FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Open-Source Instant Messaging Platform using Microservices
Student:	Bc. Vladyslav Volodin
Supervisor:	Ing. Marek Suchánek
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Secured and private communication between people through the Internet always depends on the trustworthiness of the service provider. This thesis should design and implement an open-source instant messaging (IM) platform that will be possible to deploy on-premise and adjust security mechanisms.

- Analyse the main security mechanisms related to IM.
- Briefly research the currently widely-used solutions for IM (security as well as other features).
- Set the requirements for the solution and describe use cases to fulfil them.
- Design the solution using a microservices architecture. The architecture must support scalability and extensions (e.g. new security mechanisms, chatbots, or other sendable content).
- Implement the solution based on the design in Go programming language. Briefly describe other selected technologies and justify the choice.
- Document, test, and evaluate the overall solution.

Electronically approved by Ing. Michal Valenta, Ph.D. on 27 August 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Open-Source Instant Messaging Platform using Microservices

Bc. Vladyslav Volodin

Department of Software Engineering
Supervisor: Ing. Marek Suchánek

May 3, 2022

Acknowledgements

I want to thank my supervisor Marek Suchánek for discussing and enhancing the thesis topic with me and for all the feedback he provided while I was writing this thesis.

Furthermore, I want to thank my wife, Faina, and my family for all the support and patience they provided to me during the whole process. They were always there for me when I needed them and supported me over the years in all my endeavors. Thank you for being with me during hard times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 3, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Volodin, Vladyslav. *Open-Source Instant Messaging Platform using Microservices.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce je věnována návrhu a implementaci Instant Messaging platformy, která podporuje škálovatelnost a rozšíření pomocí architektury mikroslužeb. Implementace je provedena v programovacím jazyce Go pro backendové služby a JavaScript pro frontend. Výstupem této práce je prototyp poskytující základní funkce IM a podporující end-to-end šifrované zprávy mezi dvěma účastníky.

Klíčová slova instant messaging, mikroslužby, webová aplikace, Go, chi, JavaScript, ReactJS, MongoDB

Abstract

This work is devoted to designing and implementing the Instant Messaging Platform that supports scalability and extensions using the microservices architecture. The implementation is done in Go programming language for the backend services and JavaScript for the frontend. The output of this work is the prototype providing the essential IM features and supporting the end-to-end encrypted direct messages between two participants.

Keywords instant messaging, microservices, web application, Go, chi, JavaScript, ReactJS, MongoDB

Contents

List of Figures

Listings

Introduction

Communication is one of the most necessary parts of a human's life. Every day we communicate, discuss, and interoperate using the natural language to achieve our career goals or stay in touch with people close to us.

Instant Messaging helps people living in different parts of the globe stay connected for personal and business purposes. So, nowadays, thanks to it, there is no need to overuse business trips in most cases. Moreover, families can stay in touch even when living in different parts of the planet.

There are privacy and security concerns related to Instant Messaging and ways to prevent them. Because of that, it is crucial to have Instant Messaging platforms compliant with them and make it easy to prove this compliance.

Because such services are essential for people across the globe, they need to be highly available and quickly scalable on demand. The ease of maintenance and extending is essential as well to be able to react to frequently changing security requirements and possible leaks.

The output of this thesis will be beneficial as a service implementing security and privacy mechanisms for users who care about it.

Chapter 1

Goals and Structure

This thesis aims to design and implement a scalable, reliable, and extendable Instant Messaging Web Service that will be possible to deploy on-premise and adjust security mechanisms.

It is needed to analyze the primary security mechanisms related to Instant Messaging, briefly research the currently widely-used solutions for IM, including the security mechanisms used there.

It is necessary to set the requirements for the service and describe use cases to fulfill them.

The author will design the service using the microservices architecture to support scalability and extensions like new security mechanisms, chatbots, or other sendable content.

The other goal is to implement the solution based on the design in the Go programming language and briefly describe other selected technologies and justify the choice.

Another important topic is to document, test, and evaluate the solution according to requirements.

The first part of the thesis describes and analyzes Instant Messaging Platforms and general technologies and approaches they use. The next analyzes the security, privacy, and security mechanisms related to Instant Messaging. The following part is the research of existing widely-used solutions. The next is the analysis and design of an Instant Messaging service, which will be the output of this thesis. After that, the thesis will describe the implementation of the solution, testing, documentation; and will justify the technologies choice. The final part of the thesis is the evaluation of the implemented platform.

Chapter 2

Instant Messaging Platforms

Instant Messaging, in general, refers to text-based, real-time communication carried out between two or more people over a digital network. [9]

Nowadays, Instant Messaging Platforms have an unbelievable amount of monthly active users. When we calculate a sum for the most popular ones like WhatsApp [10], WeChat [11], Facebook Messenger [12], QQ [13], Snapchat [14], and Telegram [15], we will get 5.9 billion users as of January 2022. And the leader is WhatsApp, with its 2 billion monthly active users. [3]

Given that fact, we can conclude that the IM platforms are highly demanded and crucial for people all over the globe. Obviously, with such an amount of monthly active users, the system that serves requests should be easily scalable and optimized. This chapter describes the Instant Messaging Platforms and standard technologies used to make them reliable and possible to exist.

2.1 Importance of IM Platforms

Instant Messaging Platforms found their usage not only in personal lives. During the last 30 years, after it developed in the 1990s, the IM Platforms became a standard for communication inside organizations over the globe. One of the reasons for such success is the possibility to communicate with clients and colleagues quickly, privately, and on the fly. One of the other essential features is that people can use Instant Messaging systems to talk to their families, friends, co-workers, meet new people, build relationships, join discussions and chat rooms by their interests, hobbies, etc. In this way, people can talk to anyone in the world. [9]

2.2 Networking Models

Networking models' purpose is to connect computers over a network. The most popular models are client-server and peer-to-peer. [16]

2.2.1 Client-Server

The client-server model idea is the relationship of so-called clients and servers. The client is the computer that requests the information, and the server is the computer that serves such requests and sends the response to the client with the requested information, as you can see in figure 2.1. [16]

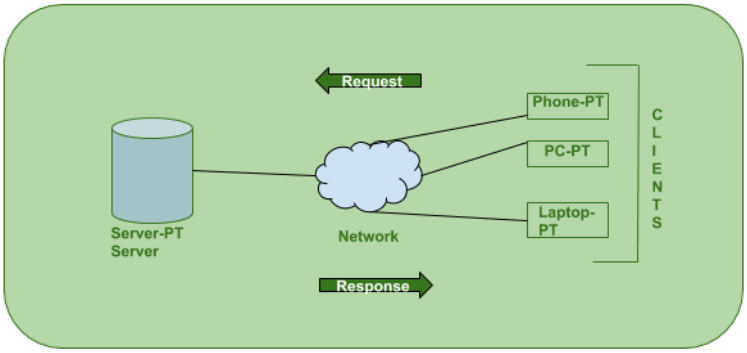

Figure 2.1: Client-Server model illustration [1]

According to Ajay Sarangam [17], the client-server architecture has the following advantages and disadvantages:

Advantages

- The data is centralized within the system that is maintained in a single place.
- The model is efficient in delivering resources to the client and also requires low-cost maintenance.
- It is easy to manage, and the data can be easily delivered to the client.
- As the data is centralized, this system is more secure and serves added security to the data.
- Within this type of model, more clients and servers can be embedded into the server, which makes the performance outstanding and increases the model's overall flexibility.

Disadvantages

- Clients' systems can get a virus or any malicious scripts if any are running on the server.
- Extra security must be added so that the data does not get spoofed in between the transmission.
- The main problem can be server down. When the server is down, the client loses its connection and will not access the data.

2.2.2 Peer-to-Peer

In a Peer-to-Peer network, there is no centralized provider like the server responsible for delivering the content. All participants store some data, and communication happens directly between all participants, so-called peers [16]. So, in terms of Instant Messaging, it is advantageous since the sender and recipient will not worry about some server storing and processing their data. But this should be implemented with End-to-End encryption because sending the messages in plain text is not secure — more about this in Chapter 3 which focuses on security and privacy aspects.

Shane Avron defined the advantages and disadvantages of Peer-to-Peer networks [18]. The following list is the version adapted for Instant Messaging Platform usage:

Advantages

- Low latency — With low latency come better response times and shorter waiting times between requests. The length of the connection path that is between peers is reduced, making the network more efficient by eliminating redundant steps on the way towards the final destination.
- High bandwidth — A peer-to-peer network provides you with high bandwidth, so there is no need for central servers to provision resources.
- Low cost — Due to the fact that there is no central server in a peer-to-peer network, each peer is responsible for storing and sending the requested information. There are no fees charged by the server that is hosting the application.
- Decentralization — In a client-server architecture, the company that owns the server controls its users. It can monitor user activity and delete information. That is not the case when it comes to peer-to-peer networks, which let users control their own data.

Chapter 3

Security and Privacy in IM

Privacy and security have become extremely important and valuable nowadays thanks to the spreading of Information Technologies, and it is becoming an essential part of a human being. The proper handling and usage of data are critical nowadays. Experienced hackers make numerous cyberattacks, but it is not the only risk for personal data. Many leaks are happening because of inappropriate usage and storing of data. Alternatively, companies owning free-to-use services with poor privacy policies and trading with personal data are the other problem. This chapter will describe the Security and Privacy in Instant Messaging Platforms and cover some mechanisms for achieving it.

3.1 Security

According to Collins dictionary, security refers to all the measures taken to protect a place or ensure that only people with permission enter it or leave it '[19]. In terms of Instant Messaging, the secure messaging system should control the access to the messages and any other personal information users send to each other to prevent unauthorized access. The ideal state is when only the sender and recipient can access the message content and metadata. For example, we can achieve it by using peer-to-peer technology and end-to-end encryption, which will be covered later in this chapter. However, both bring a certain level of complexity and inconvenience for users who use or want to use more than one device for participating in communications. The ideal product in terms of security contradicts the ideal product in terms of usability. That is why designing and implementing an Instant Messaging Platform is mostly about finding the acceptable trade-off between a sufficient level of security and a user-friendly application.

No one wants their private communication to become public. All people have their reasons for that. Private business communication between top management about the brand new feature or product should not leak to third parties or hackers for apparent reasons — it is the situation when the company

probably loses its money, reputation, and leadership on the market. Even if industrial espionage does not sound like a problem for full-time employees or families, there are still cases when the leaked data may harm them. For example, when the person is sharing their billing information, addresses, and vacation plans, the leak may potentially harm them, the people close to them, or their property. Another example — is sharing with the family the plans to change the employer for some complicated reasons. The leak of this communication may harm the company and them directly because of the contract, including the non-disclosure part.

There is a wide range of mechanisms to secure the product, whether it is a web service, platform native application, or web application aggregating the data from third parties. Next, it is crucial to cover some of them applicable for Instant Messaging Platforms and will be used in implementation later.

3.1.1 Security Mechanisms

The Instant Messaging systems store personal information and act with it. Examples of such information are email addresses, phone numbers, passwords, full names, messages content, and attachments. The easiest recommended and the most common way to secure such information is encryption of the data and the channel used to transfer it. [20]

3.1.1.1 SSL/TLS Encryption

Initially, the network data were transferred as plain text, allowing everyone to inspect the content. SSL inventors created it to correct this problem and protect user privacy. [21]

SSL (Secure Sockets Layers) and TLS (Transport Layer Security) are cryptographic protocols to secure the data-transferring channel. They provide authentication and data encryption between servers, machines, and applications operating over the network (client-server architecture). Both SSL 2.0 and 3.0 are deprecated, and there are many discovered vulnerabilities. However, since the TLS protocol is based on SSL and replaced it, there are still some people who accidentally refer to the TLS protocol as "SSL." [22]

TLS encrypts all the data transmitted across the web. So, anyone who tries to read the content can see only the meaningless symbols unless they have the key for decryption. In addition, it provides authentication through the handshake and the data integrity, verifying that no one modified the data during the transportation. [21]

3.1.1.1.1 Man-in-the-Middle attack The widely known attack is the man-in-the-middle attack, where the malefactors position themselves in a conversation between the participants, as displayed in figure 3.1, and read all the content sent between them to still their personal information [2]. In terms

of the encrypted connection, the user connects to the malefactor's server and establishes the secured connection, thinking it is the original server. Thanks to the SSL/TLS authentication through the handshake using the certificate issued by the trusted Certificate Authority, the TLS protocol is resistant to this attack. [23]

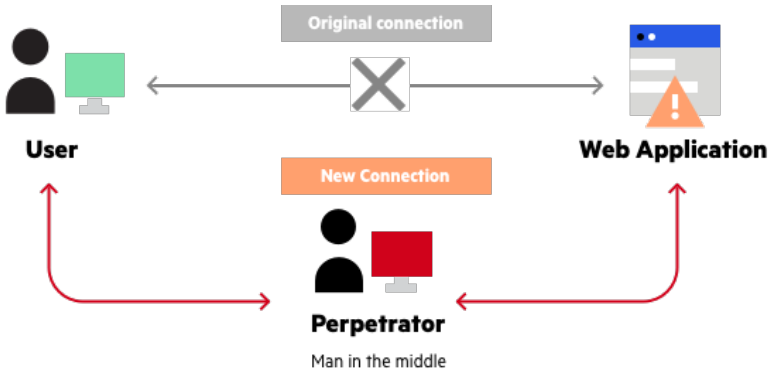

Figure 3.1: Man-in-the-middle attack illustration [2]

3.1.1.2 Storing the Passwords

All Instant Messaging systems have some authentication. The implementation often relies on the phone number and one-time codes sent via SMS. Another possible way to authenticate the user is to rely on the pair username/email and password. There is a requirement for services storing user passwords to store them securely. Storing them in plain text is not an acceptable option. The commonly used mechanism combines hashing and salting the passwords to complicate the cracking.

3.1.1.2.1 Hashing Hashing is the process of applying the one-way hashing function to generate a hash value. The application then stores this value instead of the original password and, during the authentication process, hashes the value that is coming from the user and compares two hashes. [24]

3.1.1.2.2 Salting Salting is a concept widely used with hashing to add an additional layer of security to hashing process, especially against brute-force attacks. Essentially, salt is a unique value that can be added to the end of the password to create a different hash value. [24]

3.1.1.3 End-to-End Encryption

End-to-end encryption guarantees a secure data exchange between two endpoints [20]. In terms of Instant Messaging Platforms, only the sender and the

recipient may access the sending message and its content.

