Tampere University of Applied Sciences

# Design and Development of the Backend Web Architecture for Knowledge Marketplace Mobile App

Roman Zinkevich

BACHELOR'S THESIS
March 2026

Bachelor's Degree Programme in Software Engineering

## ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Software Engineer

ZINKEVICH, ROMAN:
Design and Development of the Backend Web Architecture for Knowledge Marketplace Mobile App

Bachelor's thesis 52 pages
March 2026

---

This thesis documents the design and implementation of the backend infrastructure for TutorSwap, a peer-to-peer knowledge marketplace designed to facilitate skill exchange among university students. The platform addresses the "Double Coincidence of Wants" in traditional barter systems by implementing a time-banking credit system, where one minute of teaching earns one credit that can be spent on learning other skills.

The technical objective was to establish a scalable, secure, and maintainable architectural foundation. The chosen technology stack includes Java with the Spring Boot framework, PostgreSQL for relational data integrity, and Docker for containerized deployment. A monolithic layered architecture was selected to balance development speed with system robustness, utilizing JWT (JSON Web Tokens) for stateless authentication and STOMP over WebSockets to enable real-time communication between users.

Key features implemented include a greedy matching engine for tutor discovery, a polymorphic messaging system to handle various data types, and an automated CI/CD pipeline via GitHub Actions. Evaluation of the system through unit and integration testing confirmed that the backend meets performance requirements, with API response times consistently under 400ms. The project concludes that while the current MVP provides a stable foundation, future iterations should explore microservices and dedicated search engines like Elasticsearch to support large-scale user growth.

---

Key words: backend, spring boot, sharing economy, rest api, websockets

**USE OF AI IN THESIS**

I have used AI tools in my thesis:

☐ No

☒ Yes

The AI tools used in my thesis and the purpose of their use has been described below:

Names of the tools and versions: ChatGPT (GPT 5.2), Google Gemini 3

Purpose of the use: Gemini helped summarize the technical implementation and findings into a formal academic abstract. It assisted in searching for relevant academic sources to prove specific points in my thesis. I independently verified every suggested source, checked the original context of the quotes, and manually integrated them into the text to ensure the arguments were my own. Gemini was also used to refine my original text for better clarity and professional expression.
GPT 5.2 was utilized to generate a logical structure for the thesis, helping to define the focus areas ensuring a coherent technical flow. GPT 5.2 also assisted in debugging human-made code, wrote PostgreSQL schema based on my pseudocode and helped in writing boilerplate code if it was not generated by Spring Initializr.

Parts in which AI was used: Language refinement and proofreading were used across all chapters. Similarly, across all chapters, AI was used to identify supporting literature for technical and social claims. AI was used to generate the Table of Contents. Finally, Gemini was used to draft Abstract part of the thesis. Finally, AI was used to help debug and generate database schema and write boilerplate code.

---

I am aware that I am totally responsible for the entire content of the thesis, including the parts generated by AI, and accept the responsibility for any violations of the ethical standards of publications.

**CONTENTS**

**GLOSSARY**

| | |
|---|---|
| TAMK | Tampere University of Applied Sciences |
| cr | credit |

# 1 INTRODUCTION

During my studies at university, I noticed that a lot of students, including myself, are eager to learn new things that are not taught in universities. These things usually are new skills or hobbies, and these are not easy to learn by yourself. More than that, universities often do not teach this, and it makes sense, since you cannot get a teacher for every niche skill that a student may want to learn. Another place where students can look for help is the internet, but the lack of personalization makes this approach weak. "The average tutored student was above 98% of the students in the control class" (Bloom, 1984, p. 4). So, a solid approach to learning new skills would be a “tutor,” someone who already has the skill and can guide you through your journey. There are two options in this case: asking a friend, however, it is not guaranteed that any of your friends have the skill or knowledge you want; or looking for an actual private tutor, but they cost a lot, especially for a student on a budget.

As a student, I felt the need for this personal guidance myself. I wanted to get real feedback on my mistakes to truly master a certain skill, but it was impossible. After some time, I realized that the university is full of people and every one of them knows something that can be taught. The easiest example for me was Finnish people, obviously, knowing Finnish, which I needed to learn myself. And there are people who would want to learn my native language, or learn coding, since I am a software engineer. These skillful people who are already knowledgeable and are also eager to learn something new could help each other out by exchanging knowledge. That is when the idea of a knowledge marketplace came to life.

Knowledge Marketplace took place from the Sharing Economy, “the peer-to-peer-based activity of obtaining, giving, or sharing the access to goods and services, coordinated through community-based online services” (Hamari et al., 2016). The concept of Sharing Economy has become more popular nowadays. Apps like Airbnb and Uber let people rent something instead of fully buying it, e.g., cars and apartments. However, the sharing economy with knowledge

hasn’t been executed yet, even though the desire for knowledge definitely exists (universities and online courses are proof of that), and more than that, knowledge is not a physical thing that is limited, and therefore “duplicating” instead of sharing is a better word for such a marketplace, since you never lose it. On top of that, teaching often helps you understand the topic better; research confirms that the act of explaining “encourages helpers to reorganize and restructure the information in their own minds so they, in turn, develop clearer and more elaborate cognitive understandings” (Gillies, 2016, p. 41), effectively improving your own skill. The project of this thesis, “TutorSwap,” is a platform that makes all this possible without using real money.

Transition from an idea to an actual implementation is where problems start appearing. Of course, frontend/mobile implementation is a complicated issue itself, but I am going to focus on the backend, which was my responsibility. Backend is extremely important; without it, the whole app is just static pictures and texts, and there is no real communication between users. The project was also dealing with the “Double Coincidence of Wants,” a famous barter problem, occurring when two people must simultaneously have what the other wants for a trade to happen. This error existed in the barter system long ago and was fixed with the invention of money, since money is what both parties want; therefore, it is easier to find a middle ground when money is involved. Realizing that, the backend must have had some credit system, which introduces high risks: if the database messes up, people lose their “earned time,” which is not a minor inconvenience and can cause users to lose trust in the monetary system of the app.

There are a few services out there that could be considered like TutorSwap. First, it is all the online courses you can find (e.g., Coursera). Once again, these solutions are great, but they lack personality and proper feedback, which is why they cannot be considered close to the TutorSwap project. Paid platforms are other solutions that come close to TutorSwap, but the whole point was to make our project free. Another niche one is forums. They are great, free, and have personal feedback, but they are messy, and there is no incentive for a person behind the screen to reply to all your questions. After I investigated all these

services, I realized there was nothing close to what I wanted, so I must implement it using my software engineering degree.

The primary objective of this thesis is to design, describe implementation, and evaluate the backend web architecture for the TutorSwap mobile application. The scope of this work is strictly focused on the server-side engineering necessary to support the platform. This includes the design of a relational database schema that accurately models complex relationship between users, skills, and sessions; the development of a RESTful API to serve as the communication layer between the database and the mobile client; and the implementation of core business logic such as user authentication, session scheduling, and the credit exchange mechanism. For the technology stack of this implementation I chose Java with the Spring Boot framework, mainly because of it dependency injection capabilities, and strong security ecosystem, PostgreSQL as the database, because it can ensure data integrity through strict relational constraints, Docker, to containerize the application, ensuring consistency across development and deployment environments.

It is important to define the boundaries of this project. While the backend is designed to support a mobile application, the development of the frontend mobile user interface is outside the scope of this thesis since another person, my co-founder, was responsible for that. The matching logic was implemented to connect students with tutors based on skills; however, advanced machine learning algorithms are not included in this project, since it felt and still feels like an overkill. The focus remained on establishing a solid foundation that ensures scalability, maintainability, and security. The system is designed to handle the core functional requirements of a marketplace, searching and booking, while also covering non-functional requirements such as response time, latency, and secure data handling.

The structure of this thesis is organized to guide the reader through the complete software development lifecycle of the project. Next chapter, for example, provides an analysis of the current state of peer-to-peer learning platforms. In

this chapter I looked at available tutoring platforms on the market and highlighted their pros and cons, proving the need for TutorSwap platform. I also went through existing architectural patterns that could be used for TutorSwap's backend implementation and explained why I chose certain technologies.

Moving to practice, I selected specific engineering tools for development. I provided a technical justification for the choice of Spring Boot, PostgreSQL, and Docker, comparing them briefly against alternatives to demonstrate why they are the optimal choices for this specific use case. I also outlined the development workflow, including version control stratenegies and testing tools.

It was also important to properly plan a project, so I have a bigger picture in my head through the whole development process. The plan will serve later as the blueprint for implementation. There are functional requirements, what the system must do, and non-functional requirements, how the system must perform.

With the architectural blueprint ready, the implementation phase began. Here I did not paste the whole codebase, but described the technical core of the thesis project, introduced the database design, presenting Entity-Relationship diagrams, and schema definitions. I decided not to describe every single API endpoint I got in the backend, but instead just explain the most important modules in TutorSwap. Additionally, I explained the security configuration for authentication and authorization on this platform.

I also had to test it and make sure everything works accordingly. I described the testing methods I used in one of the chapters. There I went through unit testing of individual components and integration testing of the API endpoints. I summed up the results of these tests and discussed the performance characteristics of the system, verifying that the initial requirements set previously have been met.

Other than writing about numerical results of my system, like functional and non-functional requirements, I had to write my personal impression of the result.

I critically analyzed the whole project and its implementation, without any personal feelings towards it. I went through positive sides of my implementation of course, but I also honestly addressed weaknesses my implementation had and mistakes I made. In this part, I also proposed potential improvements and outlined future work that could extend the platform's capabilities, such as improved architecture and integration of 3rd party search engines.

Finally, at the end of the thesis I summarized the work that was done and highlighted the project’s contribution to the field of educational technology. In there I mentioned my personal learning outcomes achieved regarding backend architecture and software design patterns. Overall, the thesis aims to demonstrate not only the viability of the TutorSwap platform but also the engineering skill required to build the invisible but essential infrastructure for it.

# 2 LITERATURE REVIEW AND RELATED WORK

This chapter presents an analysis of existing solutions. It is important to analyze what already exists and what does not before rushing to code. To build something great, you must analyze first what was already done, why it did or did not work, and how I can use it for my project. In this chapter, I will go through peer-to-peer learning platforms, the business model of knowledge marketplaces, and technical options for the server.

## 2.1 Peer-to-Peer Learning Platforms

The peer-to-peer (P2P) model is not something new. P2P means students/ peers helping each other. It used to happen in university libraries, but now it is also possible in digital platforms.

