.N. (2024). Best Practices for Securing MERN Stack Applications: A Comprehensive Study of Authentication, Authorization and Data Protection. *Bachelor's Thesis – Jyväskylä University of Applied Sciences.* Available at: http://www.theseus.fi/handle/10024/876758. [Accessed 3 June 2025].

Codecademy (2023). *Back-End Web Architecture*. [online] Codecademy. Available at: https://www.codecademy.com/article/back-end-architecture. [Accessed 06 October 2024].

Colandrea, D. (2023). *Level up Your TypeScript Projects: Discover the Power of ESLint and Prettier*. [online] DEV Community. Available at: https://dev.to/_domenicocolandrea/linting-in-typescript-using-eslint-and-prettier-5f44 [Accessed 23 May 2025].

Docker Documentation. (2020). *Docker Documentation*. [online] Available at: https://docs.docker.com. [Accessed 23 May 2025].

Fenton, S. (2017). *Pro TypeScript*. Apress.

Google Cloud (2024a). *What Is a Relational Database*? [online] Google Cloud. Available at: https://cloud.google.com/learn/what-is-a-relational-database. [Accessed 24 May 2025].

Google Cloud. (2024b). *What Is PostgreSQL? Databases Explained*. [online] Available at: https://cloud.google.com/discover/what-is-postgresql. [Accessed 23 May 2025].

Hono.dev. (2024). *Hono - Ultrafast Web Framework for the Edges*. [online] Available at: https://hono.dev/docs/. [Accessed 21 February 2025].

IBM (2021). *REST APIs*. [online] Ibm.com. Available at: https://www.ibm.com/think/topics/rest-apis. [Accessed 01 May 2025].

Jongmans, E., Jeannot, F., Liang, L. and Damperat, M. (2022). *Impact of website visual design on user experience and website evaluation: the sequential mediating roles...* ResearchGate - Journal of Marketing Management. 38. 1-36. 10.1080/0267257X.2022.2085315.

katalon.com. (2025). *What is Web Testing? Definition, Tools, Best Practice*. [online] Available at: https://katalon.com/resources-center/blog/what-is-web-testing. [Accessed 21 May 2025].

Le, D.A. (2023). E-Commercial Full Stack Web Application Development: with React, Redux, NodeJS, and MongoDB. *Bachelor's thesis - Vaasa University of Applied Sciences*. Available at: http://www.theseus.fi/handle/10024/802860. [Accessed 03 January 2025].

Levy, J.-J. (2023). *Principles of Software Development: SOLID, DRY, KISS, and More*. [online] Scalastic.io. Available at: https://scalastic.io/en/solid-dry-kiss/. [Accessed 21 May 2025].

MariaDB (2018). *ACID Compliance: What It Means and Why You Should Care*. [online] MariaDB. Available at: https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/. [Accessed 21 May 2025].

MDN Web Docs. (2023). *MVC Architecture*. [online] Available at: https://developer.mozilla.org/en-US/docs/Glossary/MVC. [Accessed 21 May 2025].

Microsoft (n.d.). *Documentation - TypeScript for the New Programmer*. [online] www.typescriptlang.org. Available at: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 21 May 2025].

Oksanen, M. (2021). *Cross-platform UI Development: React Vs Svelte*. [online] bachelor's thesis – Metropolia University of Applied Sciences. Available at: https://www.theseus.fi/bitstream/handle/10024/500643/Oksanen_Miikka.pdf. [Accessed 09 April 2025].

Packt. (2025). *Full-Stack Web Development with TypeScript 5 | Web Development | Paperback*. [online] Available at: https://www.packtpub.com/en-gb/product/full-stack-web-development-with-typescript-5-9781835885581 [Accessed 23 May 2025].

PostgreSQL (2020). *1. What Is PostgreSQL?* [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/intro-whatis.html. [Accessed 08 May 2025].

European Labour Authority (ELA). 2021 *Recruiting in Europe: A guide for employers.* [pdf] Bratislava: European Labour Authority. Available at: https://www.ela.europa.eu/sites/default/files/2024-02/Recruiting_in_Europe_2021.pdf [Accessed 3 June 2025]

reviewNprep (2024). *Mapping ORM beginner guide*. [online] ReviewNPrep. Available at: https://reviewnprep.com/blog/object-relational-mapping-orm-a-beginners-guide/ [Accessed 21 May 2025].

Sumner, J. (2025). *What Is Bun? | Bun Docs*. [online] Bun. Available at: https://bun.sh/docs. [Accessed 21 May 2025].

sveltejs (2024). *Vite is treating TypeScript code in svelte files as JavaScript on* GitHub. Available at: https://github.com/sveltejs/vite-plugin-svelte/issues/928 [Accessed 24 May 2025].

TiDB Team. (2024). *Understanding Prisma ORM*. [online] Available at: https://www.pingcap.com/article/understanding-prisma-orm/. [Accessed 22 May 2025].

Watson, D. (2024). *JavaScript and TypeScript Trends 2024: Insights from the Developer Ecosystem Survey | The WebStorm Blog*. [online] The JetBrains Blog. Available at: https://blog.jetbrains.com/webstorm/2024/02/js-and-ts-trends-2024/. [Accessed 21 May 2025].

Zakas, N.C. (2024). *Core Concepts - ESLint - Pluggable JavaScript Linter*. [online] Eslint.org. Available at: https://eslint.org/docs/latest/use/core-concepts/. [Accessed 21 May 2025].

Declaration of AI usage in Thesis Preparation

Following modern academic practices, the author wishes to transparently disclose the use of Large Language Model (LLM) technology during the preparation of this thesis. While AI tools have become increasingly prevalent in research and writing, the author has implemented strict protocols to ensure the integrity and originality of the author’s work remains uncompromised.

The writing process followed the ethical guidelines and practices of Turku AMK at every stage. The first draft, including all original research, technical implementations, and core arguments, was entirely the author’s work, created without any AI-generated content to preserve the author’s authentic thought process and academic voice. Only after finalizing the independent draft, the author employed several LLM tools, and their role was strictly limited to providing structural, stylistic and wording suggestions. Every AI-generated recommendation underwent thorough review, with me evaluating each suggestion against four critical criteria: consistency with the author’s original meaning, technical accuracy, appropriateness for academic context, and alignment with the author’s writing style. Throughout this process, the author has final authority over all content decisions on every modification or inclusion, ensuring the author maintains complete understanding and ownership of the final work in its entirety.

Specific applications of AI assistance included:

A. Structural Clarity

- Breaking long paragraphs into focused sections.
- Adding transitional phrases to improve flow between ideas
- Reorganizing sentences within paragraphs for better coherence

B. Language Precision

- Grammar fixes: Correcting verb tenses, subject-verb agreement, and article usage.
- Replacing informal phrasing: for example, "Those frameworks will be big" to "These technologies show significant potential"

C. Technical Readability

- Ensuring consistent terminology (e.g., standardizing "backend" → "back-end" per academic style guides)
- Trimming redundant explanations (e.g., removing repetitive definitions of MVC in multiple chapters or Advantages and Disadvantages of certain technologies)

This approach follows current best practices in academic writing, where AI serves as an editorial tool rather than a content originator. The fundamental research, analysis, and conclusions remain entirely the author’s work, with AI assistance limited to the refinement of expression and presentation.

By disclosing this process, the author affirms that while the author benefited from AI-powered language tools, the intellectual substance and academic rigor of this thesis represent the author’s original research and understanding. All concepts, implementations, and conclusions were thoroughly vetted and approved through the author’s expertise and knowledge.