For example, with Open Whisper System's Signal Protocol [25], in order to send the encrypted message, the sender should first get the recipient's public keys and combine them with their private keys to generate the shared master key. Then the sender sends the message encrypted with this shared key to the recipient. After receiving the message, the recipient does the same work — gets the sender's public keys and combining them with their private keys generates the same shared master key and decrypts the message. Participants continue to use this master key for communication between them. This process is possible using an extended version of the Diffie-Hellman key exchange. [26]

3.2 Privacy

Privacy is a human right that Warren and Brandeis described in 1890 as the right of the individual to be free from intrusion [27]. Later in 1983, Stone et al. defined informational privacy as "The ability of the individual to personally control information about one's self." [28]

"The Internet is the primary environment for informational privacy, as this is where most information is transferred, collected, and stored." [29]

In order to protect users' privacy on the internet, specific legislations such as the California Consumer Privacy Act (CCPA) and the General Data Protection Regulation (GDPR) have been undertaken [29]. So, every company that does business in California or has users from California should follow the rules of CCPA. A similar situation is for European Union countries and GDPR.

As mentioned in Security 3.1 section, the examples of information the Instant Messaging platform operates with are email addresses, phone numbers, passwords, full names, messages content, and attachments. Also, we need some classification to address this data and find the best strategy to ensure that users' privacy is secure and we use the personal data the proper way.

3.2.1 Personal Information

There are two different classifications of data that often confuse companies that collect users' data — personally identifiable information (PII) and personal data. The United States uses PII, but there is no legal document that defines it. However, personal data has the legal meaning defined by GDPR, and European Union accepted it as law. [30]

3.2.1.1 Personally Identifiable Information

Even when Personally identifiable information is not defined in any legal document [30], the United States Government Accountability Office provided a

well-describing definition of PII for purposes of their report — "Any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individual's identity, such as name, social security number, date and place of birth, mother's maiden name, or biometric records; and (2) any other information that is linked or linkable to an individual, such as medical, educational, financial, and employment information." [31]

3.2.1.1.1 Linked and Linkable Information PII consists of linked and linkable information. Both kinds are essential, and the system should manipulate them properly without allowing unauthorized access.

Linked information is the information directly associated with the individual and makes it possible to identify them [30]. For example:

- Full name;
- Home address;
- Social security number;
- Credit card number;
- Date of birth;
- IP or MAC address;
- Cookies.

Linkable information is not enough to identify a person's identity, but combining it with other information can identify them [30]. For example:

- Common first or last name;
- Country, state, city;
- Gender;
- Race;
- Job position;
- Non-specific age (e.g., 30–40 instead of 30).

3.2.1.2 Personal Data

Personal data is a legal term, and according to the definition coming from GDPR, it "means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person." [32]

Besides the most prominent examples from linked and linkable information, the definition of personal data covers even such information as: [30]

- transaction history,
- browser history,
- posts on social media.

3.3 Open Source

Implementing the Instant Messaging Platform according to the last security best practices and recent law regulations is not enough to make the customers sure that the service respects their security and privacy. Customers' trust is essential for every service. One of the most effective ways to convince customers and build trust is to make the code base Open Source. With such an approach, everyone can look at the implementation and check all security mechanisms and approaches used in the implementation.

According to Rad Hat, open source software is the code designed to be publicly accessible, so anyone can read it, make any modifications, and distribute it. [33]

The Open Source Initiative provides a more concrete definition of open source, accessible on its website. According to it, open source does not just mean access to the source code by any individual, it defines that the distribution of the open source software must comply with the following criteria: [34]

1. Free redistribution.
2. The program must include the source code.
3. The license must allow modifications and derived works.
4. Integrity of the author's source code.
5. No discrimination against persons or groups.
6. No discrimination against fields of endeavor.

7. The rights attached to the program must apply to all to whom the program is redistributed.
8. License must not be specific to a product.
9. License must not restrict other software.
10. License must be technology-neutral.

Chapter 4

Existing IM Platforms

There are a lot of solutions implementing the Instant Messaging platforms on the market. All of them have different purposes, advantages, and disadvantages. This chapter will cover some of them, including the security mechanisms.

4.1 Popularity

According to the statistics from DataReportal [3] already mentioned in Chapter 2, we can draw chart 4.1 comparing the popularity of the most popular IM Platforms as of January 2022.

As you can see, WhatsApp [10] is the most popular platform. It is 63.15 % more popular than the Chinese IM platform WeChat [11], more than two times more significant than the Facebook messenger [12], and almost four times more popular than other mentioned services.

For this thesis, the author chose the following implementations:

- WhatsApp [10] — the most popular IM platform;
- Telegram [15] — extensible and, according to its pages — private and secure;
- Signal [35] — combines both security and privacy on a high level [36];
- Slack [37] — an example of a business solution for Instant Messaging designed for communication inside a company;

The following sections will cover them in more detail.

4.2 WhatsApp

WhatsApp is the Instant Messaging platform created by Brian Acton and Jan Koum and initially released in January 2009 without the Instant Messag-

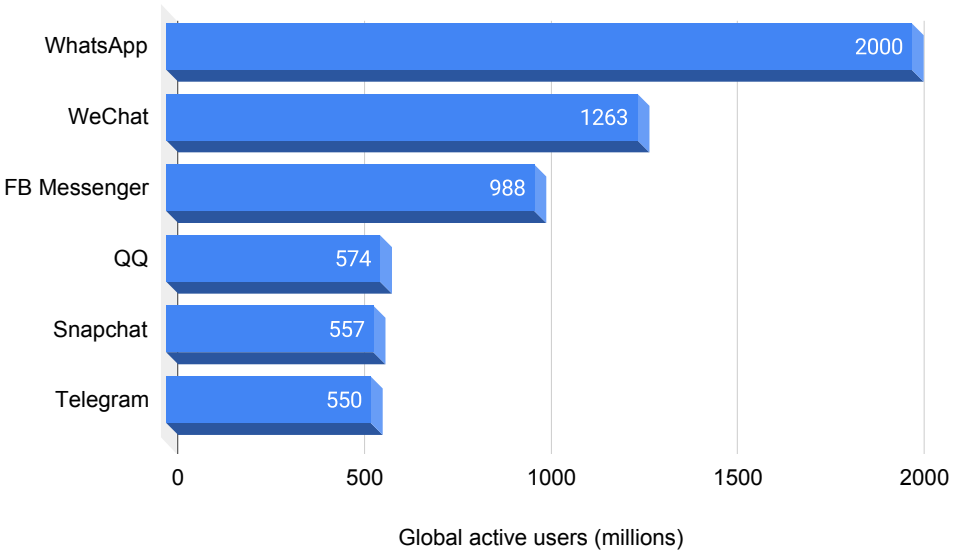

Figure 4.1: Ranking of selected Instant Messaging Platforms [3]

ing features, which were a part of future releases. Later in February 2014, Facebook acquired WhatsApp for 19 billion US dollars. [38]

But later, after the acquisition, on February 8th, 2021, WhatsApp announced the change in its terms of service and privacy policy. The biggest concern was about how the company processes the user data. More specifically, WhatsApp can use Facebook hosted services to store and manage chats and partner with Facebook to integrate itself into Facebook Company Products. This change allows WhatsApp to share users' information like registration, phone numbers, transaction data, interactions, IP addresses, and similar with other Facebook companies. This data usage violates the GDPR, so this change does not apply to EU countries. Another important fact is that if the user does not want to accept such policies, WhatsApp will make their accounts inaccessible. [39]

In figure 4.2 you can see the application UI. The view is separated into two columns: the left one with chats and the right one with chat messages. At the bottom of the messages view, there is a place for typing a message. And at the top, there is information about the open chat. This layout is typical for most Instant Messaging platforms, and as you will see in other implementations, all applications are similar in this sense.

4.2.1 Platforms

As of April 3rd, 2022, WhatsApp supports the following platforms [4]:

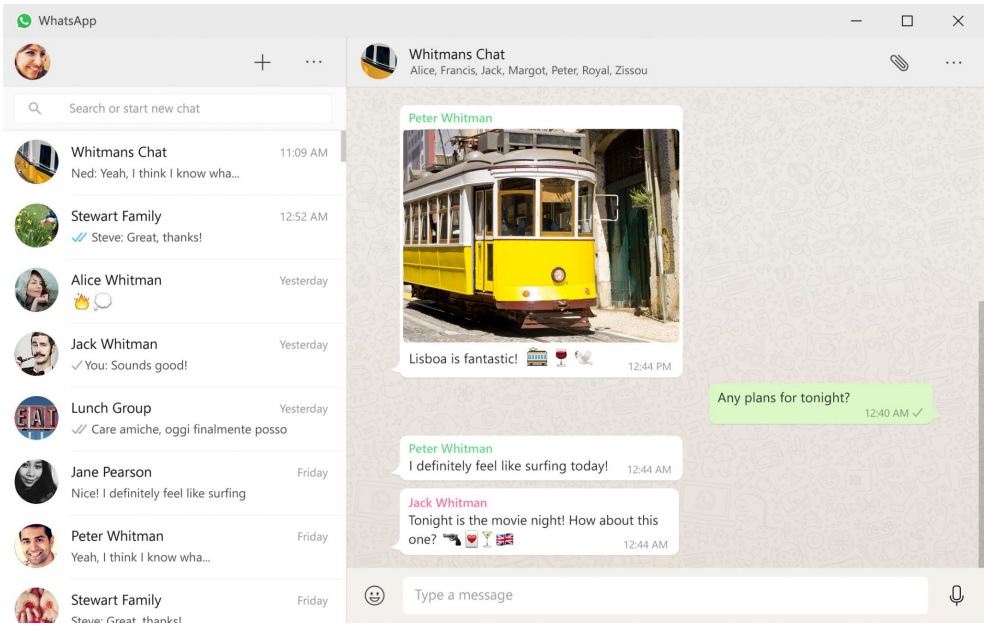

Figure 4.2: WhatsApp application UI [4]

- iPhone,
- Android,
- macOS,
- Windows,
- Web.

Unfortunately, there is no solution for iPad or Linux, so users need to use a Web application. There are some third-party implementations of WhatsApp clients. However, using them is forbidden by the company, and the account may be permanently banned. [40]

4.2.2 Features

Previously, WhatsApp did not support having independent devices except the user's mobile phone. The user was able to connect their devices to the mobile phone using a QR code and communicate through it. However, when the phone went offline, the user was forced to do the connection procedure again. But later, in July 2021, the company started beta testing to support multiple devices fully, and it works now. [41] [42]

Nevertheless, it still has some limitations. For example, the "linked device" will be logged out if the user does not use their phone for more than 14 days. And some other features are not supported on "linked devices" as of April 3rd, 2022: [42]

- Clearing or deleting chats if the primary device is an iPhone.
- Messaging or calling someone using an old version of WhatsApp on their phone.
- Viewing live location.
- Creating and viewing broadcast lists.
- Sending messages with link previews from WhatsApp Web.

The set of essential features WhatsApp has that the author considers crucial for a successful Instant Messaging platform is following:

- End-to-end encryption,
- Group chats,
- Support for multiple devices,
- Sharing the different additional content like images, videos, documents, and similar,
- Voice messages,
- Video and voice calls.

4.3 Telegram

Telegram is the Instant Messaging platform initially launched in 2013 by the brothers Nikolai and Pavel Durov, who invented the Russian social network VK in October 2006. In figure 4.3 you can see the application UI.

4.3.1 Platforms

As of April 3rd, 2022, Telegram supports the following platforms [43]:

- iPhone,
- iPad,
- Android,
- macOS,

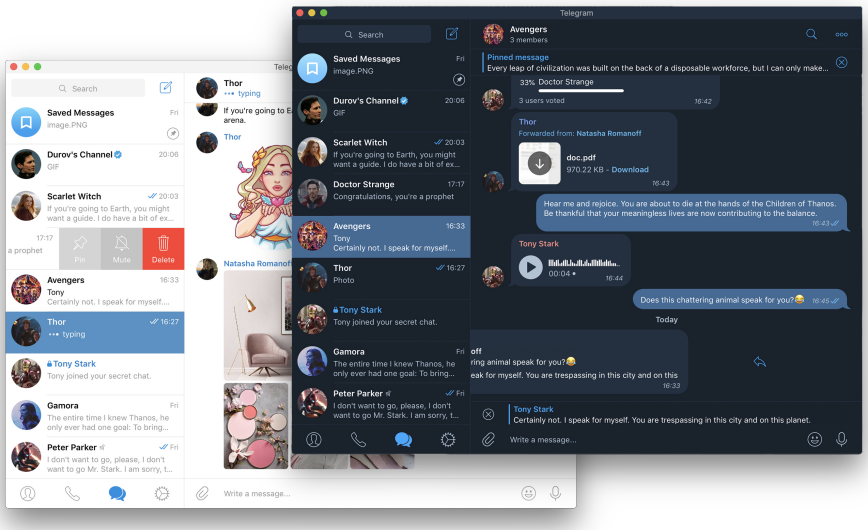

Figure 4.3: Telegram application UI [5]

- Windows,
- Linux,
- Web.

In addition, Telegram designed the Telegram Database Library [44] to help third-party developers create their custom applications using the Telegram platform. This approach differs from forbidding third-party clients as WhatsApp has.

4.3.2 Features

Telegram has a wide range of features that make it stand out among competitors:

- Channels — a form of one-way messaging where admins can post messages for a wider audience,
- Bots — Telegram accounts operated by programs that react to user messages and commands, accept online payments, and may be helpful in any other scenarios since the Telegram team provides an API and an easy way for creating bots,

- Telegraph — publishing tool for creating formatted article pages with embedded media,
- Video and voice calls,
- Voice messages,
- Sync between devices,
- Sharing the different additional content like images, videos, documents, and similar,
- Open source.

As for personal chats, the situation is not as straightforward as someone wanted it to be. By default, Telegram saves all communications on its server, and such chats are not end-to-end encrypted. If the user wants end-to-end encrypted communication, they need to initiate a so-called Secret chat. With this approach, the user will have two conversations with the same recipient, which may be confusing. In addition, the single Secret chat is accessible only on the devices that initiate it, so there is no access on other devices. As for advantages, users may easily remove the whole communication and send self-destructing messages.