Students perform better when taught by tutors (Bloom, 1984, p. 4). However, it is not possible to give a tutor to every student since it is too expensive for both students and the government. P2P platforms try to solve this by making the most effective way of learning free of charge (at least in a monetary sense).

There are several types of P2P platforms that I found: Stack Overflow, great for coding, but it has no real conversation between users rather than a single post and responses to it; Brainly, not a great solution because it is mostly used for copying answers and not learning the material; Tandem, which is closer to TutorSwap, but it introduces Double Coincidence of Wants, when it can be hard to find someone who knows your goal language and wants to learn language you know.

## 2.2 Knowledge Marketplaces

A knowledge marketplace is a more specific concept. It is like e-commerce like Amazon, where you use money to buy goods from others; in a knowledge marketplace, you use your knowledge to buy time.

Knowledge Marketplace uses the concept of “Time Banking.” Time banking is “a system of bartering various services for one another using labor-time as a unit of account which was developed by various socialist thinkers based on the labor theory of value” (Rohrs Schmitt, 2022). It means that I earn currency when I spend time with someone who requested it, and I spend currency when I request to spend time with someone. In the TutorSwap context, it means person A can teach mathematics for 1 hour to person B, earning 10 credits, and then use it for 1 hour of physics lesson from person C, spending the same 10 credits. By enforcing this system, I made it impossible for freeloaders to constantly take it but never give it. This concept ensures fairness on the platform: you must contribute to get help from the community.

Available “knowledge” marketplaces lack this system. Known examples are Preply, paid tutoring; Fiverr, a freelance marketplace, which is closer to ordering a finished product, rather than a lesson. TutorSwap sits in the middle; it enables learning from “tutors,” but makes it free, so it is more accessible to others.

## 2.3 Backend Architectures for Similar Platforms

Choosing the right architecture is important for the performance and maintainability of the software. There are three main patterns in modern web development that I could choose from.

### 2.3.1 Monolithic Architecture

In a monolith, all components, database connection, business logic, API layer, real-time chats feature, and background token exchange mechanism are packaged together.

There are upsides and downsides of this approach. Obviously, since it is the most straightforward approach, it is easy to develop, test and deploy, and all IDEs (Integrated Development Environments) support it. More than that, you, as

a developer, do not have to think about data transactions, since every module in your app uses the same memory space and can eventually reach needed data.

However, since it is the easiest approach, there are downsides to it as well. For example, big projects tend to get messy if they are written using monolithic architecture. Also, it is hard to scale a project that uses this architecture, because you must clone the whole app, not just the busy parts.

Having all that in mind, it is still the best choice for a startup MVP (Minimum Viable Product) (Fowler, 2014). And the reason for that is developers spending time on developing new features that users want, instead of focusing on complications, that other architectures present, which are not visible to customers.

### 2.3.2 Microservices Architecture

This architecture splits the application into small, independent services, for example, a User Service, Payment Service, or Notification Service.

There are upsides and downsides to this approach. On the positive side, it is highly scalable. If the “Search” feature is slow, you can add more servers just for Search instead of cloning the whole application, like in monolithic architecture.

However, there are significant downsides to it as well. It introduces massive complexity to the system. You need to handle network latency, distributed data consistency, and service discovery (Newman, 2015). Because of this added complexity and the need for manage many moving parts, it is usually inefficient for a single developer team to use this architecture.

### 2.3.3 Serverless Architecture

This architecture is also known as Function-as-a-Service (FaaS), and it heavily relies on cloud functions. In serverless architecture, "the cloud service provider

dynamically manages the allocation of compute resources," meaning you only write the function code while the provider handles the servers (Rajan, 2020).

Obviously, there are great advantages to this for an early-stage startup that tries to save some money. Research confirms that “the serverless functions are cheaper to operate than the microservices when the number of monthly requests is limited” (Allen et al., 2023). Additionally, “it provides the opportunity for faster time to market by dynamically and automatically allocating computers and memory based on user requests” (Arora et al., 2021), which means serverless functions automatically scale based on the usage.

However, just like the easiest approaches, there are downsides to this as well. As noted by Wen et al. (2023), “Scaling to zero makes incoming new requests face the cold start problem, which takes a long time to prepare required runtime environments from scratch.” This delay is a critical concern for real-time applications where user experience and satisfaction depend on sub-second response times. Furthermore, serverless models restrict you to specific vendors, “which limits flexibility and increases dependency risks” (Das et al., 2024). Relying on a single provider’s proprietary API means that migrating the architecture later could be too expensive for an early-stage startup.

In conclusion, I have selected Monolithic Architecture. Since I am the sole developer, maintaining a distributed system would take too much time. A modular monolith allows me to write clean code without the operational headache of microservices.

## 2.4 Technologies Overview

Based on the architectural decision, the following technology stack was selected. These tools are industry standards and are known for their reliability.

Backend Framework: Spring Boot, the most popular framework for Java backend development.

- It uses "Convention over Configuration," which saves time setting up the project (Walls, 2016).
- It has excellent security features (Spring Security), which are necessary for protecting user data.
- It is strictly typed (Java), which helps catch bugs before running the code.

Database: PostgreSQL, an open-source relational database.

- TutorSwap relies on complex relationships: Users have Skills, Skills have Connections (other Users), Users have Reservations.
- A relational database is better than NoSQL, like MongoDB, for this use case because I needed strict data integrity. I cannot allow a situation where a booking exists without a user. PostgreSQL enforces these constraints rigorously (PostgreSQL Global Development Group, 2025).

Containerization: Docker, the most popular, and the only one I know, containerization platform.

- Docker ensures consistency. The code runs the same on my laptop as it does on the university server (Merkel, 2014).
- It simplifies the database setup, for example I do not need to install PostgreSQL on my operating system; I simply run it as a container.

## 2.5 Summary of Related Work

To summarize, while there are many educational platforms available, they lack something that is important to me as a student. They are either high-quality but expensive, like Preply, or free but disorganized, like Discord. There is a clear lack of a platform that has the best of both worlds: structured and free knowledge exchange.

And for the technical side, there are many different architectures and technologies that could be used, for example many student projects use simple stacks like Node.js with MongoDB. However, to build a robust system that handles scheduling and transactions correctly, a more structured approach is needed. In addition. By utilizing a Monolithic architecture with Spring Boot and PostgreSQL, this thesis aims to create a backend that is both scalable and maintainable. This combination of a social need and a solid engineering approach forms the basis for the design and development of work described in the following chapters.

# 3 TECHNOLOGY STACK AND TOOLS

This chapter covers the technology stack and tools I chose for the project. Picking the right technology is the first major step before building an app. I wanted to learn something new, so I looked at what stack is popular for Backend engineers among recruiters.

One obvious choice was the “MERN” stack (MongoDB, Express, React, Node). It remains a popular choice due to massive popularity of React and Node.js (Stack Overflow, 2025), and a lot of companies are looking for MERN developers (SourceKode, 2026). However, I have already done multiple projects using this stack, and one of the criteria was something new. Another popular option was .NET and Spring Boot. They seemed similar to me, so the final choice was more based on my personal feelings towards the companies behind them (Microsoft and Oracle). And I had more negative experiences with Microsoft products, that is why I decided to learn Spring Boot, since Java is also more familiar to me than C#.

To accompany Spring Boot, I chose PostgreSQL as a database, since it is easier to structure complex data in it. Another tool I used was Docker for containerization, and this choice was made purely on the fact that I did not know any alternatives to it and, honestly, did not see any reason to look for it.

## 3.1 Spring Boot

### 3.1.1 Framework Overview

Spring Boot is an open-source Java-based framework. It is built on top of the “Spring Framework.” Spring Boot solves the issue of Java being too verbose. In the past, you had to write huge XML files just to start a server. Nowadays, Spring Boot uses “Autoconfiguration.” As described by Walls, this feature “leverages Spring 4 support for conditional configuration to make reasonable guesses about the beans your application needs and automatically configure them” (Walls, 2014, p. 541). This allows me to focus more on my specific business

logic, rather than writing a lot of boilerplate code and setting up all configurations.

### 3.1.2 Features Relevant to TutorSwap

There are several Spring Boot features that are relevant for TutorSwap and its backend.

In a complex app like TutorSwap, you have many “Services” (User logic, Reservation logic, Notification logic). Connecting them manually is messy and creates spaghetti code. Spring solves this issue by having a dependency injection or “wiring.” Spring manages this connection automatically. If ReservationService needs a database, Spring “injects” the necessary repository at runtime (Walls, 2014, Chapter 2). This makes the code modular and much easier to test.

Another relevant feature is Spring Security. Building a login system from scratch is a lot of work and, more than that, it is extremely dangerous. Using Spring Security to handle authentication, and especially JWT (JSON Web Tokens). This makes the server Stateless, which is a standard practice for modern scalable applications (Walls, 2014, Chapter 9).

Spring Data JPA seemed extremely useful during development. Writing raw SQL queries is risky and not a good practice because of SQL Injection attacks. Spring Data JPA acts as a translator; I can create a Repository class, call its functions, and the framework automatically generates sage SQL queries (Walls, 2014, Chapter 11). It saves a lot of time, makes code more readable, and protects the database from SQL injection.

## 3.2 PostgreSQL

### 3.2.1 Database Design Principles

For a database layer, I chose PostgreSQL, an open-source relational database. Another reliable option would be a NoSQL database like MongoDB. NoSQL databases are good for unstructured data; TutorSwap's data is rigid and heavily connected. For example, the Reservation table has a link to the User table. SQL databases enforce these links; I cannot create a booking for a nonexistent user. Another useful feature of SQL databases is atomicity. If User A sends credits to User B, atomicity ensures that either both happen (A loses, B gains) or neither does (PostgreSQL Global Development Group, 2025). A situation where credits vanish in the air is impossible with it.

Among all the SQL databases, I chose PostgreSQL because it is an industry standard for open-source relational databases and handles complex queries much better than MySQL (Obe & Hsu, 2017).

### 3.2.2 Use in TutorSwap

PostgreSQL is considered the most advanced open-source option. That is the reason I decided to have my database in it instead of MySQL, for example. Stricter SQL standards and better handling of complex queries played a huge role as well (Obe & Hsu, 2017).

One of the features TutorSwap needed is to ensure database integrity. PostgreSQL uses foreign key constraints heavily; I will not be able to delete a User if they have reservations, even if I wanted to. This prevents "orphan data."

PostgreSQL supports JSONB columns. It means that while the main data is relational, it is possible to store other data, like user profile preferences or flexible notification settings, as JSON documents. I did not use this feature in the current version of TutorSwap, but it is good to have it in mind for future releases of the app.

## 3.3 Docker

### 3.3.1 Containerization Concepts

One of the most frustrating problems in software development I faced was that code worked on my laptop but did not work on the production server. This happened mostly because of differences in operating systems, installed libraries, or versions of software used in the development.

Docker solved this issue for me. Docker is a containerization platform, and “containers are isolated processes for each of your app’s components” (n.d.). These processes made the application run quickly and reliably from one environment to another. Unlike a Virtual Machine (VM), which requires a full operating system, a Docker container shares the host machine's operating system’s kernel but keeps the application isolated. This makes containers extremely lightweight and easy to set up.

### 3.3.2 Deployment with Docker

For this project, I used Docker for both development and deployment. I created a Dockerfile for my Spring Boot Java application. Dockerfile is a script that tells Docker how to build the image of my app, for example it includes the version of Java used in the program and the actions needed to be executed.

Another Docker feature I used was Docker Compose. In my application, the backend required another service to be run before it was available. In my case, it was the PostgreSQL database. Docker Compose is a tool I used to define and run multiple containers to ensure all services are running together with my backend. For this to work I had to write docker-compose.yml file that mentioned 2 services: app, Spring Boot Java application, and database, PostgreSQL.

This setup made the development environment reproducible. A new developer, or me with a new laptop, could simply install Docker and run one command: docker-compose up. Docker automatically downloads the correct version of

PostgreSQL, sets up the network between the app and the database, and starts the system. This reduces configuration errors and ensures the environment on my current laptop, my future laptop, and the server that hosts my backend is the same.

## 3.4 Development Workflow

### 3.4.1 GitHub Flow

To manage the source code, I used Git, hosted on GitHub. Since I am the only developer of the backend system, I decided to use a simplified version of GitHub Flow to ship updates fast. It is built to be a lightweight, branch-based system (GitHub, n.d.), which was exactly what I needed for a startup.

I utilized feature branches instead of committing everything to the main branch. For every new requirement, for example, "implement login" or "create GET endpoint for users," I created a new branch, for example, feature/login. This allowed me to test my code on the server without changing the production code and breaking everything.

Once a feature was complete, I created a pull request. I did not have a teammate to review the code, so I had to do it myself. But it was a good practice, since I was reviewing my code before merging it into the main branch.

### 3.4.2 Testing Tools and Practices

It is not enough to produce the code; I had to make sure it actually works, so I used different testing strategies for that.

I created unit tests for the service layer using Junit and Mockito. For example, I tested my GET endpoints by ensuring they return consistent results, given a fixed fake database I used for tests.

Besides regular unit tests, I also wanted to interact with my API myself when I did not have time to write any tests. I used Postman, a popular API client. All I had to do was to put the address of my backend and endpoint I wanted to test, for example, GET /api/v1/users. This allowed me to manually verify that the JSON responses were correct and that the HTTP status codes were being returned properly. It also helped me verify that my backend handles wrong inputs properly by returning an error message.

In conclusion, I was happy about technologies I chose. Spring Boot, PostgreSQL and Docker provided a solid foundation for the first version of TutorSwap. After making a whole backend system with them I realised that these tools are industry standard for a reason, they reduced technical risks and allowed me to focus on shipping new features fast.

# 4 SYSTEM REQUIREMENTS AND ARCHITECTURE

This chapter covers the requirements that the TutorSwap backend must meet. It will cover what the backend must do (functional requirements), how it must do that (non-functional requirements), the architecture I chose for the project, and key components of the app. The goal for this project was to create a working MVP (Minimum Viable Product). However, I was still trying to make it scalable enough for a real university environment to handle.

## 4.1 Functional Requirements

Functional requirements define specific system behaviors or features. In the context of TutorSwap, these requirements come from two primary actors: the Student and the Tutor. However, I must note that one user acts as both actors.

### 4.1.1 Authentication

For the authentication part, there was not much creative freedom for me since it is the same everywhere. However, to have improved security, I decided to allow authentication only through Google accounts, since it reduces the load on me as a backend developer and outsources security measures to Google's side (Morkonda et al., 2021, p. 195-195). Basic functionality for authentication is registration, login, logout, and profile management. Registration, login, and logout were mostly handled on Google’s side; however, I still had to implement a system to save some app-related info for each user separately. Profile management was made completely on TutorSwap’s side. Users can edit their name, profile picture, bio, and most importantly, the list of skills they want to learn or can teach.

### 4.1.2 The Marketplace (Matching and Searching)

The main point of TutorSwap is connecting people together. For that reason, I had to give search capabilities backend. I decided to make a search based on the name and the skill the user teaches. Results are paginated to lessen the load on the frontend. For more passive users matching system built inside the GET all users endpoint. It works by pushing on top of people who match the user the most, based on the skill they teach and the last time (to maintain a connection between active users and not “dead” ones). In later versions, it was decided to add a confidence level to each skill user can teach. This way, people who already have a decent understanding of a certain skill or knowledge do not match with people on the same skill level who want to teach absolute newbies.

### 4.1.3 Booking and Session Management

The system gives a user the ability to add their available time into the app and then share it with “students,” so that they can select what time suits them the most. Of course, if one student already booked a lesson, then this time slot is not vacant anymore. Reservations go through several stages in their lifecycle: first, students select a suitable time and request a lesson (pending status), then the teacher can reject (canceled status) or accept (accepted status). Users can also cancel a session. If it is canceled too close to the lesson, the system punishes that user.

### 4.1.4 Credit System (Time Banking)

All users have a certain number of tokens from the start; one token is equal to one minute of a lesson. Teachers can specify their preferred time frames (15, 30, 60 minutes) so that students cannot book too little or too much time. The system always checks the user’s balance before confirming any lessons to ensure that all tokens are transferred properly. After the teacher confirms the lesson, students’ tokens move to reserved tokens, so that they cannot use them again in a different lesson, but also to not transfer them to the teacher too early.

If the student cancels the lesson too close to its start, reserved tokens would be transferred, nevertheless. Also, to protect students, reserved tokens return to the main user's tokens if the teacher does not show up for the lesson.

## 4.2 Non-Functional Requirements

While functional requirements describe what the system does, NFRs (non-functional requirements) describe how well the system does it.

### 4.2.1 Scalability

The main NFR for this project was scalability. The backend must be able to handle a load of up to 1000 concurrent students during exam weeks. The architecture should be stateless, by using JWT (JSON Web Tokens) to allow horizontal scaling (running multiple instances of the backend server behind a load balance).

### 4.2.2 Performance and Latency

The API should respond to standard requests quickly, for example, loading the user list should be done within 400 milliseconds. Slow apps frustrate users and lead to the loss of customers.

### 4.2.3 Security

All data in transit must be encrypted using the HTTPS protocol. Another important security NFR is authorization: one user should not be able to access another's data, like chats and messages.

### 4.2.4 Availability and Reliability.

The system should aim for 99.9% up time during the semester. If the backend crashes, it should be able to restart automatically without data corruption. This should be handled by Docker.

### 4.2.5 Maintainability.

Since the project's scale is big and was meant to be developed over a long time, it should be written following "Clean code" principles. It ensures that the code is understandable and easy to debug or edit later, as code is read far more often that it is written (Martin, 2008, ch. 1). Other than that, the code must be modular, meaning that functions and classes should have a single responsibility so that errors in one logic do not break another (Martin, 2008, ch. 3). This modularity ensures that components are decoupled unless they are working one after another, where one's output goes into another's input.

## 4.3 System Architecture Overview

The architecture of TutorSwap follows the standard Client-Server model. The Client is the mobile application, in my case React Native, but it is out of scope for implementation details, and the Server is the Spring Boot application connecting to a PostgreSQL database.

Internally, the backend utilizes a Layered Architecture. This is a standard pattern in software development (Fowler, 2012, p. 17). Layered Architecture enforces a strict Separation of Concerns. The application is divided into three horizontal layers. The first one is a presentation layer or controllers. This is the entry point of the system; it handles HTTP requests, parses JSON input, and validates parameters. It does not contain business logic; it simply delegates tasks to the service layer, also called the business logic layer, which is the "brain" of the application. It contains the rules of the system, for example, checking if the user has enough credits before booking a lesson. Service layer orchestrates the

flow of data. The final layer is a data access layer or repositories. This layer communicates directly with the database and performs CRUD (Create, Read, Update, Delete) operations. It abstracts the SQL details away from the rest of the app.

This separation makes the system robust. If I decide to change the database from PostgreSQL to MySQL in the future, I only need to modify the Repository layer; the Controllers and Services remain untouched.

## 4.4 Key Components

### 4.4.1 User Management

The user management module is the foundation of the system. It handles the lifecycle of a user account.

To store users' information, I had to use database and since the whole app is built around users' interactions, the User entity is the central node in the database graph. It holds personal details and is linked to Role entities (ADMIN, USER).

Just having a table in database for users is not enough, so I started configuring Google authentication in the system. When a user logs in, the AuthController receives the credentials. The AuthService verifies them using Google Authentication methods and links returned from Google ID to ID stored in the database. If the process succeeds, a JSON Web Token (JWT) is generated and returned to the client. This token acts as a digital passport for all subsequent requests.

Before returning response from most of the system endpoints, it checks if the user can perform certain actions by checking if his user ID is linked to the entity he is trying to change/delete/create.

### 4.4.2 Matching Engine

The matching engine is the unique value proposition of TutorSwap. Unlike a simple search by name, this module handles the logic of double coincidence of wants. The engine performs a database query filtering by exact skill match and last online time. For this thesis, a greedy matching algorithm is implemented that prioritizes the most recent online status among users with exact skill matches. The query joins the User, UserSkill, and UserLastSeen tables, then filters for users whose teaching skills exactly match the logged-in user's learning interests by comparing skill details where the potential match has skillType='TO_TEACH' and the current user has skillType='TO_LEARN'. The results are ordered by lastSeenAt in descending order to prioritize recently active users. This approach ensures users receive highly relevant suggestions rather than browsing through lists, streamlining the matching process for skill exchange.

### 4.4.3 Real-time Communication

For a marketplace to function, buyers and sellers, or students and tutors, must communicate. The simplest solution for in-app communication is chat, where users can discuss lesson details before booking. I had a choice of making communication near real-time or true real-time, but the last one sounded like a better approach in general and a good opportunity for me to learn more, so I ended up implementing true real-time communication using WebSockets, even though they introduce significant complexity, for example, managing open connections or statefulness (Fette & Melnikov, 2011, p. 5).