4.4 Signal

Signal is an encrypted Instant Messaging platform developed by the non-profit Signal Foundation and Signal Messenger LLC, initially released on July 29th, 2014. It is a result of merging the RedPhone [45] and TextSecure [46] applications, and it is known and popular for its security and privacy policy [47]. Signal was the first iOS app that enabled end-to-end encrypted voice calls for free [48]. In figure 4.4 you can see the application UI.

4.4.1 Platforms

As of April 3rd, 2022, Signal supports the following platforms [6]:

- iPhone,
- iPad,
- Android,
- macOS,
- Windows,
- Linux.

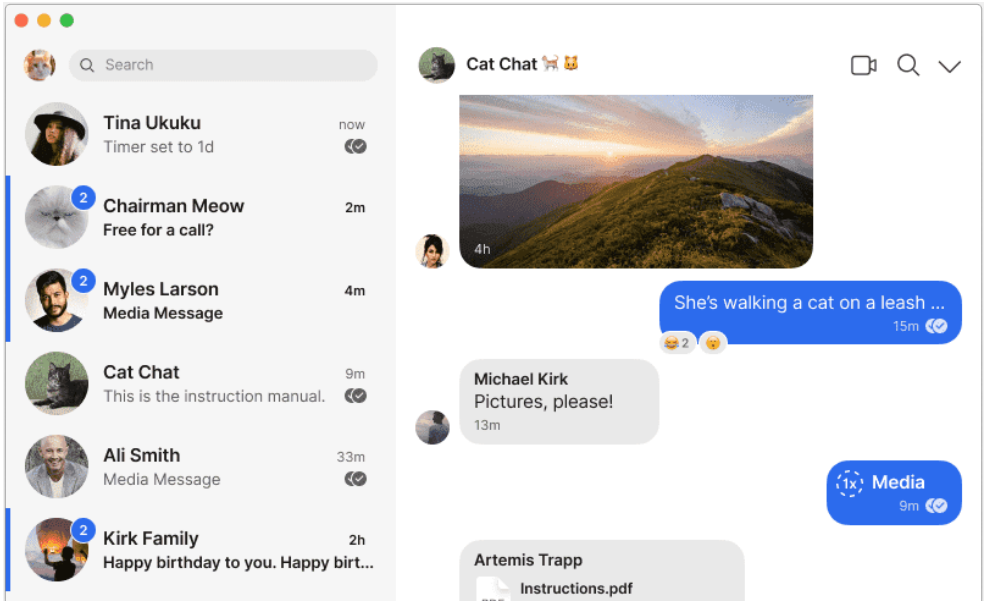

Figure 4.4: Signal application UI [6]

4.4.2 Features

Signal implements all features essential for the Instant Messaging platform and does it without ignoring the users' privacy and security:

- Personal and group chats,
- End-to-end encryption,
- Support for multiple devices,
- Sharing the different additional content like images, videos, documents, and similar,
- Video and voice calls,
- Voice messages,
- Open source.

Signal does not store users' messages — after applying the end-to-end encryption, the client sends the message to the Signal servers, which store it in queues until all devices receive it. But this may cause unexpected problems in edge cases. For example, when the person is on vacation, the queue for their desktop may overflow, resulting in missed messages.

4.5 Slack

Slack is a business communication platform developed by Slack Technologies, initially released in August 2013. Many companies use it to provide convenient and effective communication inside the company and between teams. In figure 4.5, you can see the application UI.

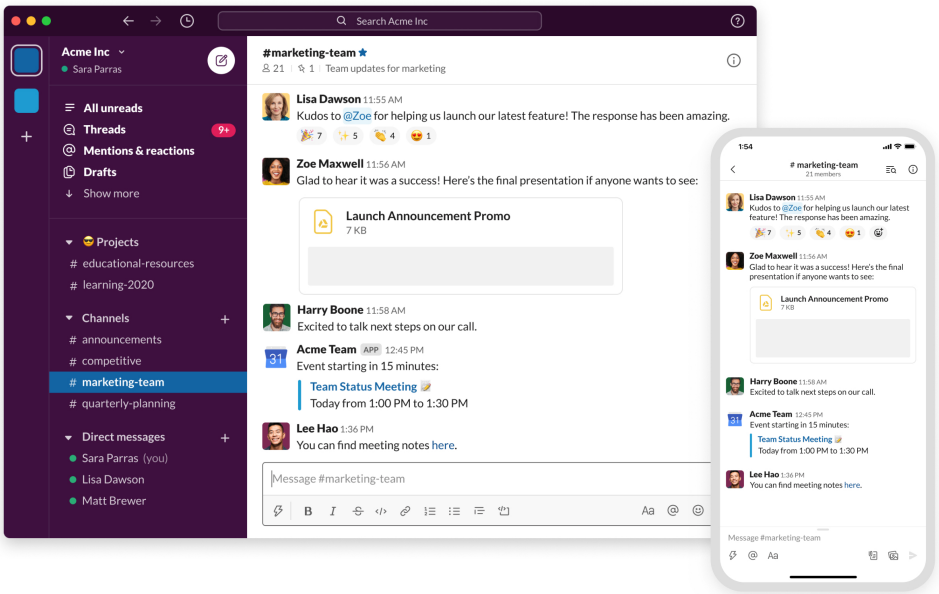

Figure 4.5: Slack application UI [7]

4.5.1 Platforms

As of April 3rd, 2022, Slack supports the following platforms [49]:

- iPhone,
- iPad,
- Android,
- macOS,
- Windows,
- Linux,
- Web.

4.5.2 Features

Slack offers many IRC-style features, for instance:

- Persistent chat rooms (channels) organized by topic,
- Private groups,
- Direct messaging,
- Sharing files.

Chapter 5

Analysis and Design

In this chapter, the author will go through the requirements and use cases to fulfill them during the implementation. In addition, the author will design and present the platform architecture, required entities models for each service implemented using the micro-service architecture, and design the chat application User Interface.

5.1 Requirements Analysis

This section aims to gather and formalize the functional and non-functional requirements for the Instant Messaging Platform implemented in this thesis. The author will later use the output of this section as the base for defining the Use Cases.

5.1.1 Functional Requirements

For the implementation of the Instant Messaging Platform prototype, which is the goal of this thesis, the author considers the following functional requirements as the essential ones:

1. **Users Search** — the user should be able to find other users knowing their username and initiate the communication with them — the chat page will provide the field to search for users and the button to create a new chat next to it.

2. **Display the conversations** — the application will provide the list of conversations the user participates in as one of the essential parts of the UI.

3. **Create an account** — the new user should be able to create an account without providing any sensitive or personal information — the user who

is not registered will see only the registration page with only two required fields — username and password.

4. **View the conversation history** — the user should be able to access the conversation history — when scrolling up the conversation history, the old messages will appear.

5. **Real-Time messaging** — the user should be able to access the new messages without the page refresh — all messages sent after the initial loading of the application will reach the user's device in real-time.

6. **Highlight new messages in conversations** — the fact of new messages reaching the client should be visible on the conversation level — the application will highlight the conversations containing new messages.

7. **Open the conversation** — the user should be able to display the messages related to the concrete conversation — when clicking on the concreate conversation, the chat with messages will appear.

8. **Send the message** — the user should be able to send the message to the selected conversation.

5.1.2 Non-Functional Requirements

Besides the functional requirements that base on features visible to general users, it is crucial to define the non-functional requirements necessary for the system.

1. The application should be accessible with the standard Web Browser using the HTTP protocol.

2. The communication between participants should be end-to-end encrypted.

3. The systems should store only the essential personal information needed to operate.

4. The application should be reliable and behave smoothly.

5. The application design and architecture should support the possible extensibility and scalability.

6. The application should use the microservices architecture.

Regarding the personal information item 3, the idea of the Instant Messaging Platform implemented in this thesis is to store only the username, which may be any value user prefers to use. In this case, the system will not store emails, phone numbers, real names, or other personal information.

There are the mechanisms such as pagination, limiting the amount of data sent in each HTTP request and response, using asynchronous calls on the frontend, and others to achieve reliability and smooth behavior, as mentioned in item 4. Reaction to the growing number of user requests to maintain reliability is possible using horizontal scaling, which the thesis does not cover but designs services to support it

5.2 Use Cases

In this section, the author will describe how users will perform tasks using the application in so-called use cases. There is only one role — the user who uses the Instant Messaging Platform. The use cases are based on functional requirements defined in the previous section. Each of them has the following short description. The reader can see the Use Cases diagram in figure 5.1.

UC1 Finding other users

- The user will start typing the username in field.
- The user can select the desired recipient from suggestions and click on it.

UC2 Initiating the communication with the concrete user

- The user will select the desired user (**UC1**).
- Will click on the adding button located near the search field.

UC3 Display the conversations the user participates in

- The user will open the main application page.
- They can see the conversations on the left side of the screen.

UC4 Create the account without providing any sensitive or personal information

- The user who is not registered will open the application, and the application will redirect them to the authentication page.
- The user will fill username and password.
- The user will click the register button, and the application will redirect them to the main page.

UC5 Displaying the conversation messages

- The user will click on the conversation located on the left side of the screen.
- The conversation messages appear on the right side of the screen.

UC6 Accessing the conversation history

- The user will display the conversation messages (**UC5**).
- The user will scroll up, and old messages will appear when the user continues to scroll.

UC7 Accessing the new messages for current conversation without refreshing

- All new messages appear at the bottom of the conversation messages.
- The user needs to scroll down if not at the bottom.

UC8 See new messages on the conversation level

- The conversation on the left side will indicate that the new message has arrived.

UC9 Accessing the new messages for other conversations without refreshing

- The user will see an indication in the conversations list on the left side of the screen showing that a new message arrived (**UC8**).
- The user will click on the conversation and will see new messages.

UC10 Sending the message

- The user will select some conversation (**UC5**).
- The user will type the message to the field at the bottom of the chat view.
- The user will click the "Send" button.

5.3 Architecture

To fulfill all functional and non-functional requirements, the author designed the application with three services — two of them use databases to store the data, and one without any data store. The reader can see the architecture diagram displayed in figure 5.2.

The Users Service and Chat Service are stateless services that can quickly and infinitely scale horizontally without any risk of breaking the application logic, as in the case of stateful services.

Unlike them, the Delivery Service is stateful. It stores the information of the connected devices in the in-memory map to deliver real-time messages to them. However, it does not imply the inability to scale horizontally. The horizontal scaling only requires the more intelligent sharding approach like, for example, MongoDB [50] has. So, to make the Delivery Services scale

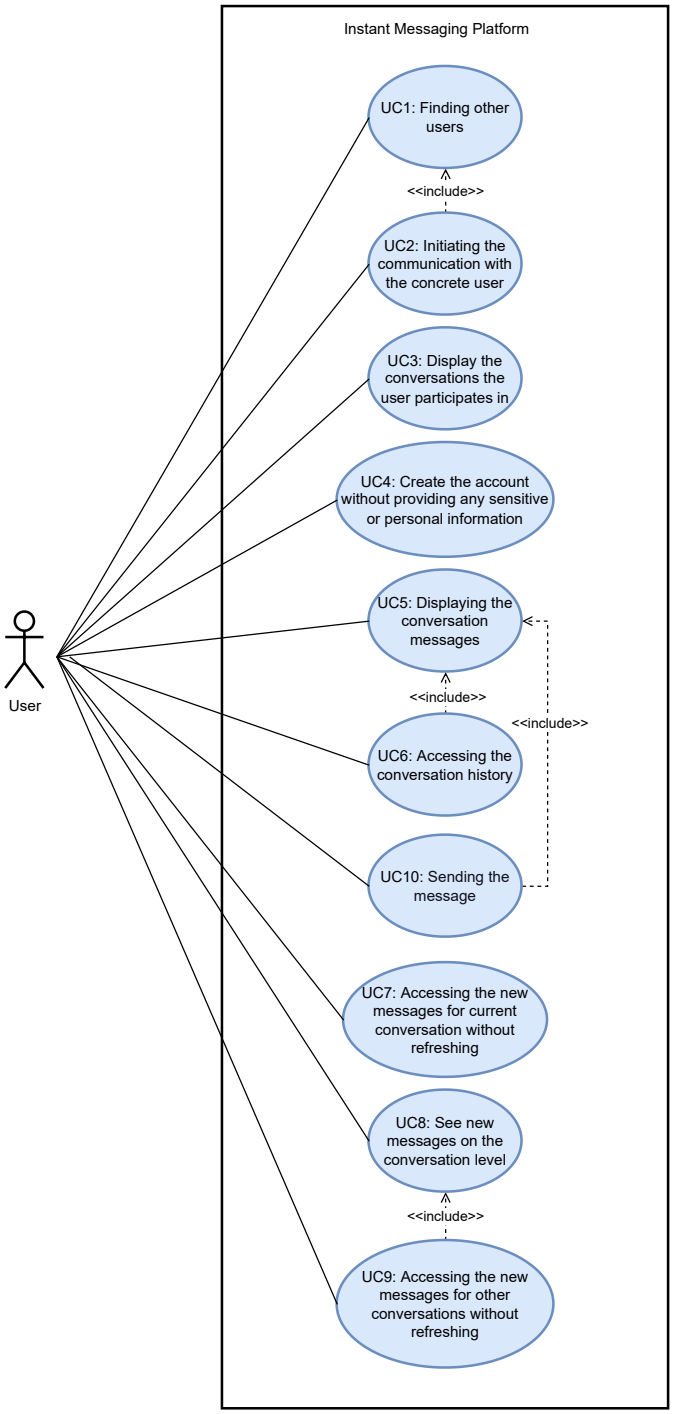

Figure 5.1: Use Cases diagram

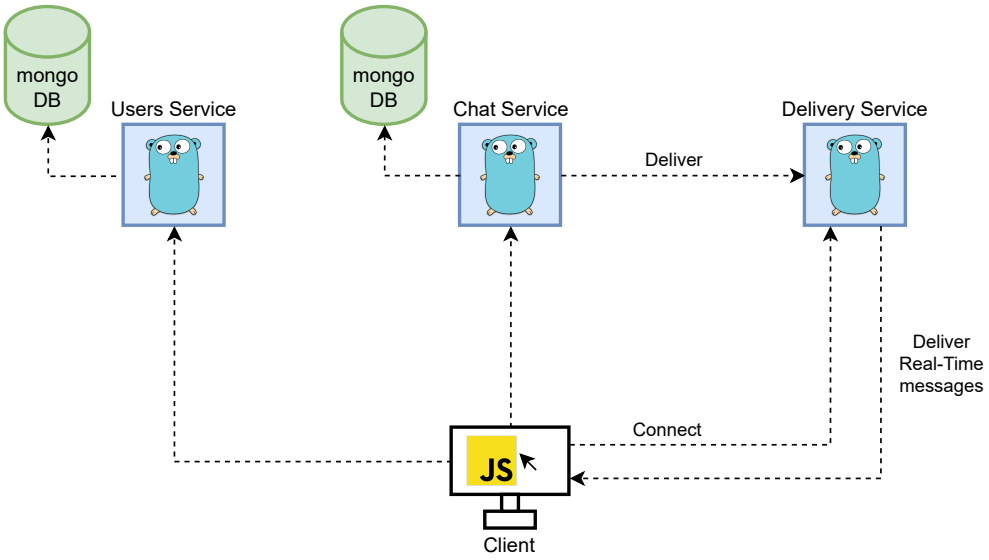

Figure 5.2: Architecture diagram

horizontally, it is enough to implement the reverse proxy, which will guarantee to send the requests for a particular user to the same Delivery Service instance both for Chat Service and Client calls.