### 4.4.4 Data Flow and Integration

Data flows through the system in a predictable, unidirectional manner for most operations. Mobile app sends a JSON payload to the backend’s endpoint; controller layer validates the input based on defined rules. Then, the service layer checks business rules, for example, if the user has enough credits or not, and

then the repository layer performs actions on database to save/edit/delete some data. After it went down to the repository layer, server returns HTTP response with additional details.

This standardized flow ensures that errors are caught early, and that data is consistent before it ever reaches the database.

## 4.5 Security Considerations

One of the main concerns of backend developers is security. The first security decision I made was avoiding server-side sessions. This eliminates session fixation attacks, that rely on the server maintaining a “state” (a record in a database or memory) that links a specific ID to a user (Rahmatullo et al., 2019, p. 37). The JWT is signed with a secret key, so users cannot tamper their identity.

To prevent SQL Injection, I used the Hibernate ORM, which automatically escapes SQL parameters. To prevent cross-site scripting (XSS), the API verifies that user input does not contain malicious scripts.

By adhering to these rigorous requirements and architectural standards, the TutorSwap backend is designed to be a secure, reliable, and efficient platform for peer-to-peer learning.

# 5 BACKEND DESIGN AND IMPLEMENTATION

## 5.1 Database Design

Good database design is the foundation of a proper backend system. It ensures data consistency and efficiency, while poor design leads to slow performance and later development complications.

### 5.1.1 Schema Overview

The database schema in TutorSwap was designed according to 3NF (Third Normal Form) to minimize redundancy. It requires all non-key attributes to depend directly on the primary key (Kent, 1983, p. 120). This constraint makes it easier to add new tables to the database later.

One of the core decisions I made was the use of UUIDs (Universally Unique Identifiers) for all primary keys instead of plain integers. This choice provides two benefits: malicious users cannot guess the ID of anything in the database by simply incrementing a known ID; UUIDs can be generated by the server without checking the database beforehand.

The schema also implements a polymorphic design for the messaging system. Rather than stuffing all message types (text, video, lesson requests) into a single table, the system uses a base messages table containing common metadata, which is then extended by specific tables like textmessages and imagemessages. It helps to keep the database structure clean and easy to understand.

Naming conventions for the database are a little bit trickier since Java and PostgreSQL have different rules. Java uses “camelCase,” and PostgreSQL uses snake_case. However, it was easily solved by Hibernate, which automatically maps Java names to SQL using naming strategies. Unfortunately, nothing is

perfect, so sometimes it did not work, and I had to use JPA as well to manually specify the table name in SQL.

JSONB columns were not used in this version of the software, even though they were one of the reasons to choose PostgreSQL.

### 5.1.2 Entity-Relationship Diagrams

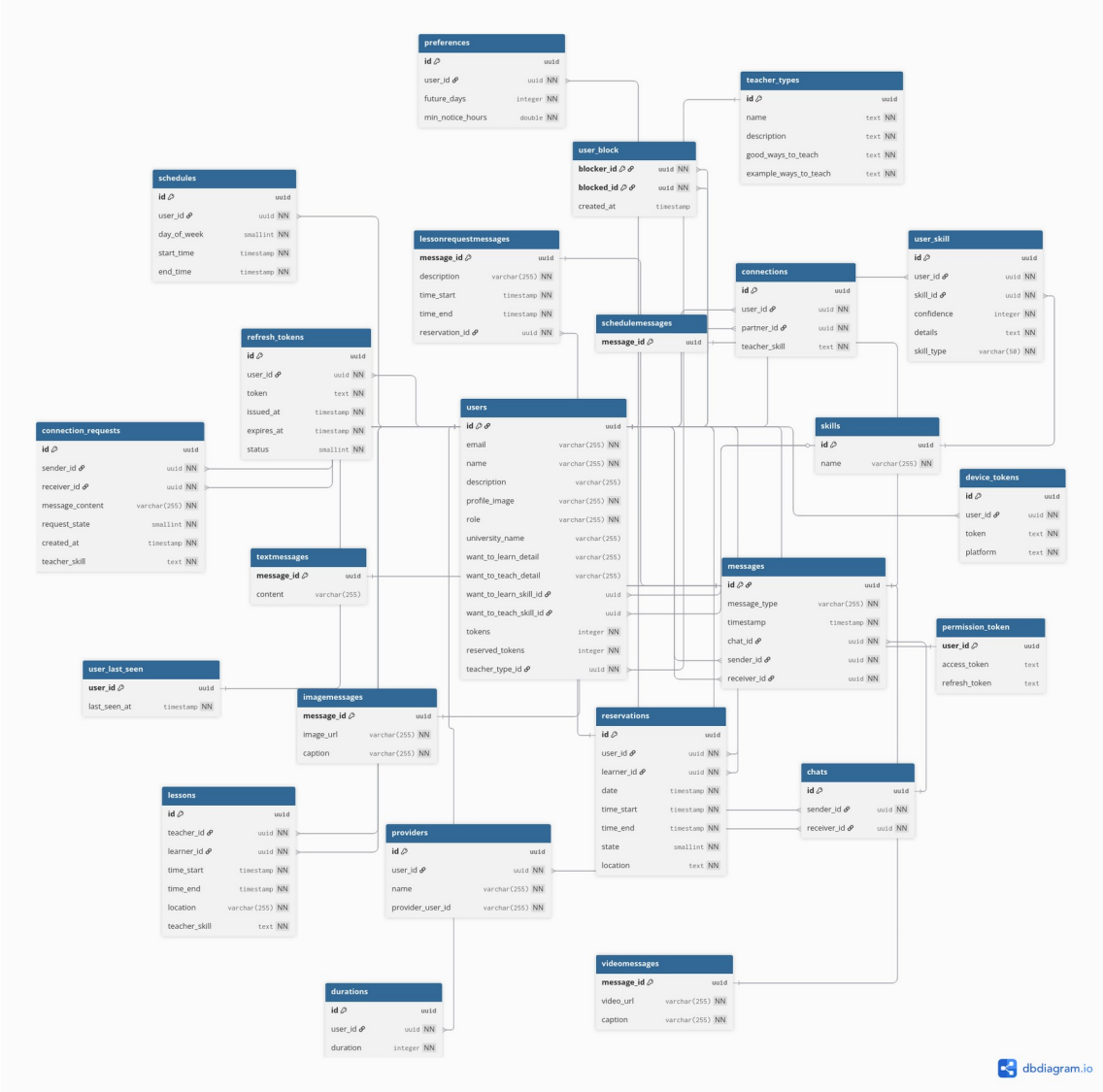


Figure 1. Entity-Relationship Diagram of the TutorSwap Database (Figure: Roman Zinkevich, 2026).

The Entity-Relationship Diagram (ERD) is presented in Figure 1. There are many tables in the diagram, and all are important, so I must cover them.

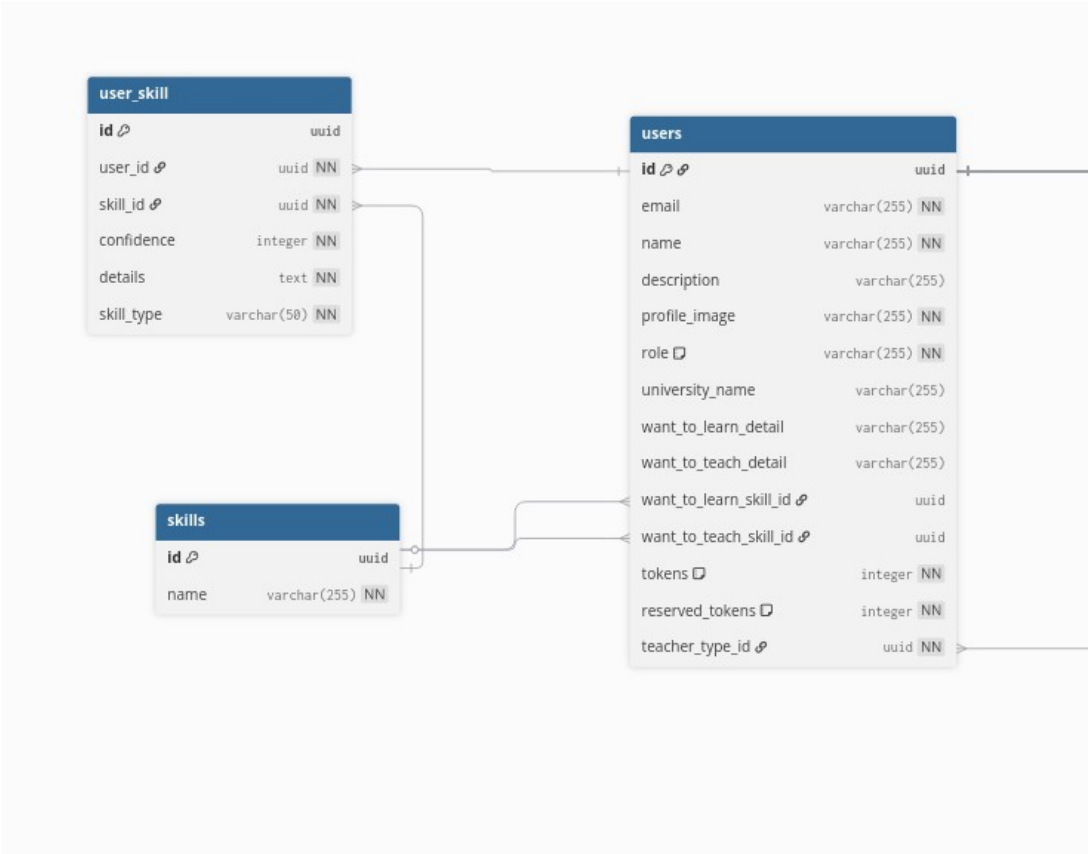


Figure 2. Entity-Relationship Diagram of the TutorSwap Database, highlighting user and skill relationship (Figure: Roman Zinkevich, 2026).

The central part of the system is the Users table. Since TutorSwap relies on a "prosumer" model where everyone can be both a student and a teacher, I did not split them into separate tables. Instead, users are distinguished by their skills. The Skills table holds all subjects. To link them, I used an intermediate table called User_Skill. It is not just a simple link; it also stores the confidence level details about why the user is good at that skill and if they want to teach or learn this skill. This allows the search algorithm to be much more precise. I used a separate table to link users and skills to make it possible for users to have several skills he wants to learn or teach. It is also visible in Figure 2 above, that there is direct connection between users and skills through user’s want_to_learn_skill_id or want_to_teach_skill_id. This was made during the first version of TutorSwap when I thought having one skill to learn/teach is good enough. It is still present in the database to serve users who signed up during that time.