Even if the actual horizontal scaling is out of the scope of this thesis, these facts show that the platform components support it by design and can scale independently. Thanks to it, the thesis achieved one of its goals.

With this architecture design, the author was trying to separate related logic to multiple services to split a load of parts with different usage frequencies.

The author will describe the Users Service, Chat Service, and Delivery Service responsibilities and models in the following subsections.

5.3.1 Users Service

The Users Service is responsible for handling the new users' registration requests, generating API access tokens, and storing the users' public keys for end-to-end encryption purposes by providing the API for registered users to access them.

Its model is displayed in figure 5.3 and contains only one entity. "User" is the entity storing the user's username, password, and public key, as mentioned before. The thesis will describe the database model in more detail in the Implementation 6 chapter.

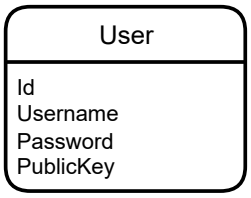

Figure 5.3: Model of the Users Service

5.3.2 Chat Service

The Chat Service is responsible for all the Instant Messaging Platform operations and covers the rest of the requirements. It is also responsible for sending the request to Delivery Service for every new incoming message to provide the real-time messaging feature. Figure 5.4 displays the Chat Service model. It is different from the actual database model because of additional fields used for faster queries and indexes, and the Implementation 6 chapter covers these details.

The implementation does not use foreign keys or any object or document references. It deals with three independent collections of documents storing the values, and the service objects do not use any joins. That is why there are no references in the Chat Service model.

The User entity stores only the user's username and a list of conversations documents with a relatively simple structure: each document contains only the id and type of the conversation. The current implementation provides only the private conversation type, which the Implementation 6 chapter also covers.

The Conversation entity is simple as well. It stores only the list of participants' usernames, so the client can access them when needed using the API secured by the access token validation.

The Message entity does not define any message structure. It allows the Client application to fully handle it, use the end-to-end encryption to encrypt it, and store it on the server since it is also responsible for displaying the message. That is why the Content field is just an array of bytes. The From field specifies the sender of the message to retrieve the sender's public key from the Users Service to decrypt the shared encryption key provided with the initial message and display the message author on the Client side. The last field is the ConversationId used for retrieving the messages for concrete conversation.

5.3.3 Delivery Service

Delivery Service does not operate with structured data, and its purpose is clear from this section — delivery of the real-time messages. However, it is

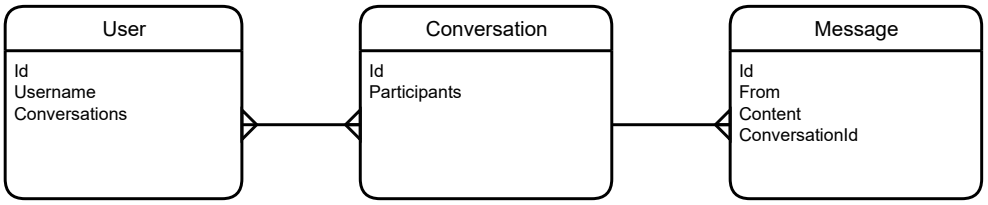

Figure 5.4: Model of the Chat Service

still worth mentioning because of its usage.

Before the client can receive the real-time messages, it needs to register itself in Delivery-Service and initiate a one-way communication from the Delivery Service. After that, the Delivery Service sends all messages coming to this user after getting them from the Chat Service using the endpoint secured by another short-time access token issued by the Chat Service for its usage.

5.4 User Interface

Figures 5.5 and 5.6 display the wireframes of the User Interface of the application main and login pages, which cover mentioned functional requirements.

On the main page, there is a header containing the search field and button for adding new conversations and the username of the currently logged-in user. There is a list of conversations the user participates in on the left side, and on the right side, the messages for the currently selected conversation. There is a field and the button for sending the message at the bottom of the currently opened chat.

The login page has only necessary fields for registration - username and password, as mentioned in the requirements.

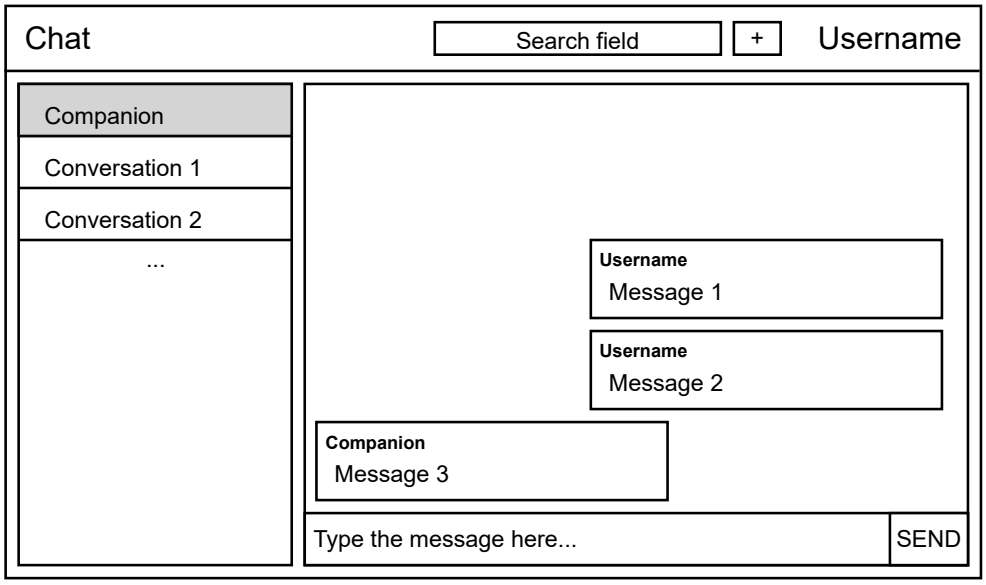

Figure 5.5: Main page UI wireframe

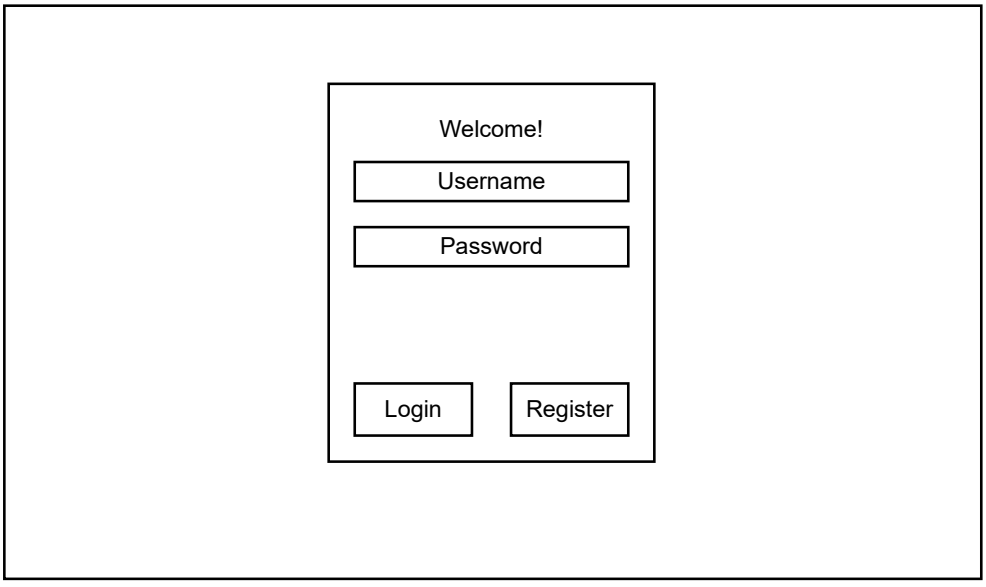

Figure 5.6: Login page UI wireframe

Chapter 6

Implementation

This chapter will present and describe the implementation of the solution using the microservices architecture written in the Go programming language. Furthermore, it will briefly describe the technologies used during the implementation and justify the choice. This chapter describes the essential parts of the implementation and does not cover the whole codebase. In order to explore the source code, please see the attached source files.

6.1 Technologies

6.1.1 Go Chi

When coming to web service implementation, it is essential to decide about the application's technology to create the router responsible for routing the requests to responsible handlers.

According to the official GitHub page, *chi* [51] is a lightweight, idiomatic and composable router for building Go HTTP services. It provides an elegant and comfortable design for writing API servers [52]. The author used the following modules from the *chi* project: *chi*, CORS [53], and render [54].

The *chi* module provides the tools for creating the router, middleware, and other essential features for implementing the web service. The CORS module provides an easy way for enabling the Cross-Origin Resource Sharing requests for our services to allow the requests from all domains. Finally, the render package helps manage HTTP request and response payloads, making it more accessible.

The author chose the go-chi compared to the other alternatives because of the following key factors [52]:

- Chi is lightweight,
- Chi is fast,
- Chi is 100 % compatible with standard net/http package.

6.1.2 MongoDB and Mongo Driver

The database used for the implementation is MongoDB [50]. It is a document-oriented database that perfectly suits the primary needs of the implemented prototype:

- It is schema-less, which implies the possibility to develop the application fast,
- It is horizontally scalable out-of-the-box thanks to sharding,
- MongoDB is highly available,
- The data model suits the document orientation.

To use it from services written in the Go programming language, the author used the official MongoDB Go Driver [55], which provides an easy way to call the MongoDB API for developers familiar with using the official mongo shell with no complicated abstractions.

6.1.3 JSON Web Tokens

Thanks to the stateless way of authentication and easy usage, the author decided to use the JWT tokens for authentication and authorization purposes. For implementing the generation of access and refresh tokens and the tokens verification, the author used the jwt-go [56] library, which provides the main needed functionality. The main features are [57]:

- JWT tokens generation,
- JWT tokens parsing and verification,
- Signing of JWT tokens,
- Possibility to add the own custom implementation of the signing algorithm.

6.1.4 Bcrypt

The author decided to use the Bcrypt adaptive hashing algorithm and its implementation from the Go Cryptography [58] library to store the passwords securely. The main reasons to choose this algorithm were:

- It incorporates a sault to protect against rainbow table attacks.
- It is an adaptive function that allows increasing the iterations count, making it resistant to brute-force search attacks but making the algorithm slower.
- The bcrypt function is the default password hash algorithm for OpenBSD. [59]

6.1.5 ReactJS

ReactJS [60] is an open source library for creating the frontend applications in the JavaScript programming language, which was developed and is maintained by Facebook and individual contributors. The key factors that influenced this technology's choice are:

- High popularity,
- Virtual DOM,
- Combining the JavaScript and HTML in JSX,
- Making the component easy to use and understand,
- The possibility of the server-side rendering.

6.1.6 Other Technologies

For implementing the User Interface in ReactJS, the author used an open source components library, PrimeReact [61], providing easy-to-use components with minimalistic and customizable themes, which allows to speed up the development by achieving more in less time.

For routing the URLs in a single-page application, the author used the React Router [62] library, which perfectly suits the React ecosystem and is one of the most popular solutions [63]. The author also uses it to redirect all unauthorized requests to the login page and disable the login page for authorized users.

For the end-to-end encryption, the author used a SubtleCrypto [64] interface of the Web Crypto API that provides several low-level cryptographic functions because all modern browsers support it.

The author used Docker [65] as the most popular container technology to create and deploy the backend services. It provides all features needed for the Instant Messaging Platform prototype implementation. In addition, it allows to speed up the development process by having multiple instances of databases running without any conflict on the developer's machine and easy management of services.

6.2 Containers

As mentioned in the previous section, the author used Docker containers to speed up the development, automate the creation and initialization of the database, and build and start the Go application. This section will describe the Dockerfiles and docker-compose configurations for backend services.

6.2.1 Users Service

Each service implemented for the Instant Messaging Platform, the output of this thesis, has its Dockerfile for building the images. Since the logic is defined on the application level, all three Dockerfiles are made using the same template, so the thesis describes only the Users Service Dockerfile. Chat Service and Delivery Service use the modified version, mostly defers in paths and names. However, the file skeleton stays the same.

```
1  # syntax=docker/dockerfile:1
2
3  FROM golang:1.18-alpine
4
5  ENV APP_PATH=/rchat-users
6  ENV SRC_PATH=/build/rchat/users/app
7
8  # Update packages
9  RUN apk update -q && \
10     apk add --upgrade apk-tools && \
11     apk upgrade --available
12
13 EXPOSE 8080
14
15 # Create the limited user
16 RUN addgroup -S rchat-users && \
17     adduser -S -G rchat-users rchat-users
18
19 WORKDIR $SRC_PATH
20
21 # Build the application
22 ADD . ./
23 RUN go mod download
24 RUN go build -o $APP_PATH
25
26 # Continue as the limited user
27 RUN chown rchat-users:rchat-users $APP_PATH
28 USER rchat-users
29
30 CMD ${APP_PATH}
```

Listing 6.1: Users Service Dockerfile

On listing 6.1, the reader can see the Dockerfile for Users Service. It is relatively simple. Thanks to the `FROM` instruction, the produced image becomes based on the official golang Alpine Linux image. It sets the environment variables that the file later uses for dealing with paths. Their purpose is to define the paths only once for the whole Dockerfile and reuse them, which reduces the number of mistakes possible in specifying the same paths multiple times. Also, it updates the Alpine packages on lines 9–11 to ensure that the image contains the latest security updates and the container is not outdated. Line 13 uses the `EXPOSE` instruction to indicate that the application running in the container is listening for requests on port 8080. The last line related to the

image configuration is creating the limited user on lines 16–17, which will be later responsible for running the application. It ensures that the application does not have root access to the system and reduces possible security risks.