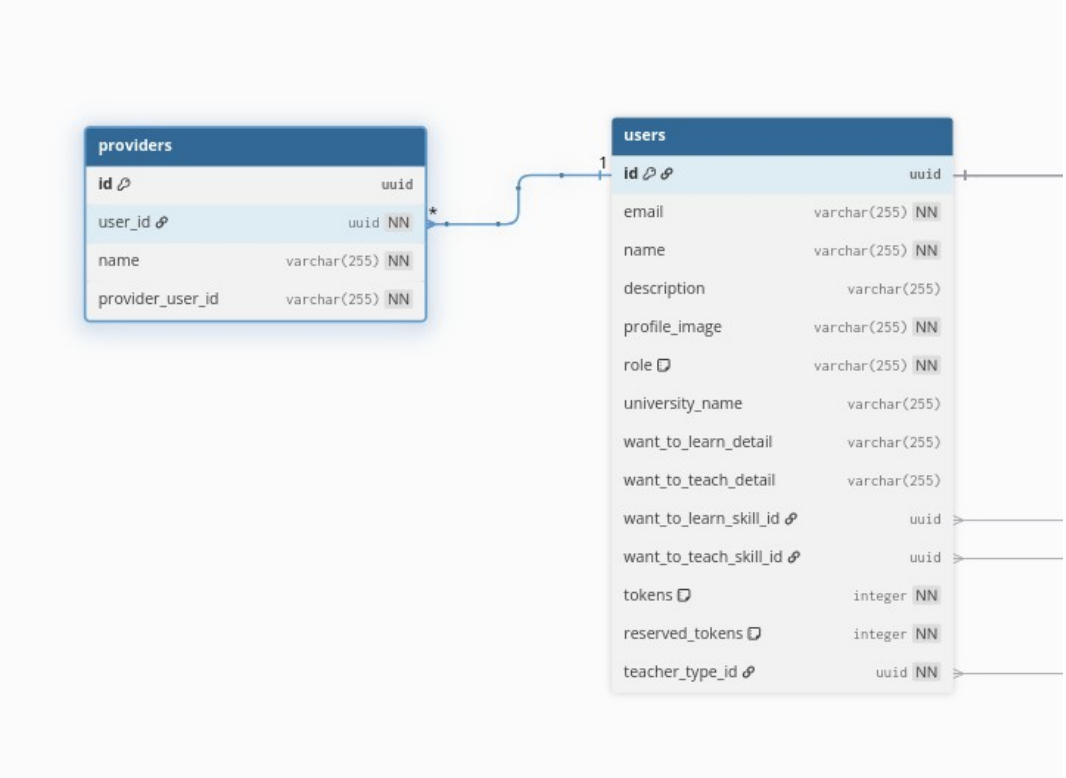


Figure 3. Entity-Relationship Diagram of the TutorSwap Database, highlighting users and providers structure (Figure: Roman Zinkevich, 2026).

Closely linked to the user profile is the Providers table; connection can be seen above under Figure 3. Since the application supports OAuth authentication (Google Sign-In), the system does not store passwords for these users. Instead, the Providers table links the local user's ID to the external provider's unique identifier (e.g., the Google Subject ID). This decouples the authentication method from the user identity, making it possible to add other login providers (like Facebook or GitHub) in the future without altering the core user table.

Connected to the user profile, several configuration tables define how a tutor operates, visible below under Figure 4. Teacher_Types defines the teaching style (e.g., "The Yapper" or "The Doodler", one is good by explaining in words, another one is good by “doodling” schema and pictures for better understanding), while Preferences and Durations store settings like how much notice a tutor needs before a lesson or how long their sessions usually last. To manage availability, the Schedules table tracks the specific days and times in a week a user is willing to teach.

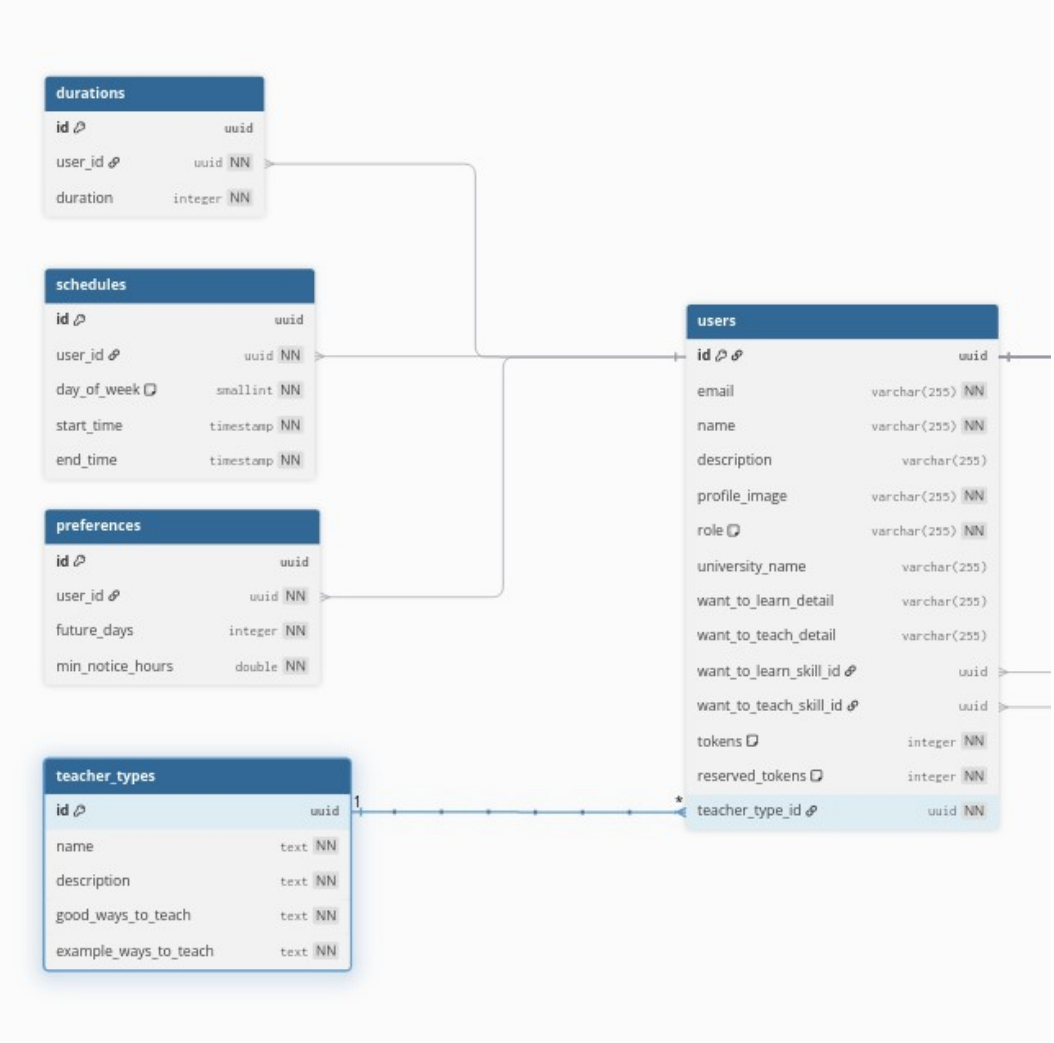


Figure 4. Entity-Relationship Diagram of the TutorSwap Database, highlighting users and users' details and preferences relationship (Figure: Roman Zinkevich 2026).

The social aspect of the app is handled by Connections and Connection_Requests, visible under Figure 5 below. Before users can chat freely, they must establish a connection. The requests table tracks the state of this "friend request" (pending, accepted, or rejected), and once accepted, a record is created in the connections table.

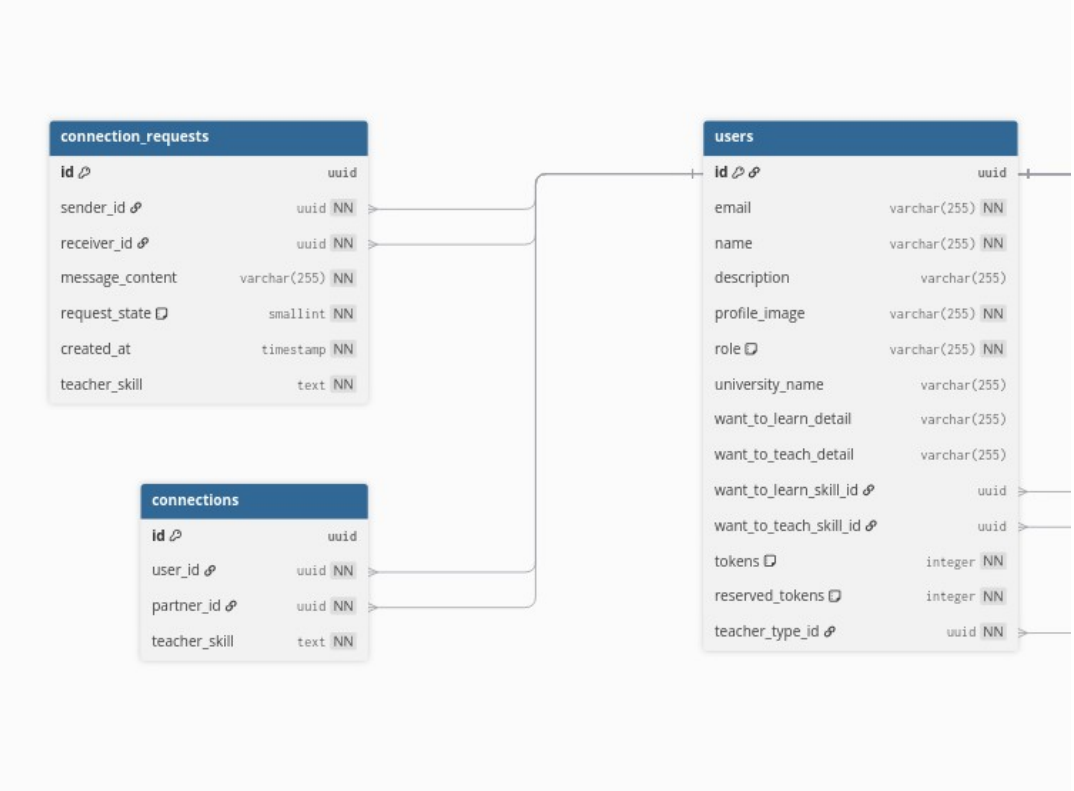


Figure 5. Entity-Relationship Diagram of the TutorSwap Database, highlighting connection structure (Figure: Roman Zinkevich, 2026).

Communication is built around the Chats table, which simply links two users. Figure 6 with related tables can be seen below. The actual content is stored in the Messages table. As mentioned in the schema overview, this uses a poly-morphic pattern. The main Messages table stores the sender, receiver, and timestamp, while the specific content goes into separate tables: TextMessages, ImageMessages, VideoMessages, and ScheduleMessages. A special case is the LessonRequestMessages table, which links a chat message directly to a Reservation. This allows users to discuss and accept bookings directly inside the chat interface.

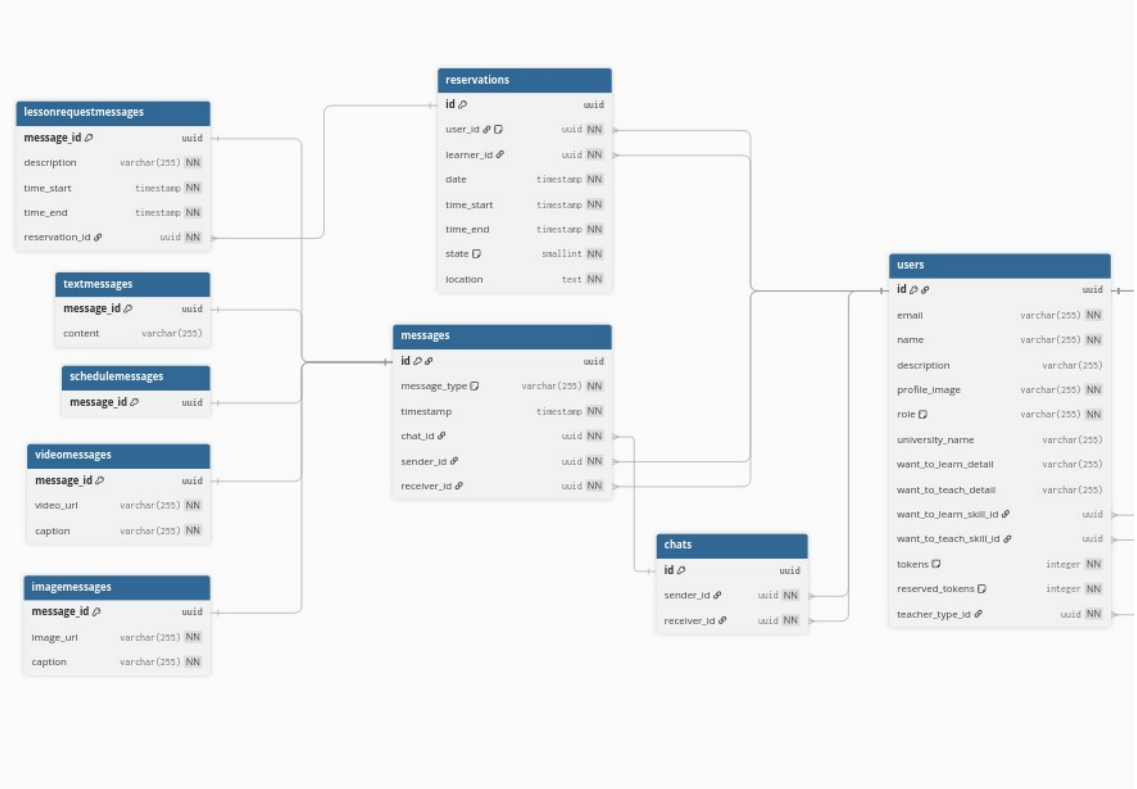


Figure 6. Entity-Relationship Diagram of the TutorSwap Database, highlighting tables related to chat logic (Figure: Roman Zinkevich, 2026).

The booking process itself is split between two tables, visible below under Figure 7. Reservations represent the negotiation phase; it holds the proposed time and the status (pending/accepted). Once a reservation is confirmed and the lesson happens, the data is finalized in the Lessons table, which serves as the permanent record for history and transaction logs.

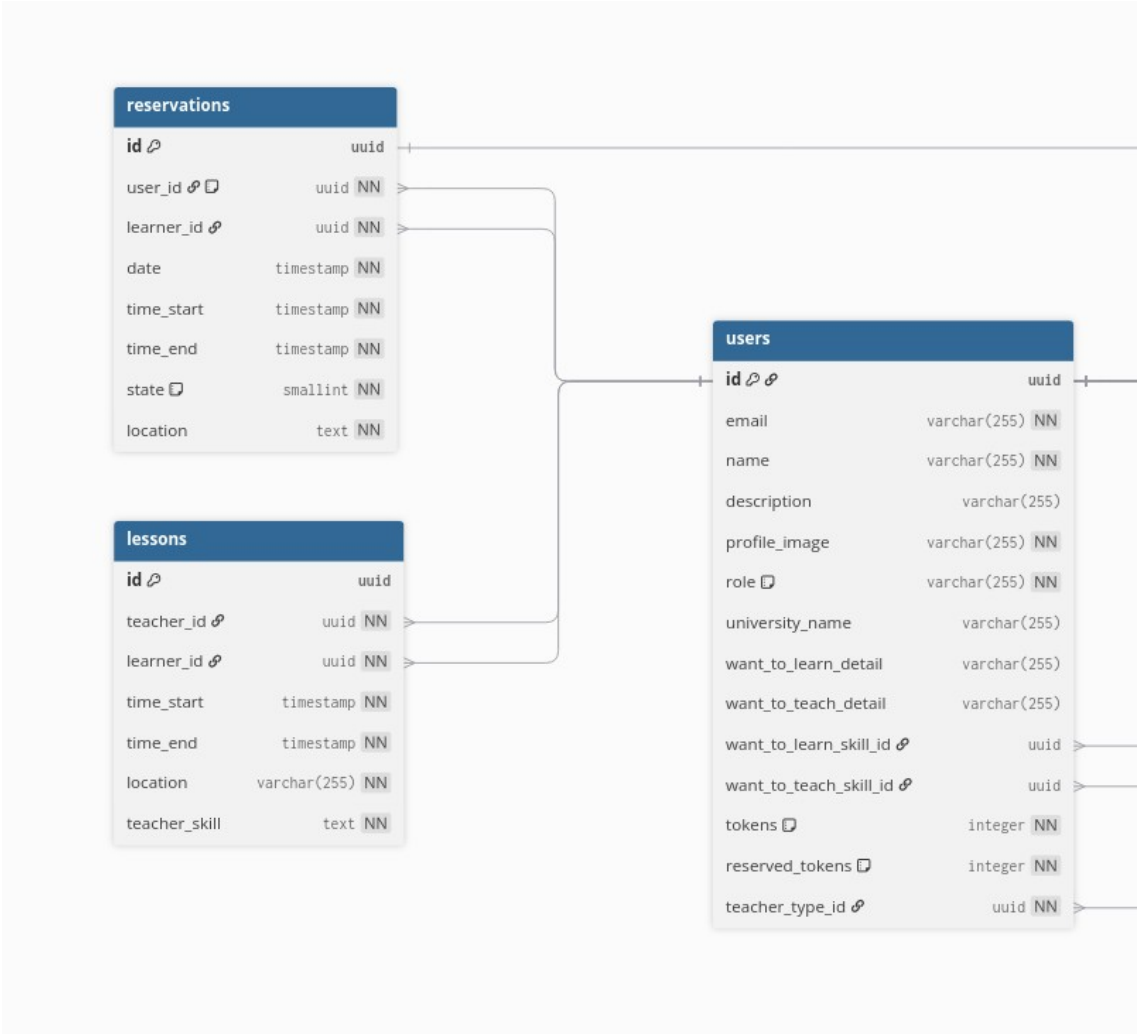


Figure 7. Entity-Relationship Diagram of the TutorSwap Database, highlighting tables related to reservation system (Figure: Roman Zinkevich, 2026).

Finally, under Figure 8 below, there are several utility tables used for security and background features. Refresh_Tokens and Permission_Token are used to manage user sessions and JWT authentication securely. Device_Tokens stores identifiers for sending push notifications to mobile devices. To ensure user safety, the User_Block table allows users to prevent others from contacting them, and User_Last_Seen tracks activity status to show if a user is online.