The second part of the file is related to the Go application itself. Using the `WORKDIR` instruction on line 19, the file changes the directory where all commands will execute, adds all files from the service folder to that directory using the `ADD` instruction, and finally builds the application on lines 23–24. After that, it updates the owner of the produced binary to the limited user created in the first part of the file and, using the `USER` instruction, sets this user to execute the rest of the commands in this file. The last line is making the application run on container start.

Listing 6.2 shows the `docker-compose.yml` file used for creating and running the needed containers for Users Service. It defines the database "db" and the Go application "service" services.

```
1  version: "3.8"
2  services:
3    db:
4      image: mongo
5      container_name: rchat-usersdb
6      environment:
7        - PUID=1000
8        - PGID=1000
9      volumes:
10       - ./db/mongodb:/data/db
11       - ./db/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init
       .js:ro
12     restart: unless-stopped
13
14   service:
15     build:
16       dockerfile: Dockerfile
17       context: app/
18     container_name: rchat-users
19     environment:
20       - RCHAT_DB_URL=mongodb://rchat-usersdb:27017
21       - RCHAT_JWT_KEY=secret-key
22     ports:
23       - "8080:8080"
24     depends_on:
25       - db
26     restart: unless-stopped
```

Listing 6.2: Users Service docker-compose.yml

As mentioned previously, the database is MongoDB. The most important lines are 10–11, which define the volumes for the database container. The first one maps the `db/mongodb` directory on the host to the `/data/db` directory in the container to make the data stored in the database persistent even when the container restarts or is deleted. The second one is used for the database

initialization. It uses the `mongo-init.js` file to create indexes which will be discussed later.

Then, the `docker-compose.yml` defines the "service" service. It sets the build context, the directory containing the application files, and the `Dockerfile` in this directory to build the image for the service container. Furthermore, later on, line 23 maps the host port 8080 to container port 8080, which makes the application accessible from the host.

Thanks to the docker default networking, the file can provide the container name as the domain name without exposing the database port to the host. It does that on line 20 to provide the database URL to the application as the environment variable. Line 21 provides the secret for JWT tokens in plain text. Ideally, for production applications, plain text secrets should not be used since it is a security risk, but creating a shared secret store was out of the scope of this thesis.

6.2.2 Chat Service

Listing 6.3 shows the `docker-compose.yml` file used for creating and running the needed containers for Chat Service. The database service is configured the same way as for the Users Service. The only differences are in the content of the `mongo-init.js` script, which does not affect the `docker-compose.yml` file and the chat network. Adding the `chat` network is required to connect the Chat Service application with the database. It can not work using the default networking because the file defines the new network `chat-delivery-network` to connect to the Delivery Service. However, the only difference for database service is lines 13–14.

```
1  version: "3.8"
2  services:
3    db:
4      image: mongo
5      container_name: rchat-chatdb
6      environment:
7        - PUID=1000
8        - PGID=1000
9      volumes:
10       - ./db/mongodb:/data/db
11       - ./db/mongo-init.js:/docker-entrypoint-initdb.d/mongo-init
       .js:ro
12     restart: unless-stopped
13     networks:
14       - chat
15
16   service:
17     build:
18       dockerfile: Dockerfile
19       context: app/
20     container_name: rchat-chat
21     environment:
```

```
22      - RCHAT_DB_URL=mongodb://rchat-chatdb:27017
23      - RCHAT_JWT_KEY=secret-key
24      - RCHAT_DELIVERY_JWT_KEY=delivery-secret-key
25    ports:
26      - "8081:8080"
27    depends_on:
28      - db
29    restart: unless-stopped
30    networks:
31      - chat
32      - delivery
33
34 networks:
35   chat:
36     name: chat-network
37   delivery:
38     name: chat-delivery-network
```

Listing 6.3: Chat Service docker-compose.yml

The service is different in networks as well. It connects to the `chat` and `delivery` networks to connect to the database and the Delivery Service for real-time message delivery. The other differences are another port 8081 on the host mapped to the 8080 port in the container and the additional JWT key used for authenticating the delivery requests to Delivery Service to restrict access to the delivery endpoint only for the Chat Service.

6.2.3 Delivery Service

The delivery service is different because it does not store any data in the database, so the only defined service displayed on listing 6.4 is the Delivery Service application itself. It gets both JWT secrets through the environment variables as the Chat Service. It needs the general JWT key to allow only the authenticated users to access the connecting endpoint. In addition, it maps the next available host port 8082 to 8080 port in the container and connects to the same `chat-delivery-network` as the Chat Service.

```
1  version: "3.8"
2  services:
3    service:
4      build:
5        dockerfile: Dockerfile
6        context: app/
7      container_name: rchat-delivery
8      environment:
9        - RCHAT_JWT_KEY=secret-key
10       - RCHAT_DELIVERY_JWT_KEY=delivery-secret-key
11     ports:
12       - "8082:8080"
13     expose:
14       - 8080
15     restart: unless-stopped
```

```
16      networks:
17        - delivery
18
19  networks:
20    delivery:
21      name: chat-delivery-network
```

Listing 6.4: Delivery Service docker-compose.yml

6.3 Shared Library

All three backend services are web applications with JWT tokens authentication. All of them have some standard parts of code, such as error responses, logging, token generation, verification and middleware, and hashing, which the author decided to implement in the shared library for reuse. This section will describe some of them.

6.3.1 Logging

There was no need for some intelligent logging for the prototype implemented during this thesis. Hence, the author defined a `Logger` interface to make the logging algorithm easily changeable and the `ConsoleLogger` implementation for debugging purposes. Listing 6.5 displays both of them.

```
1  type Logger interface {
2      Info(message string)
3      Warning(message string)
4      Error(message string, err error)
5  }
6
7  type ConsoleLogger struct{}
8
9  func (l *ConsoleLogger) Info(message string) {
10     l.log("[INFO]", message)
11 }
12
13 func (l *ConsoleLogger) Warning(message string) {
14     l.log("[WARN]", message)
15 }
16
17 func (l *ConsoleLogger) Error(message string, err error) {
18     if err == nil {
19         l.errorMessage(message)
20         return
21     }
22     l.log("[ERROR]", fmt.Sprintf("%s\nError: '%s'", message, err.
       Error()))
23 }
24
25 func (l *ConsoleLogger) errorMessage(message string) {
```

```
26      l.log("[ERROR]", message)
27  }
28
29  func (l *ConsoleLogger) log(prefix string, message string) {
30      fmt.Printf("%s %s\n", prefix, message)
31  }
```

Listing 6.5: Logging implementation

6.3.2 Tokens Authentication

One of the goals of this thesis was to make the security mechanisms adjustable and easy to change. That is why the author defined the interface for tokens authentication that all services use and its implementation, which operates with JWT tokens.

Listing 6.6 shows the `TokensProvider` interface with only two functions — `GenerateTokens` and `Verify`, which usage is evident from their signatures. The current implementation generates access and refresh tokens and provides a way to verify the access token. In case of the successful verification, it returns the claims containing only the username needed during the implementation.

```
1  type Tokens struct {
2      AccessToken  string `json:"access_token"`
3      RefreshToken string `json:"refresh_token"`
4  }
5
6  type Claims struct {
7      Username string `json:"username"`
8  }
9
10 type TokensProvider interface {
11     GenerateTokens(username string) (*Tokens, error)
12     Verify(token string) (*Claims, error)
13 }
```

Listing 6.6: TokensProvider abstraction

Listing 6.7 shows the structure `JwtTokensProvider` and its creation using the `NewJwtTokensProvider` function.

```
1  type jwtClaims struct {
2      Username string `json:"username"`
3      jwt.StandardClaims
4  }
5
6  type JwtTokensProvider struct {
7      secret         string
8      accessTokenTTL time.Duration
9  }
10
11 func NewJwtTokensProvider(secret string, accessTokenTTL time.
       Duration) (*JwtTokensProvider, error) {
```

```
12      if len(secret) == 0 {
13          return nil, fmt.Errorf("Empty string is not allowed as a
    secret for JWT tokens")
14      }
15      return &JwtTokensProvider{
16          secret:         secret,
17          accessTokenTTL: accessTokenTTL,
18      }, nil
19  }
```

Listing 6.7: JwtTokensProvider structure

The `GenerateTokens` function displayed on the listing 6.8 creates the access token with the provided time to live and the refresh token, valid for seven days using the jwt-go library and HS256 signing method.

```
1   func (p *JwtTokensProvider) GenerateTokens(username string) (*
    Tokens, error) {
2       accessClaims := &jwtClaims{
3           Username: username,
4           StandardClaims: jwt.StandardClaims{
5               ExpiresAt: time.Now().UTC().Add(p.accessTokenTTL).
    Unix(),
6           },
7       }
8       refreshClaims := &jwtClaims{
9           StandardClaims: jwt.StandardClaims{
10              ExpiresAt: time.Now().UTC().Add(7 * 24 * time.Hour).
    Unix(),
11          },
12      }
13
14      accessToken, err := jwt.NewWithClaims(jwt.SigningMethodHS256,
        accessClaims).SignedString([]byte(p.secret))
15      if err != nil {
16          return nil, fmt.Errorf("Error generating the access token
    : %w", err)
17      }
18      refreshToken, err := jwt.NewWithClaims(jwt.SigningMethodHS256
    , refreshClaims).SignedString([]byte(p.secret))
19      if err != nil {
20          return nil, fmt.Errorf("Error generating the refresh
    token: %w", err)
21      }
22
23      return &Tokens{AccessToken: accessToken, RefreshToken:
    refreshToken}, nil
24  }
```

Listing 6.8: JwtTokensProvider's GenerateTokens function

The `Verify` function of the `JwtTokensProvider` structure on listing 6.9 verifies the provided JWT access token and returns the `Claims` coming from the interface definition.

```
func (p *JwtTokensProvider) Verify(token string) (*Claims, error) {
    claims := &jwtClaims{}
    tkn, err := jwt.ParseWithClaims(token, claims, func(token *jwt.Token) (interface{}, error) {
        return []byte(p.secret), nil
    })
    if err != nil {
        return nil, err
    }
    if !tkn.Valid {
        return nil, fmt.Errorf("JWT token is invalid")
    }
    return &Claims{Username: claims.Username}, nil
}
```

Listing 6.9: JwtTokensProvider's Verify function

The Shared Library also defines the middleware used by the services to validate the token and add the username to the request context. Its implementation is visible on the listing 6.10, and it supports providing the token both in the `Authorization` header using the ``bearer {token}'' string and in the query parameter ``?token={token}''. The implementation of `getHeaderToken` and `getQueryToken` functions is not part of the listing but is part of the attached source files.

```
type Middleware = func(http.Handler) http.Handler

func ValidateTokenAndAddUsernameToContext(provider providers.TokensProvider, logger logger.Logger) Middleware {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            token, err := getHeaderToken(r)
            if err != nil {
                token, err = getQueryToken(r)
                if err != nil {
                    logger.Error("Can't get token from header/query", err)
                    render.Render(w, r, e.Unauthorized)
                    return
                }
            }
            claims, err := provider.Verify(token)
            if err != nil {
                logger.Error("Invalid token", err)
                render.Render(w, r, e.Unauthorized)
                return
            }
            ctx := context.WithValue(r.Context(), UsernameContextKey, claims.Username)
            next.ServeHTTP(w, r.WithContext(ctx))
        })
    }
```

```
25 }
```

Listing 6.10: Tokens middleware

6.3.3 Passwords Hashing

Thanks to the interface, the password hashing logic can easily change, like the tokens authentication. The library defines the simple `HashingProvider` interface visible on listing 6.11 with only two methods, `Hash` and `Verify`. In the case of the bcrypt algorithm, it is not enough to hash the received password and compare the hashes, so the abstraction should also support this scenario.

```
1 type HashingProvider interface {
2     Hash(value string) (string, error)
3     Verify(expected string, provided string) error
4 }
```

Listing 6.11: HashingProvider interface

The bcrypt implementation of the above interface is short because it uses the bcrypt implementation from the Go Cryptography library. The listing 6.12 shows the `BcryptHashingProvider` implementation.

```
1  type BcryptHashingProvider struct {
2      cost int
3  }
4
5  func NewBcryptHashingProvider(cost int) *BcryptHashingProvider {
6      return &BcryptHashingProvider{cost: cost}
7  }
8
9  func (p *BcryptHashingProvider) Hash(value string) (string, error
       ) {
10     bytes, err := bcrypt.GenerateFromPassword([]byte(value), p.
       cost)
11     if err != nil {
12         return "", err
13     }
14     return string(bytes), nil
15 }
16
17 func (p *BcryptHashingProvider) Verify(expected string, provided
       string) error {
18     return bcrypt.CompareHashAndPassword([]byte(expected), []byte
       (provided))
19 }
```

Listing 6.12: BcryptHashingProvider implementation

6.3.4 Referencing the Shared Library

Since the Shared Library is private, the Go module cannot easily add the dependency using the GitHub URL. That is why each service directory contains the `copy-lib.sh` script that copies the library sources to the `build/lib/rchat-lib/` directory. The listing 6.13 displays the reference to it in the go.mod file.

```
1 ...
2 require rchat/lib v1.0.0
3
4 replace rchat/lib => ./build/lib/rchat-lib
5 ...
```

Listing 6.13: Shared Library reference in go.mod file

6.4 Users Service

This section will describe the Users Service implementation details. It will cover the most critical topics and provide relevant code listings.