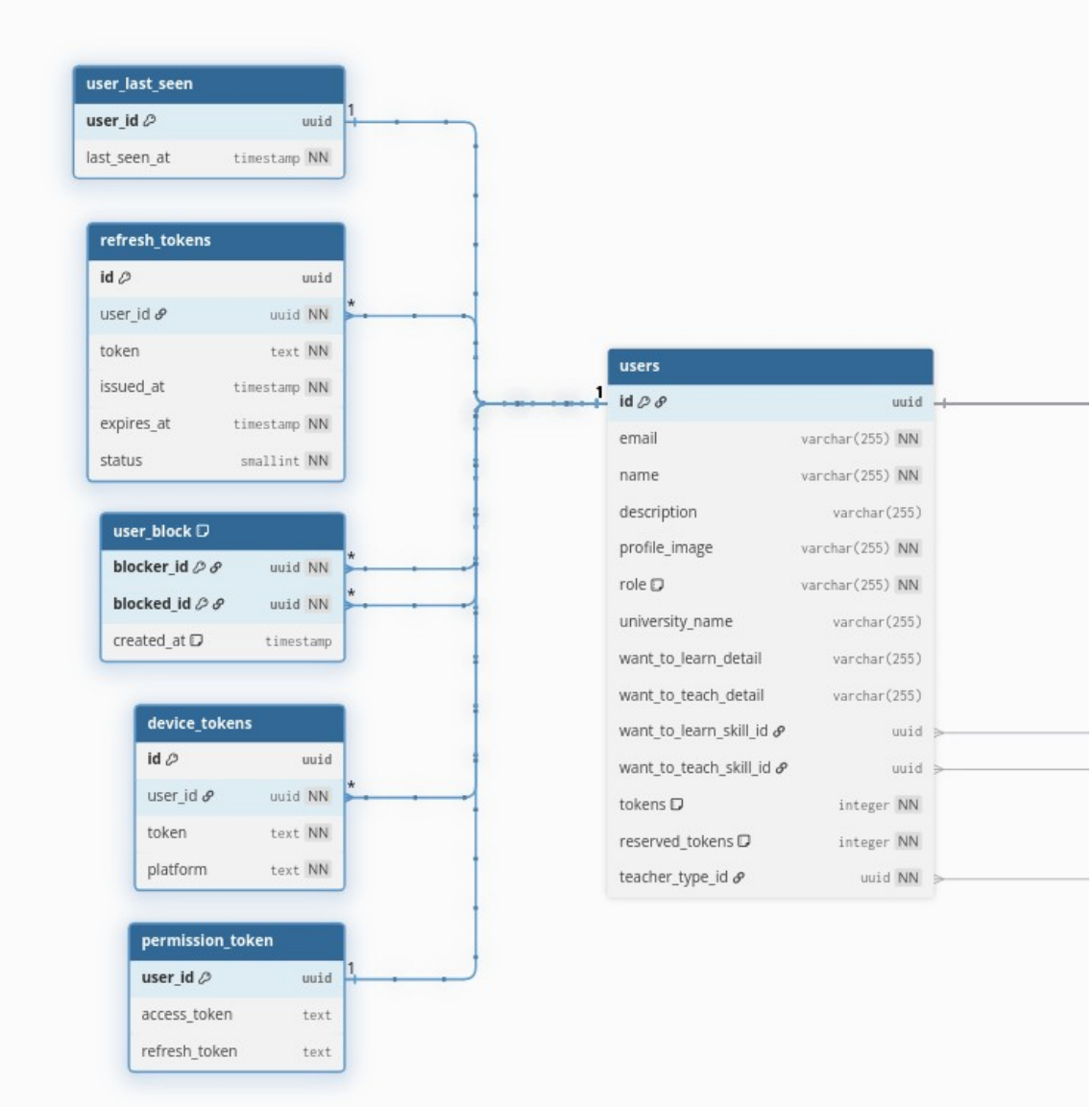


Figure 8. Entity-Relationship Diagram of the TutorSwap Database, highlighting users' extra information (Figure: Roman Zinkevich, 2026).

## 5.2 API Design

### 5.2.1 REST Endpoints

The backend exposes a RESTful API to communicate with the frontend (mobile app). I organized the controllers by "Resource" rather than by function, meaning all operations related to bookings are handled by the ReservationController, while connection logic resides in the ConnectionController.

A key architectural decision was to implement API Versioning from the start. The URL structure follows the pattern /api/v1/{resource}. For critical components like Authentication and User Management, I introduced a V2 iteration (AuthControllerV2, UserControllerV2). This strategy allows me to deploy breaking changes or improvements to the login logic without disrupting older versions of the mobile app that might still be running on users' devices.

While most of the applications use standard HTTP REST endpoints, the Chat feature required a hybrid approach. I implemented a ChatRestController to handle standard history fetching (loading old messages), but I also created a dedicated ChatController to handle real-time communication via WebSockets. This separation ensures that the overhead of maintaining a persistent WebSocket connection is only incurred when necessary.

To ensure the API is maintainable and easy to test, I integrated Swagger (SpringDoc OpenAPI). This tool automatically generates interactive documentation for all endpoints, allowing me to test requests directly in the browser without writing external scripts.

### 5.2.2 Request and Response Models

One of the most important rules I followed was to never return raw database entities (the classes linked to the DB tables) directly to the user. Doing that causes problems, one of which is security. Returning a User entity would accidentally expose sensitive fields like the password hash or internal database metadata. Another issue is the possibility of infinite recursion. Since the database has bidirectional relationships, for example, a user had reservations, and a reservation has a user, converting this directly to JSON would cause the server to crash with an error.

To solve this, I used DTOs (Data Transfer Objects). These are simple Java classes that define exactly what data enters and leaves the API. For example, the UserDto contains profile information but strictly excludes the password field.

To bridge the gap between the Database Entities and these DTOs, I utilized the MapStruct library (Morling et al., 2014). Instead of writing hundreds of lines of manual conversion code (e.g., dto.setName(entity.getName())), MapStruct automatically generates the mapper implementations at compile time. This kept my codebase clean and reduced the risk of human error when mapping complex objects like Lesson or ConnectionRequest.

## 5.3 Core Backend Modules

The backend business logic is divided into modular services, each responsible for a specific domain of the application.

### 5.3.1 User Module

The User Module is responsible for managing the lifecycle of user profiles. Since authentication is handled via OAuth, the primary responsibility of this module is Profile Enrichment. When a user first logs in via Google, the system creates a basic account. The User Module then handles the onboarding wizard, where users add their "Skills to Learn" and "Skills to Teach."

A critical component here is the Credit Management logic. The user entity holds the tokens balance. The User Module exposes methods to atomically update this balance, ensuring that credits are locked (reserved tokens) when a booking is requested and only transferred when the lesson is completed.

### 5.3.2 Matching Module

The Matching Module serves as the search engine for TutorSwap. I did not want to rely on a simple text match, so I implemented a sophisticated SQL-based matching algorithm directly in the UserRepository.

The search logic is not just a single query; it combines multiple criteria to find the best fit. First, it looks for users who list the specific skill in their TO_TEACH list. Then it checks if the user's name or bio details match the query strings. After that, the module applies exclusions. The system automatically filters out users who are already connected to the requester, users who have been blocked, and obviously, the requester themselves. Finally, the results are ordered by recency to ensure active users appear first.

By handling this complex filtering at the database level rather than in Java memory, the system remains fast even with many users.

### 5.3.3 Communication Module

For the chat functionality, I needed a true real-time performance. I implemented the STOMP protocol (Simple Text Oriented Messaging Protocol) over WebSockets. I chose STOMP because it provides a standardized format for defining message destinations, which is much cleaner than handling raw WebSocket frames (Stumpf et al., 2012).

The configuration supports two messaging patterns. Topics pattern is used to broadcast messages to all participants in a specific chat room. Queues pattern is used for private, user-specific updates.

To ensure reliability, I enabled SockJS fallback. If a user's network blocks standard WebSocket connections (common in some university firewalls), the client automatically downgrades to HTTP polling without breaking the chat experience.

### 5.3.4 Notifications Module

To keep users engaged, the system sends push notifications for events like "New Message" or "Lesson Request." Instead of integrating Firebase Cloud Messaging (FCM) and Apple APNs separately, which is a configuration night-

mare, I utilized the Expo Push Notification Service since frontend is developed using React Native and Expo.

Using the expo-server-sdk-java, the backend sends a unified notification payload to Expo's servers, which then handles the delivery to both iOS and Android devices. The module categorizes notifications by type (e.g., LESSON_REQUEST, CHAT_MESSAGE) and includes relevant metadata, allowing the mobile app to navigate the user directly to the correct screen when they tap the notification.

## 5.4 Authentication and Authorization

### 5.4.1 JWT Implementation

Since the backend is stateless, I implemented JSON Web Tokens (JWT) for session management. When a user successfully logs in via Google, the server issues two tokens: an access token and a refresh token. Access token is short-lived, only 15 minutes, and used to authorize API requests. A refresh token, on the other hand, is long-lived and is stored securely in the database.

I created a custom JwtFilter that intercepts every HTTP request. It extracts the token, validates the digital signature, and automatically sets the user's identity in the Spring Security context. This ensures that controllers do not need to manually check who is logged in.

### 5.4.2 Role-Based Access Control

Although most users are students/tutors, the system includes an ADMIN role. I used Spring Security’s annotations (like @PreAuthorize) to protect sensitive endpoints. For example, only an Admin can view the global transaction logs or ban a user. This logic is decoupled from the business code, making it easy to change permissions later without rewriting the service logic.

## 5.5 Containerization and Deployment

### 5.5.1 Docker Setup

To solve the "it works on my machine" problem, I containerized the entire application. I wrote a Dockerfile for the Spring Boot backend and a docker-compose.yml file that orchestrates the entire stack. This includes not just the Java app and the PostgreSQL database, but also a Redis container (used for caching) and an OpenTelemetry collector for monitoring performance.

### 5.5.2 Deployment Pipeline

The application is hosted on a private VPS (Virtual Private Server). To automate the deployment, I built a CI/CD pipeline using GitHub Actions.

The workflow is triggered automatically whenever the code is pushed to the repository. Workflow consists of three steps: build, registry, and deploy. On build, as the name suggests, it builds a java JAR file and creates a Docker image. On registry, the image is pushed to Docker Hub, so it can be retrieved later. On the final step, it connects to the remote server via SSH, pulls the new image from Docker Hub, and executes a rolling update using Docker Compose.

I also set up two distinct environments: Production (tutorswap.190304.xyz) and Test (test.tutorswap.190304.xyz). Pushes to the dev branch automatically deploy to the Test environment, allowing me to verify new features safely before they go live for real users.

# 6 EVALUATION AND TESTING

Since this project is an MVP (Minimum Viable Product) developed by a single developer, I did not aim for 100% test coverage, instead, my strategy was to develop new features fast but still ensure the quality of produced code. I was testing my application manually by calling API through Swagger or through existing mobile application. However, I still wrote few tests in the project.

## 6.1 Testing Approach

For automatic testing I used default Spring Boot tools, like Junit 5, Mockito and AssertJ. This part of the testing was divided into three different categories.

### 6.1.1 Unit Testing

Unit testing is used to check how individual components work in isolation. Since the testing was done for educational purposes rather than anything professional, I did not implement unit tests for every single service class.

I wrote tests for the skills repository to ensure that my custom queries were correct and to see how unit testing is done in Java.

Writing unit tests is an easy job, so I never focused on that, having in mind that if I really want to, I can do that at any moment.

### 6.1.2 Integration Testing

Integration testing, unlike unit testing, checks how different parts of the system work together. So, in unit testing I would test separately how my API endpoint works, how service layer processes data and how repository layer saves data in the database, but in integration testing all of this would be one test.

I did not implement any of the integration tests in my application, except for one simple test. That is TutorapiApplicationTests, which tries to start the application context. If any Spring Boot bean is missing of if the database configuration is wrong, this test fails. It may be simple, but it is also very important, because this test runs automatically in GitHub Actions, if I make a mistake and deploy code that does not even start, this test will fail and the broken code would never reach production.

### 6.1.3 Performance Testing

I did not perform any proper load testing with tools like Jmeter because it was not part of my “ship new features fast” ideology. Of course, I kept performance in mind since it is a crucial part of backend software.

I monitored time it takes to return data from my endpoints and optimized my code if needed. I used caching for GET endpoints to return data back faster, and I also utilized database indexing to improve speed of my queries. After everything I have done, I manually tested endpoints via Swagger and noticed that standard API requests, like searching for a tutor, consistently returned results in under 400ms, which is acceptable for a university project.

## 6.2 Usability Metrics

The main tool for testing usability was the Swagger UI I mentioned before. It provides a self-documenting interface for the API. Swagger allowed me to test endpoints without looking at the code, by simply calling APIs from the website. For the usability I focused on consistent error handling. I made sure that endpoints return structured error messages with specific codes, so that both users and frontend developer can understand what went wrong and what they can do about this. Alternative to this would be returning generic “Internal Server Error”, and very first versions of the app did that, but it made debugging so much harder for both me and frontend developer, so I had to change that.

## 6.3 Results and Analysis

In the end, the system meets the functional and non-functional requirements mentioned in Chapter 4.

The CI/CD pipeline has successfully deployed the application to the production server multiple times without issues, which proves that the Docker setup works. Through Swagger, I tested that every endpoint of my application works and that the backend server handles errors without crashing. My manual checks also confirmed that the database constraints correctly prevent invalid data from existing in the system.

Although the automated test coverage is low, the architecture and its stability form a solid foundation for a beta release.

# 7 DISCUSSION AND REFLECTION

This chapter reflects on TutorSwap backend. Now that the system is built, deployed, and tested, I can look back at the decisions I made during development. While the current version works well as an MVP, there are clearly sides to it that were made when time constraint was critical.

## 7.1 Strengths of the Backend Architecture

The strongest aspect of the system is its data integrity. By choosing PostgreSQL, and enforcing strict foreign key constraints and UUIDs, I made sure that “orphan data” is impossible. If I delete chat, all messages in it are deleted as well and nothing is left in the database. This decision to prioritize a strict schema over flexibility and faster development of NoSQL paid off, as it prevented many logical bugs during development and left me with properly structured database, which is satisfying to work with.

Another key strength is DevOps Automation. Setting up the CI/CD pipeline with GitHub Actions was difficult and it took some time before I started writing Java code, but in the end it paid off, since it saved me literal hours of manually deploying new code to the server, so that frontend can work with it. Later I added deployment of test version to the server, so that frontend can test new features before applying them to production.

Finally, the stateless architecture using JWT was the right choice. Of course, it is normal practice in every company nowadays, but I still must mention it. Since the server does not hold session data in memory, the API is fast and easily scalable. If I needed to handle 5000 users any time soon, I could simply use second Docker container behind a load balancer without rewriting any server code.

## 7.2 Identified Weaknesses and Limitations

The most significant weakness noticed so far is the test coverage. As discussed above in the previous chapter, 5% coverage is risky. While the system works now, making major changes in the future (like refactoring any logic) will be dangerous because there are no tests to confirm that new updates are safe and everything works. In fact, when I did any major update during development, I spent a lot of time fixing bugs that would be easy to catch if I had any tests. Instead, I had to manually test my endpoints after updates. If I am up to develop another backend system for any application, tests are what I would spend more time from the beginning since it pays off later.

Another weakness is searching for implementation. Currently, I use SQL LIKE queries to find tutors. While this works in current version of the app with <200 users, it is not efficient for 100,000 users and if I ever hit that number, I would have to rewrite the entire logic. If I were to change it right now, I would use a dedicated search engine.

## 7.3 Recommendations for Improvement

For the next version of TutorSwap, I would focus on two major improvements that would accelerate the quality of the product exponentially.

First of all, I would integrate ElasticSearch. Instead of querying PostgreSQL directly for user searches, the system should synchronize user profiles to an ElasticSearch cluster. This would allow the system to handle typos and improve query speed (Elastic, n.d.).

Another, more important improvement, is a shift to microservices. Right now, the monolithic architecture is working fine. However, there are modules that generate more traffic compared to the rest of the app. And this “popular” modules would change depending on the stage of the app. To solve this issue, it would be smart to split server into microservices so they can be scaled independently based on the needs of specific module.

### 7.4 Lessons Learned

This project taught me the difference between personal “pet project” development and engineering a backend for a mobile application with real users. If I were building personal pet project, I would simply change my authentication logic and forget about it, but since I have users, and backend is designed for mobile app, I have to think about people who are not going to update the app and will use old backend routes. Because of that, I must keep the old routes and add new ones. This realization helped me to understand what backward compatibility is and why it is needed.

Implementing the Chat module taught me that Stateful (WebSockets) and Stateless (REST) paradigms require completely different thinking. After practicing in REST for 3 years at university, when I first started working with WebSockets in TutorSwap I was applying the same logic to build it, and it was fundamentally wrong. Eventually I learned that these are different paradigms and I must use them differently, however, I believe I still must practice WebSockets to fully internalize this understanding inside me.

# 8 CONCLUSION

## 8.1 Summary of Work

The goal of this thesis was to build a solid backend for TutorSwap, a platform designed to let students exchange knowledge without the barrier of high costs. I started this project because I wanted to learn something that university could not offer and realized that people around me, fellow students, could teach me if only I had a way to provide value for them back. From technical perspective, I was trying to create a platform, where a user can be both a student and a tutor, and where interaction between users is managed by a time-banking credit system.

For the implementation, I used a monolithic layered architecture, to develop which I used Spring Boot, PostgreSQL, and Docker. In the end, I had a working MVP with matching engine, real-time chat, and fully automated deployment pipeline. On top of having a finished project, what I gained during its development, was experience and knowledge I could not have gotten from anywhere else at that time. This opportunity of working in my own startup taught me a lot about how to bridge software engineering practices I knew and real-world requirements that I did not understand yet.

## 8.2 Contributions

The development of the TutorSwap backend provides a stable and secure foundation for a peer-to-peer knowledge exchange platform. One of the most significant contributions is the credit management logic which ensures that earned time is handled atomically, preventing credits from disappearing. The implementation of a stateless authentication system using JWT allows the platform to remain fast and easily scalable, while the integration of STOMP over WebSockets established a real-time communication infrastructure that is resilient enough for university scale. I also contributed a fully automated CI/CD pipeline via GitHub Actions, which effectively bridged the gap between code

and production, allowing for a reliable deployment process that is manageable even for a solo developer.

### 8.3 Future Work

While the current MVP provides a solid start, the evaluation in Chapter 7 highlighted areas where the system must evolve as it leaves the early startup stage. Summarizing that chapter, the current SQL-based matching engine should be replaced with a dedicated search engine like Elasticsearch to handle typos and improve speed for a larger user base. On top of that, while the monolith serves its purpose now, future iterations should explore a shift to microservices to allow high-traffic modules to scale independently. Finally, I would focus on increasing the automated test coverage beyond the current one, since it is a critical priority for future development to ensure that refactoring logic does not lead to regressions.

**REFERENCES**

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, *13*(6), 4-16.

Gillies, R. M. (2016). Cooperative learning: Review of research and practice. Australian Journal of Teacher Education (Online), 41(3), 39-54.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

Fowler, M. (2014). Microservices: a definition of this new architectural term. MartinFowler.com. martinfowler.com/articles/microservices.html

Walls, C. (2016). Spring Boot in Action. Manning Publications.

PostgreSQL Global Development Group. (2025). PostgreSQL 16.1 Documentation: Data Definition and Integrity. www.postgresql.org/docs/current/ddl-constraints.html

Merkel, D. (2014). "Docker: lightweight linux containers for consistent development and deployment." Linux Journal, 2014(239), 2.

Walls, C. (2014). Spring in Action: Covers Spring 4.

Obe, R., & Hsu, L. (2021). PostgreSQL: Up and Running, 3rd Edition. O'Reilly Media.

Rajan, A. P. (2020). A review on serverless architectures-function as a service (FaaS) in cloud computing. TELKOMNIKA (Telecommunication Computing Electronics and Control), 18(1), 530-537.

Allen, C., Li, X., Abdelfattah, A. S., Cerny, T., & Taibi, D. (2023, March). Comparing Cost and Performance of Microservices and Serverless in AWS: EC2 vs Lambda. In Southwest Data Science Conference (pp. 60-72). Cham: Springer Nature Switzerland.

Arora, G., Tayal, A., & Sembhi, R. (2021). Determining the total cost of ownership: comparing serverless and server-based technologies. Deloitte Consulting.

Wen, J., Chen, Z., Jin, X., & Liu, X. (2023). Rise of the planet of serverless computing: A systematic review. ACM Transactions on Software Engineering and Methodology, 32(5), 1-61.

Das, A., Thampi, M. P., Shaik, K., & Kashyap, C. M. (2024, November). Serverless Cloud Computing: Navigating Challenges and Exploring Future Opportunities. In 2024 2nd International Conference on Advancements and Key Challenges in Green Energy and Computing (AKGEC) (pp. 1-6). IEEE.

Stack Overflow. (2025). 2025 Developer Survey. https://survey.stackoverflow.co/2025/

SourceKode. (2026). MERN Stack Future Scope & Job Trends 2026. https://sourcekode.in/blog/mern-stack-future-2026/

Fowler, M. (2012). Patterns of enterprise application architecture. Addison-Wesley.

Kent, W. (1983). A simple guide to five normal forms in relational database theory. Communications of the ACM, 26(2), 120-125.

Rahmatullo, A., Aldya, A. P., & Arifin, M. N. (2019). Stateless authentication with JSON web tokens using RSA-512 algorithm. Jurnal Infotel, 11(2), 36-42

Elastic. (n.d.). Elasticsearch guide. Retrieved October 24, 2023, from https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html.

Morkonda, S. G., Chiasson, S., & van Oorschot, P. C. (2021, November). Empirical analysis and privacy implications in OAuth-based single sign-on systems. In Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society (pp. 195-208).

Fette, I., & Melnikov, A. (2011). The WebSocket protocol (RFC No. 6455). Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc6455

Stumpf, J., Daggett, S., & Norris, B. (2012). Simple Text Oriented Messaging Protocol (STOMP) 1.2 specification. STOMP GitHub. https://stomp.github.io/stomp-specification-1.2.html

Hamari, J., Sjöklint, M., & Ukkonen, A. (2016). The sharing economy: Why people participate in collaborative consumption. Journal of the association for information science and technology, 67(9), 2047-2059.

Rohrs Schmitt, K. (2022). Time banking: Definition, how it works, and examples. Investopedia. https://www.investopedia.com/terms/t/time-banking.asp

Morling, G., Gudian, A., Derksen, S., & Hrisafov, F. (2024). MapStruct 1.6.3 reference guide. MapStruct. https://mapstruct.org/documentation/1.6/reference/html/

Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

GitHub. (n.d.). GitHub flow. Retrieved January 29, 2026, from https://docs.github.com/en/get-started/using-github/github-flow

Docker. (n.d.). Retrieved February 22, 2026 from https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/