6.4.1 Database

As mentioned in the Containers 6.2 section, the author created the script for initializing the database. The listing 6.14 displays the content of this script. There is nothing special in its lines. The only important line is line number 5, which creates the unique index for documents in `users` collection. Thanks to it, all queries based on the username will work fast, and we ensure that there are no users with the same usernames on the database level.

```
1 db = db.getSiblingDB('rchat-users')
2
3 let res = [
4   db.users.drop(),
5   db.users.createIndex({ username: 1 }, { unique: true })
6 ]
7
8 printjson(res)
```

Listing 6.14: Users Service mongo-init.js

The database state is updated from the Go application directly. This service does not store much data because it handles only the logic closely connected to the user. Listing 6.15 shows the `User` structure, which the service application uses to manage documents in `users` collection — all the properties are strings: the database stores only the username and public key used for end-to-end encryption. MongoDB generates the id when inserting the document.

```
1 type User struct {
2     Id        string `bson:"_id,omitempty" json:"id,omitempty"`
3     Username  string `bson:"username" json:"username"`
```

```
4     PublicKey string `bson:"public_key" json:"public_key"`
5 }
```

Listing 6.15: Users Service User structure

6.4.2 Starting the Web Server

Before creating the chi router, we need to create the dependencies and inject them into the request handler. Listing 6.16 shows how the app gets environment variables defined previously in `docker-compose.yml`, uses its values to instantiate the JWT implementation of `TokensProvider` interface, creates the MongoDB client, and uses them to instantiate the `UsersMongoService` later used as the `UsersHandler` dependency together with the `ConsoleLogger`.

```
1 mongoUrl := os.Getenv("RCHAT_DB_URL")
2 jwtSecretKey := os.Getenv("RCHAT_JWT_KEY")
3
4 jwtTokensProvider, err := providers.NewJwtTokensProvider(
    jwtSecretKey, 24*time.Hour)
5 if err != nil {
6   log.Fatal(err)
7 }
8
9 mongoClient, err := db.CreateMongoClient(mongoUrl)
10 if err != nil {
11   log.Fatal(err)
12 }
13
14 usersService := service.NewUsersMongoService(
15   providers.NewBcryptHashingProvider(14),
16   jwtTokensProvider,
17   mongoClient,
18 )
19
20 consoleLogger := &logger.ConsoleLogger{}
21 usersHandler := &handlers.UsersHandler{UsersService: usersService
    , Logger: consoleLogger}
```

Listing 6.16: Creating the UsersHandler

With all dependencies initialized, we can move to creating and configuring the chi router. Listing 6.17 shows this process. The application creates the chi router, sets the default logging middleware for incoming requests that prints the output to the console for debugging purposes, then sets the `ErrorHandlers` defined in the Shared Library to return the valid JSON `MethodNotAllowed` and `NotFound` error responses.

Lines 7–14 show the CORS requests' settings to allow the Web Applications from different domains to call this service.

Then lines 16–19 define the public routing for login and register requests to the `UsersHandler` functions. Next, on lines 20–23, there is the private endpoint for getting the user's public key for end-to-end encryption purposes.

It uses the `ValidateTokenAndAddUsernameToContext` middleware defined in the Shared Library.

Finally, the last lines make the application start and listen for requests on port 8080.

```
r := chi.NewRouter()
r.Use(middleware.Logger)

r.MethodNotAllowed(errors.ErrorHandler(errors.MethodNotAllowed))
r.NotFound(errors.ErrorHandler(errors.NotFound))

r.Use(cors.Handler(cors.Options{
  AllowedOrigins:   []string{"https://*", "http://*"},
  AllowedMethods:   []string{"GET", "POST", "PUT", "DELETE", "
    OPTIONS"},
  AllowedHeaders:   []string{"Accept", "Authorization", "Content-
    Type", "X-CSRF-Token"},
  ExposedHeaders:   []string{"Link"},
  AllowCredentials: false,
  MaxAge:           300,
}))

r.Route("/auth", func(r chi.Router) {
  r.Post("/login", usersHandler.Login)
  r.Post("/register", usersHandler.Register)
})
r.Route("/users", func(r chi.Router) {
  r.Use(middlewares.ValidateTokenAndAddUsernameToContext(
    jwtTokensProvider, consoleLogger))
  r.Get(fmt.Sprintf("/{%s}/public-key", constants.
    UsernamePathParam), usersHandler.GetPublicKey)
})

if err := http.ListenAndServe(":8080", r); err != nil {
  log.Fatal("Can't start the server.", err)
}
```

Listing 6.17: Creating and starting the Users Service router

6.4.3 Handling the Requests

This subsection will show the example of the request handling flow and will cover only the registration request. The other logic is a part of the thesis attachments.

Listing 6.18 shows the implementation of the `UsersHandler Register` function. It instantiates the `RegisterRequest` from the service package that stores the username, password, and public key using the request JSON body, calls the `Exists` function of the `UsersService` interface, and if the user does not exist — calls the `Register` function of the same `UsersService`. Besides that, it also handles all errors and returns valid JSON error responses. The response is only the successful status code and empty JSON body.

```
type UsersHandler struct {
    UsersService service.UsersService
    Logger       logger.Logger
}

func (h *UsersHandler) Register(w http.ResponseWriter, r *http.
    Request) {
    var registerRequest = &service.RegisterRequest{}
    if err := render.Bind(r, registerRequest); err != nil {
        render.Render(w, r, e.BadRequest(err))
        return
    }

    exists, err := h.UsersService.Exists(registerRequest.Username
    )
    if err != nil {
        h.Logger.Error("Error validating the user existence
    before registration", err)
        render.Render(w, r, e.InternalServerError)
        return
    }
    if exists {
        render.Render(w, r, e.BadRequest(fmt.Errorf("user '%s'
    already exists", registerRequest.Username)))
        return
    }

    if err := h.UsersService.Register(registerRequest); err !=
    nil {
        h.Logger.Error("Error registering the user", err)
        render.Render(w, r, e.InternalServerError)
        return
    }
    render.Status(r, http.StatusOK)
    render.PlainText(w, r, "{}")
}
```

Listing 6.18: UsersHandler Register function

Listing 6.19 shows the `UsersMongoService` `Register` function implementation. Thanks to the `UsersService` interface, we can have multiple service implementations for different data sources. The `Register` function first uses the `hashProvider` to compute the hash for the provided password, then changes the plain text password value with the hash in the provided request object and inserts the document into the users collection.

```
type UsersMongoService struct {
    collection     *mongo.Collection
    hashProvider   providers.HashingProvider
    tokensProvider providers.TokensProvider
    timeout        time.Duration
}

func NewUsersMongoService(
```

```
9     hashProvider providers.HashingProvider,
10    tokensProvider providers.TokensProvider,
11    mongoClient *mongo.Client,
12) UsersService {
13    return &UsersMongoService{
14        collection:     mongoClient.Database(db.DatabaseName).
      Collection(db.UsersCollection),
15        hashProvider:   hashProvider,
16        tokensProvider: tokensProvider,
17        timeout:        10 * time.Second,
18    }
19}
20
21func (s *UsersMongoService) Register(registerRequest *
      RegisterRequest) error {
22    hash, err := s.hashProvider.Hash(registerRequest.Password)
23    if err != nil {
24        return fmt.Errorf("Error generating the password hash for
       registration: %w", err)
25    }
26    registerRequest.Password = hash
27
28    var ctx, cancel = context.WithTimeout(context.Background(), s
      .timeout)
29    defer cancel()
30
31    _, err = s.collection.InsertOne(ctx, registerRequest)
32    return err
33}
```

Listing 6.19: UsersMongoService Register function

6.5 Chat Service

6.5.1 Database

The Chat Service database has its script for initializing the database as the Users Service has. Listing 6.20 shows the content of the `mongo-init.js` script. It creates the indexes for all three collections:

- The `users` collection has a unique index for the `username` field.
- The `privateConversations` collection has the unique index for the field `participants_hash`, used for the faster queries based on the participants list.
- The `messages` collection has the index for the `conversation_id` field, used for all queries that load messages in the application.

```
db = db.getSiblingDB('rchat-chat')

let res = [
    db.users.drop(),
    db.users.createIndex({ username: 1 }, { unique: true }),
    db.privateConversations.drop(),
    db.privateConversations.createIndex({ participants_hash: 1 },
     { unique: true }),
    db.messages.drop(),
    db.messages.createIndex({ conversation_id: 1 })
]

printjson(res)
```

Listing 6.20: Chat Service mongo-init.js

The model of the Chat Service is more complicated than the Users Service has. It currently supports only direct messages, but it is designed to support extensibility, group chats, chatbots, channels, and other possible extensions. The first entity shown on listing 6.21 is the `User` entity. It stores the username and the list of conversations the user participates in. The conversation, in this case, stores only the id of the conversation from the `privateConversations` collection and the type, which is the enum currently having only one possible value — `PrivateConversationType`, which represents the conversation between two participants.

```
type User struct {
    Id            string             `bson:"_id,omitempty" json:"
    id,omitempty"`
    Username      string             `bson:"username" json:"
    username"`
    Conversations []UserConversation `bson:"conversations" json:"
    conversations"`
}

type UserConversation struct {
    Id   string           `bson:"_id,omitempty" json:"id,
    omitempty"`
    Type ConversationType `bson:"type" json:"type"`
}

type ConversationType int

const (
    PrivateConversationType ConversationType = iota
)
```

Listing 6.21: Chat Service User entity

The following entity represents the conversation. The `Conversation` struct shown on listing 6.22 is standard for all possible conversation types. The `PrivateConversation` extends this `Conversation` for direct messages and

contains the `ParticipantsHash` field for making the queries faster when initializing the new conversation.

```
type Conversation struct {
    Id           string   `bson:"_id,omitempty" json:"id,
    omitempty"`
    Participants []string `bson:"participants" json:"participants
    "`
}

type PrivateConversation struct {
    Conversation     `bson:",inline"`
    ParticipantsHash string `bson:"participants_hash" json:"-"`
}
```

Listing 6.22: Chat Service Private Conversation entity

The final entity represents the message. It is common for direct and group messages and may even be used for channels. However, it is vital to use this structure based on the collection that stores it and provides the mapping to the corresponding conversations collection. The general message shown on listing 6.23 defines the standard fields for all possible message types. Furthermore, the `Message` structure extends it with the `ConversationId` field applicable for direct conversations and many possible future extensions.

```
type GeneralMessage struct {
    Id      string `bson:"_id,omitempty" json:"id,omitempty"`
    From    string `bson:"from" json:"from,omitempty"`
    Content []byte `bson:"content" json:"content"`
}

type Message struct {
    GeneralMessage `bson:",inline"`
    ConversationId string `bson:"conversation_id" json:"
    conversation_id,omitempty"`
}
```

Listing 6.23: Chat Service Private Message entity

6.5.2 Starting the Web Server

In many ways, configuring and starting the Web Server for Chat Service is the same as for Users Service.

Firstly, the chat service has more handlers and uses the Delivery Service `/deliver` endpoint, which requires another secret for JWT token signature. So the application needs to create the `DeliveryService`, pass it to the messages services for real-time messages delivery, and instantiate all the handlers to handle the requests coming from the client. Listing 6.24 displays the whole straightforward process. All mongo services implement the interfaces, so the implementation can be easily changed to support another store.

```
mongoUrl := os.Getenv("RCHAT_DB_URL")
jwtSecretKey := os.Getenv("RCHAT_JWT_KEY")
deliverySecretKey := os.Getenv("RCHAT_DELIVERY_JWT_KEY")

// -1 ttl because chat service does not issue user access tokens, only validates them
jwtTokensProvider, err := providers.NewJwtTokensProvider(jwtSecretKey, -1)
if err != nil {
  log.Fatal(err)
}

consoleLogger := &logger.ConsoleLogger{}

deliveryService, err := services.NewDeliveryService("http://rchat-delivery:8080/deliver", deliverySecretKey, consoleLogger)
if err != nil {
  log.Fatal(err)
}

mongoClient, err := db.CreateMongoClient(mongoUrl)
if err != nil {
  log.Fatal(err)
}

privateMessagesService := service.NewPrivateMessagesMongoService(mongoClient, deliveryService)
privateMessagesHandler := handlers.PrivateMessagesHandler{Service: privateMessagesService, Logger: consoleLogger}

messagesService := service.NewMessagesMongoService(mongoClient, deliveryService, 15)
messagesHandler := handlers.MessagesHandler{Service: messagesService, Logger: consoleLogger}

userService := service.NewUserMongoService(mongoClient)
userHandler := handlers.UserHandler{Service: userService, Logger: consoleLogger}

privateConversationsService := service.NewPrivateConversationsMongoService(mongoClient)
privateConversationsHandler := handlers.PrivateConversationsHandler{Service: privateConversationsService, Logger: consoleLogger}
```

Listing 6.24: Creating the Chat Service handlers

Listing 6.25 shows all the endpoints the Chat Service defines to satisfy the Instant Messaging Platform requirements defined previously. The Chat Service does not have any public endpoints and secures them with the same authentication middleware defined in the Shared Library using the Users Service secret key.

```
r.Group(func(r chi.Router) {
```

```
    r.Use(middlewares.ValidateTokenAndAddUsernameToContext(jwtTokensProvider, consoleLogger))
    r.Route("/users", func(r chi.Router) {
        r.Get("/", userHandler.FindUsers)
        r.Post("/init", userHandler.InitialCall)
        r.Get("/conversations", userHandler.GetConversations)
    })
    r.Route("/messages", func(r chi.Router) {
        r.Get(fmt.Sprintf("/conversation/{%s}", constants.IdPathParam), messagesHandler.FindForConversation)
        r.Route("/private", func(r chi.Router) {
            r.Post("/", privateMessagesHandler.Send)
            r.Post("/init", privateMessagesHandler.SendInitial)
        })
    })
    r.Route("/conversations", func(r chi.Router) {
        r.Route("/private", func(r chi.Router) {
            r.Route(fmt.Sprintf("/{%s}", constants.IdPathParam), func(r chi.Router) {
                r.Get("/", privateConversationsHandler.GetConversation)
                r.Get("/participants", privateConversationsHandler.GetParticipants)
            })
        })
    })
})
```

Listing 6.25: Chat Service routes

6.5.3 Sending the Initial Message

For the request flow demonstration purposes, the author chose the request for sending the initial message as the most complicated one. In addition, it uses the delivery service to send a real-time message to the recipient.

The `PrivateMessagesHandler` `SendInitial` function shown on listing 6.26 has the same structure as the handlers function in Users Service. It parses the message body, calls the `PrivateMessagesService` `SendInitial` function, and handles the errors. It returns the message with the id and conversation id to the client.

```
func (h *PrivateMessagesHandler) SendInitial(w http.ResponseWriter, r *http.Request) {
    message := &service.InitialPrivateMessage{}
    if err := render.Bind(r, message); err != nil {
        switch err.(type) {
        case *chatErrors.InvalidMessageError:
            render.Render(w, r, e.BadRequest(err))
            return
        default:
```

```
9               h.Logger.Error("Error creating the
    InitialPrivateMessage before sending the initial private
    message", err)
10              render.Render(w, r, e.InternalServerError)
11              return
12          }
13      }
14      msg, err := h.Service.SendInitial(message)
15      if err != nil {
16          switch err.(type) {
17          case *chatErrors.InvalidMessageError:
18              render.Render(w, r, e.BadRequest(err))
19              return
20          default:
21              h.Logger.Error("Error sending the initial private
    message", err)
22              render.Render(w, r, e.InternalServerError)
23              return
24          }
25      }
26      render.JSON(w, r, msg)
27  }
```

Listing 6.26: Chat Service send initial message handler

Listing 6.27 shows the `PrivateMessagesMongoService` `SendInitial` function implementation. In the beginning, it validates the user's existence. After that, it calls the `getPrivateConversationId` function, which will check if the conversation with the recipient exists, and, if not — creates it and returns the id. After that, it inserts the document into the database. In case of a successful insert, it will deliver the message in real-time using the `DeliveryService`, which calls the Delivery Service application. All private helping functions are not parts of this listing and are parts of the attached source files.

```
1  func (s *PrivateMessagesMongoService) SendInitial(initMessage *
       InitialPrivateMessage) (*store.Message, error) {
2      ctx, cancel := context.WithTimeout(context.Background(), s.
       timeout)
3      defer cancel()
4
5      for _, username := range []string{initMessage.From,
       initMessage.To} {
6          if err := s.validateUserExists(ctx, username); err != nil
        {
7              return nil, err
8          }
9      }
10
11     conversationId, err := s.getPrivateConversationId(ctx,
       initMessage)
12     if err != nil {
13         return nil, err
14     }
```

```
15      message := &store.Message{
16          GeneralMessage: store.GeneralMessage{
17              From:    initMessage.From,
18              Content: initMessage.Content,
19          },
20          ConversationId: conversationId,
21      }
22      result, err := s.messagesCollection.InsertOne(ctx, message)
23      if err != nil {
24          return nil, err
25      }
26      message.Id = result.InsertedID.(primitive.ObjectID).Hex()
27
28      s.deliveryService.DeliverInitialMessage(&services.
        DeliveryInitialMessage{
29          Message:   *message,
30          SharedKey: initMessage.SharedKey,
31      }, []string{initMessage.From, initMessage.To})
32      return message, nil
33  }
```

Listing 6.27: Chat Service PrivateMessagesMongoService SendInitial function

Listing 6.28 shows the Delivery Service client function responsible for delivering the initial message. As the first step, it serializes the message with the shared encryption key to JSON because the Delivery Service does not check the content of the message and delivers the message. After that, it prepares the request body with the recipients list in deliver function, creates the one-time JWT token, and sends the POST request to Delivery Service.

```
1   type DeliveryRequest struct {
2       Recipients []string `json:"recipients"`
3       Message    string   `json:"message"`
4   }
5
6   type DeliveryInitialMessage struct {
7       store.Message
8       SharedKey []byte `json:"shared_key"`
9   }
10
11  func (s *DeliveryService) DeliverInitialMessage(message *
        DeliveryInitialMessage, participants []string) {
12      messageJson, err := json.Marshal(message)
13      if err != nil {
14          s.logger.Error("Can't convert the message to JSON", err)
15      }
16      s.deliver(messageJson, message.From, participants)
17  }
18
19  func (s *DeliveryService) deliver(messageJson []byte, from string
        , participants []string) {
20      body := &DeliveryRequest{
21          Recipients: getRecipients(from, participants),
22          Message:    string(messageJson),
```

```
23      }
24      bodyJson, err := json.Marshal(body)
25      if err != nil {
26          s.logger.Error("Can't convert the message to JSON", err)
27      }
28
29      tokens, err := s.tokensProvider.GenerateTokens(from)
30      if err != nil {
31          s.logger.Error("Can't generate the access token for
        delivery service", err)
32          return
33      }
34      request, err := http.NewRequest("POST", s.uri, bytes.
        NewBuffer(bodyJson))
35      if err != nil {
36          s.logger.Error("Can't create the HTTP POST request to
        delivery service", err)
37          return
38      }
39      request.Header.Add("Authorization", fmt.Sprintf("bearer %s",
        tokens.AccessToken))
40      _, err = s.httpClient.Do(request)
41      if err != nil {
42          s.logger.Error("Error sending the delivery request", err)
43          return
44      }
45  }
```

Listing 6.28: Chat Service DeliveryService logic

6.6 Delivery Service

Since the Delivery Service is responsible only for delivering the messages, its configuration does not differ much from the other services.

Listing 6.29 shows the Delivery Service routes. There are two of them, and the different JWT signing secrets protect each.

```
1  ...
2
3  r.Group(func(r chi.Router) {
4    r.Use(middlewares.ValidateTokenAndAddUsernameToContext(
       deliveryTokensProvider, consoleLogger))
5    r.Post("/deliver", deliveryHandler.Deliver)
6  })
7  r.Group(func(r chi.Router) {
8    r.Use(middlewares.ValidateTokenAndAddUsernameToContext(
       jwtTokensProvider, consoleLogger))
9    r.Get("/connect", sseHandler.Connect)
10 })
11
12 ...
```

Listing 6.29: Delivery Service Routes

The delivery handler does not differ from the other handlers mentioned in this thesis and acts the same way and calls the `UserDevices` service's `NewMessage` function.

To deliver real-time messages, the Delivery Service uses the Server-Sent Events. That is why the `SSEHandler` is different from other ones, and listing 6.30 shows its implementation. It gets the username from the context and uses the `RemoteAddr` request property, which includes the port and provides the uniqueness of the device identifier, sets the needed request headers, and calls the `UserDevices Connect` function, which returns the channel where the delivery handler adds new messages. Then it defers the `Disconnect` function calls and, until the connection is closed, delivers all real-time messages that the Delivery Service receives.

```
type SSEHandler struct {
    UserDevices *services.UserDevices
    Logger      logger.Logger
}

func (h *SSEHandler) Connect(w http.ResponseWriter, r *http.Request) {
    h.Logger.Info("Get handshake from client")
    username := r.Context().Value(middlewares.UsernameContextKey).(string)
    device := r.RemoteAddr

    w.Header().Set("Content-Type", "text/event-stream")
    w.Header().Set("Cache-Control", "no-cache")
    w.Header().Set("Connection", "keep-alive")
    w.Header().Set("Access-Control-Allow-Origin", "*")

    messagesChannel := h.UserDevices.Connect(username, device)
    defer func() {
        if err := h.UserDevices.Disconnect(username, device); err != nil {
            h.Logger.Error("Error during disconnecting", err)
            return
        }
        h.Logger.Info("Client connection is closed")
    }()

    flusher, _ := w.(http.Flusher)
    for {
        select {
        case message := <-messagesChannel:
            _, err := fmt.Fprintf(w, "data: %s\n\n", message)
            if err != nil {
                h.Logger.Error("Error when delivering the message", err)
            }
            flusher.Flush()
        case <-r.Context().Done():
            return
```

```
36          }
37      }
38  }
```

Listing 6.30: Delivery Service SSEHandler implementation

Listing 6.31 displays the `UserDevices` service implementation. It stores the mapping of the user's device's messages channels. When the handler calls the `NewMessage` function, it adds the message to all recipients' devices channels so that every user will receive the message on all connected devices. The connect function adds the device to the user's devices. Finally, the `Disconnect` function does the required cleanup.

```
1  type MessagesChannel = chan string
2  type DevicesChannels = map[string]MessagesChannel
3
4  type UserDevices struct {
5      devices map[string]DevicesChannels
6  }
7
8  func NewUserDevices() *UserDevices {
9      devices := make(map[string]DevicesChannels)
10     return &UserDevices{devices: devices}
11 }
12
13 func (ud *UserDevices) NewMessage(recipients []string, message
       string) error {
14     for _, recipient := range recipients {
15         devicesChannels := ud.devices[recipient]
16         if devicesChannels == nil {
17             return errors.NoUserRegistrationError
18         }
19         for _, messages := range devicesChannels {
20             messages <- message
21         }
22     }
23     return nil
24 }
25
26 func (ud *UserDevices) Connect(username string, device string)
       MessagesChannel {
27     if ud.devices[username] == nil {
28         ud.devices[username] = make(DevicesChannels)
29     }
30     ch := make(MessagesChannel)
31     ud.devices[username][device] = ch
32     return ch
33 }
34
35 func (ud *UserDevices) Disconnect(username string, device string)
        error {
36     devicesChannels := ud.devices[username]
37     if devicesChannels == nil {
38         {
```

```
39              return fmt.Errorf("there are no user devices")
40          }
41      }
42      close(devicesChannels[device])
43      delete(devicesChannels, device)
44      return nil
45  }
```

Listing 6.31: Delivery Service UserDevices service

6.7 Testing

The author created the unit tests to test the more complex parts of the back-end services — that, besides the delegation of the function calls, have some complex logic.

Listing 6.32 displays the example of the test written for the `JwtTokensProvider`. It creates the tokens provider with the testing secret and 24 hours valid access token, then generates tokens, parses them, and validates their claims.

```
1  func TestJwtTokensProvider_Success(t *testing.T) {
2      assert := assert.New(t)
3      provider, err := NewJwtTokensProvider(secret, 24*time.Hour)
4      assert.NoError(err)
5
6      tokens, err := provider.GenerateTokens(username)
7      assert.NoError(err)
8
9      access, err := parseToken(tokens.AccessToken)
10     assert.NoError(err)
11     refresh, err := parseToken(tokens.RefreshToken)
12     assert.NoError(err)
13
14     accessClaims, ok := access.Claims.(*jwtClaims)
15     assert.True(ok)
16     assert.Less(time.Now().UTC().Unix(), accessClaims.ExpiresAt)
17     assert.Equal(username, accessClaims.Username)
18
19     refreshClaims, ok := refresh.Claims.(*jwtClaims)
20     assert.True(ok)
21     assert.Less(time.Now().UTC().Unix(), refreshClaims.ExpiresAt)
22     assert.Empty(refreshClaims.Username)
23 }
24
25 func parseToken(token string) (*jwt.Token, error) {
26     return jwt.ParseWithClaims(
27         token,
28         &jwtClaims{},
29         func(token *jwt.Token) (interface{}, error) {
30             return []byte(secret), nil
31         },
32     )
```

```
33 }
```

Listing 6.32: JwtTokensProvider Unit Test

The author tested the API functionality using the Postman [8] tool. Figure 6.1 displays the example of calling the `/auth/login` endpoint of Users Service. It sends the valid credentials, and we can use the result to compose the Authorization header and call other endpoints the same way. If the credentials are invalid, the API will return the JSON describing the error.

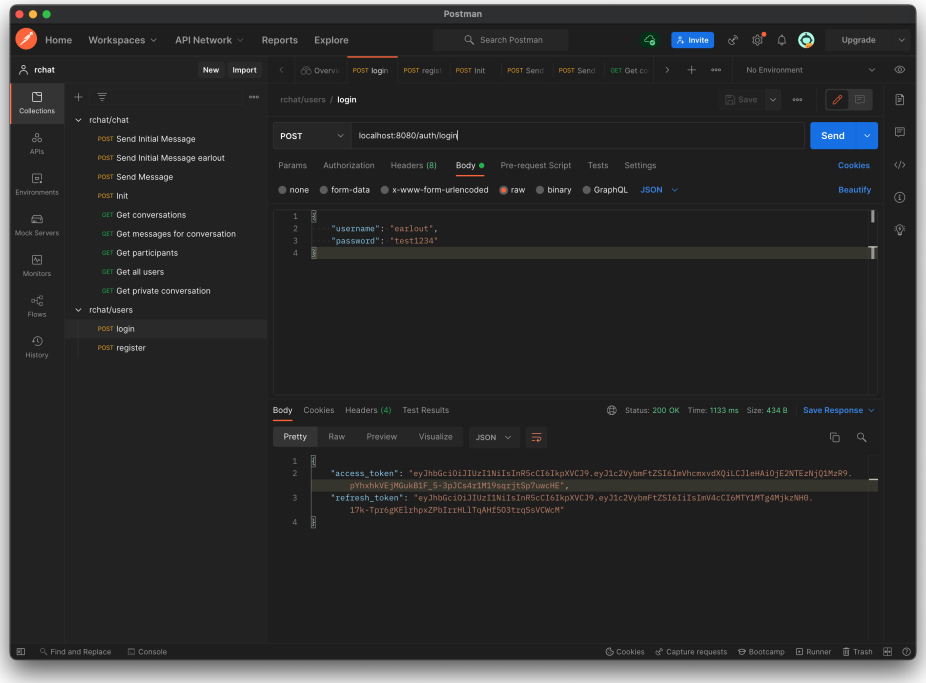

Figure 6.1: Postman [8]

6.8 Web Application Prototype

This section describes the Instant Messaging Platform frontend prototype. It does not cover the implementation of each User Interface component. However, instead of that, it covers the essential logic pieces such as pages routing, requests sending, end-to-end encryption, and the User Interface description.

6.8.1 Router

Since almost all application features require the user to be authenticated, the application should redirect the users to the login screen if they are not logged in already. Listing 6.33 shows the implementation of the logic which makes it possible. It defines two routes — the first one for the user, which lacks authentication, and the second one for the authenticated user. The application is a Single Page, so there are only two possible paths.

```
function App() {
  const [authenticated, setAuthenticated] = useState(localStorage
    .getItem(AUTH_TOKEN) !== null);
  return (
    <div style={{backgroundColor: '#badcef', height: '100%',
    width: '100%', position: 'fixed'}}>
      <BrowserRouter >
        { !authenticated && (
          <Routes>
            <Route path="/login" element={<Login setAuthenticated
    ={setAuthenticated} />} />
            <Route path="*" element={<Navigate to="/login" />} />
          </Routes>
        )}
        { authenticated && (
          <Routes>
            <Route path="/" element={<Main onLogout={() => {
    setAuthenticated(false)}} />} />
            <Route path="*" element={<Navigate to="/" />} />
          </Routes>
        )}
      </BrowserRouter>
    </div>
  );
}
```

Listing 6.33: Application navigation

6.8.2 Requests Sending

Listing 6.34 shows the example of the `POST` request sending logic. It stringifies the body, sets the headers, and sends the request. For both success and failure, it returns the valid promise, which utilizes the resolve and reject callbacks of the Promise. All `POST` API calls use this function and provide only the needed values as arguments.

```
const post = async (uri, headers, body) => {
  return new Promise((res, rej) => {
    fetch(uri, {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
```

```
        ...headers
      },
      body: JSON.stringify(body)
    }).then(async response => {
      if (!response.ok) {
        rej({
          error: (await response.json()).message
        });
        return
      }
      res({
        data: await response.json()
      });
    }).catch(e => {
      rej({
        error: e
      });
    });
  });
};
```

Listing 6.34: Function for POST requests sending

6.8.3 End-to-End Encryption

The author implemented the end-to-end encryption the following way. When the user creates the account, the application generates the RSA key pair, stores the private key locally, and sends the public key together with the registration request. Then, for each new conversation, the initiator generates the AES shared key, encrypts the message content with it, then encrypts the shared key with the recipient's public key coming from the Users Service, and sends it together with the encrypted message. Thanks to the JWT authentication, the Chat Service sets the `from` field based on the JWT token claim, ensuring that the sender sent the message and the key. Then the recipient receives the message, decrypts the shared key using the stored private key, and decrypts the message content using the decrypted shared key.

The author created the `crypto.js` file in the `src/crypto` directory to make this flow work, which defines all the needed functions and uses the Subtle Crypto interface previously mentioned in the Technologies section.

Listing 6.35 displays the functions related to the shared key logic. Thanks to this implementation, the algorithms can easily change, satisfying one of the thesis goals.

```
const generateSharedKey = async () => {
  return await crypto.subtle.generateKey(
    { name: "AES-CBC", length: 128 },
    true,
    ["encrypt", "decrypt"],
  );
};
```

```
8
9  const exportSharedKey = async (sharedKey) => {
10   return JSON.stringify(await crypto.subtle.exportKey("jwk",
       sharedKey));
11 };
12
13 const encryptSharedKey = async (sharedKey, recipientPublicKey) =>
       {
14   return Array.from(new Uint8Array(
15     await crypto.subtle.encrypt("RSA-OAEP", recipientPublicKey,
       sharedKey)
16   ));
17 };
18
19 const decryptSharedKey = async (encryptedSharedKey,
       recipientPrivateKey) => {
20   return new Uint8Array(await crypto.subtle.decrypt("RSA-OAEP",
       recipientPrivateKey, encryptedSharedKey));
21 };
22
23 const importSharedKey = async (jwk) => {
24   return await crypto.subtle.importKey(
25     "jwk",
26     jwk,
27     "AES-CBC",
28     true,
29     ["encrypt", "decrypt"],
30   );
31 };
```

Listing 6.35: Shared key logic functions

Listing 6.36 shows the implementation of the encrypt and decrypt message functions using the AES algorithm.

```
1  const encryptMessage = async (message, sharedKey, username) => {
2    const iv = encoder.encode(username.repeat(16 / username.length
       + 1).substring(0, 16));
3    return Array.from(new Uint8Array(
4      await crypto.subtle.encrypt(
5        {
6          name: "AES-CBC",
7          iv: iv
8        },
9        sharedKey,
10       encoder.encode(message)
11     )
12   ));
13 }
14
15 const decryptMessage = async (encryptedMessage, sharedKey,
       username) => {
16   const iv = encoder.encode(username.repeat(16 / username.length
       + 1).substring(0, 16));
17   return decoder.decode(await crypto.subtle.decrypt(
```

```
18      {
19        name: "AES-CBC",
20        iv: iv
21      },
22      sharedKey,
23      encryptedMessage
24    ));
25  };
```

Listing 6.36: Encrypt and decrypt message functions

6.8.4 User Interface

When the users first open the application, they will see the welcome screen shown in figure 6.2, where they can sign in or register without providing any sensitive or personal information. If the user provides the invalid credentials, they will see the error message displayed in figure 6.3. After the successful registration, the user will see the main page displayed in figure 6.4. The user can create a new conversation by typing the recipient name in the application header and clicking the blue button close to it, indicating the conversation initiation. After doing so, the conversation will appear on the left side. It will automatically open, allowing the user to type the message in the bottom input field, as displayed in figure 6.5. After typing and sending the message, the initial end-to-end encrypted message will be sent to the recipient, as shown in figure 6.6. When the user receives the new message, the conversation appears at the top of the conversations list, indicating the new message received using the asterisks shown in figure 6.7 that will disappear on the conversation open. Figure 6.8 displays the successful conversation after receiving the initial message.

6.9 Documentation

The Web Application Prototype section sufficiently describes the application functionality and how to use it. However, there is a need to document the backend services installation and its APIs. The */docs* directory, which is one of the attachments of this thesis, contains the documentation for all services in separate directories. The file *manual.pdf* describes the requirements and instructions to build and run the service in a Docker container. The file *api.pdf* describes the service endpoints and the way to access them.

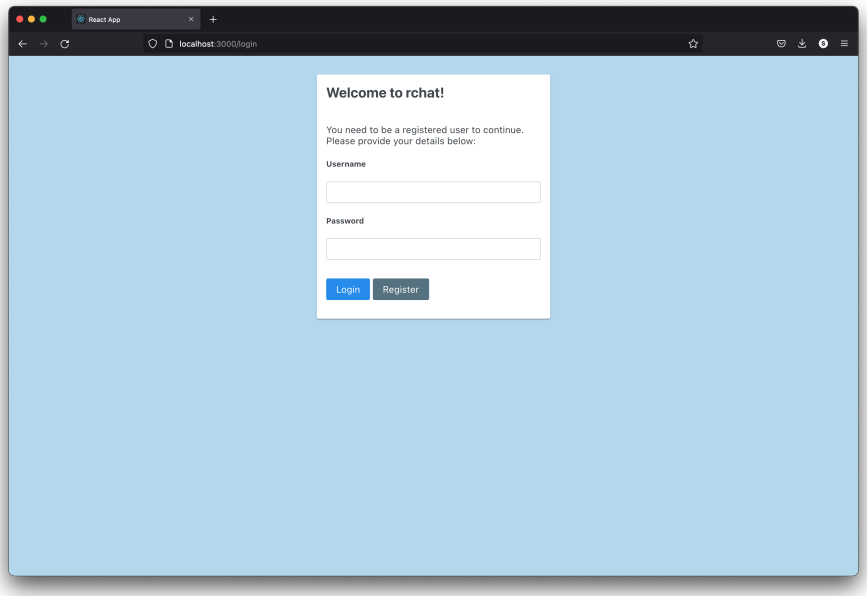

Figure 6.2: Welcome screen

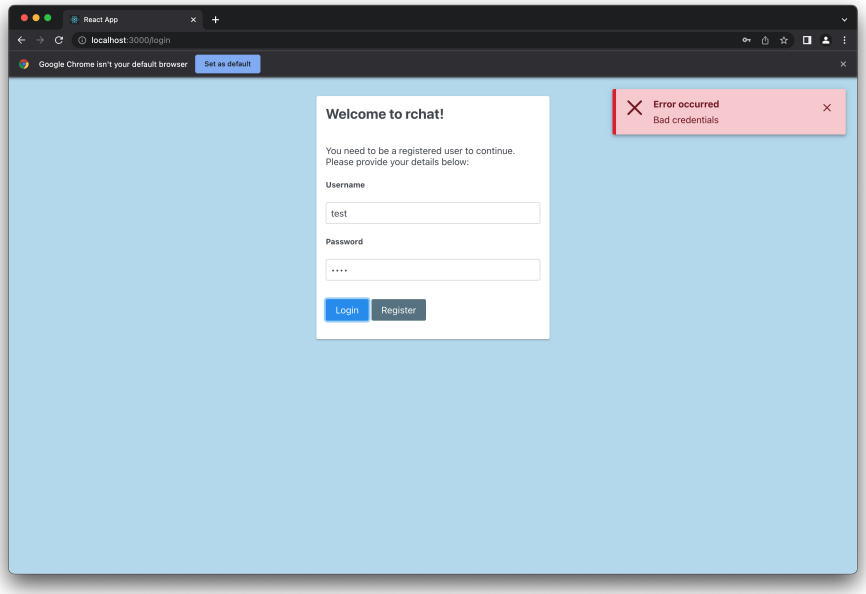

Figure 6.3: Bad credentials alert

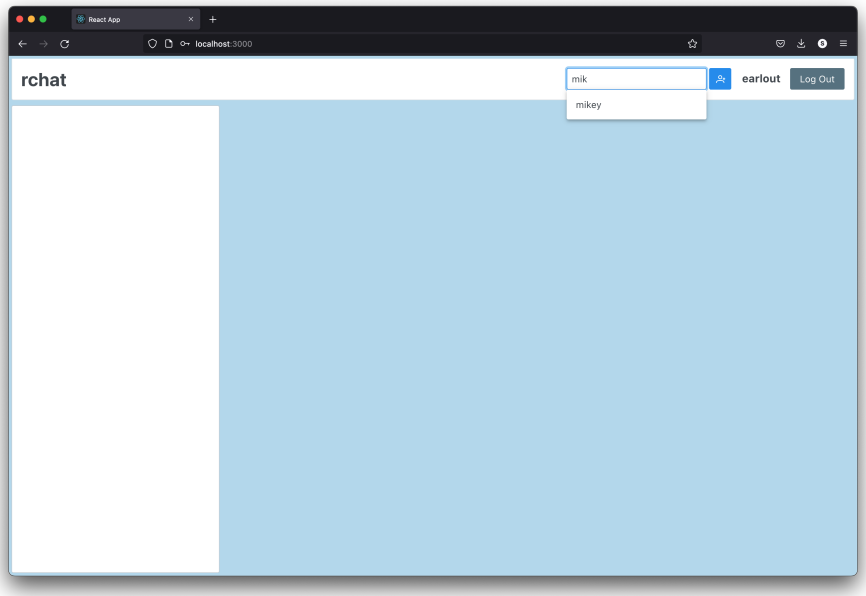

Figure 6.4: Using the search field to start the conversation

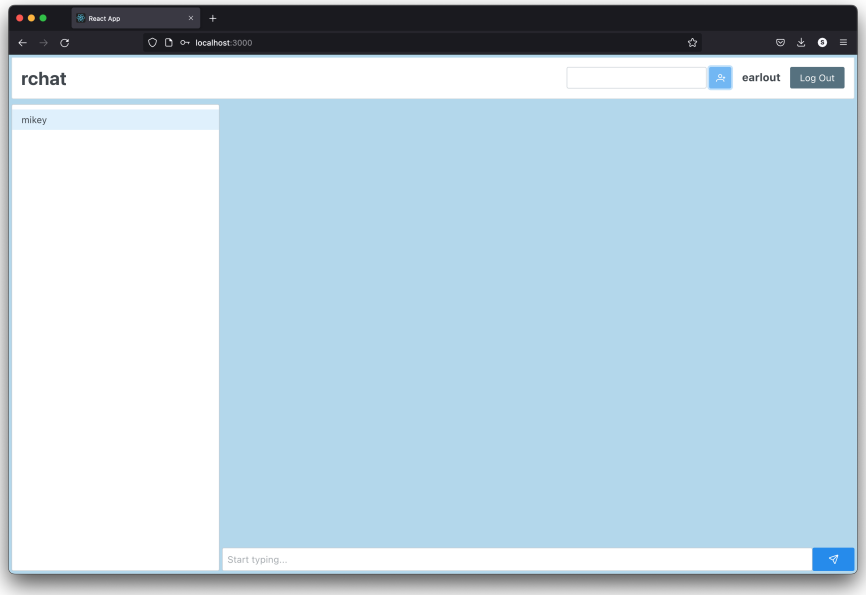

Figure 6.5: New conversation screen

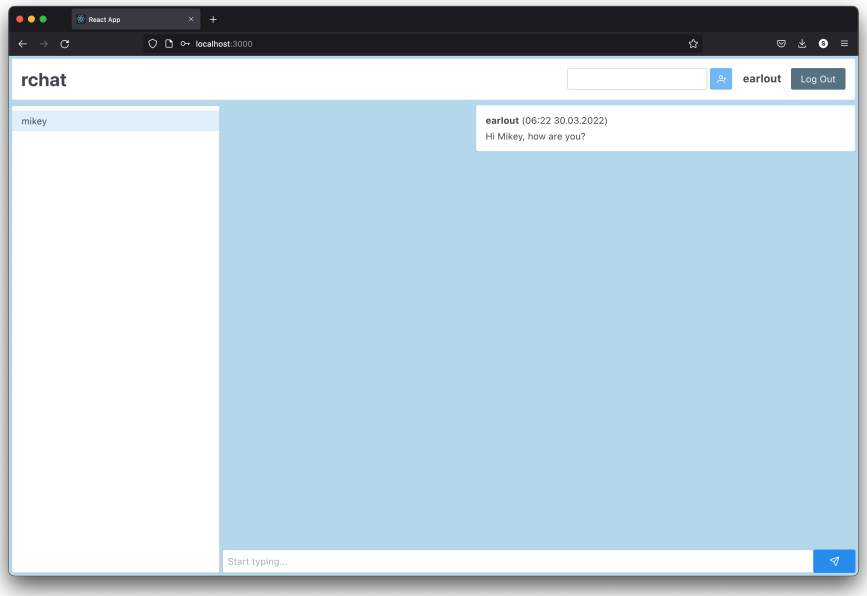

Figure 6.6: Sending the initial message

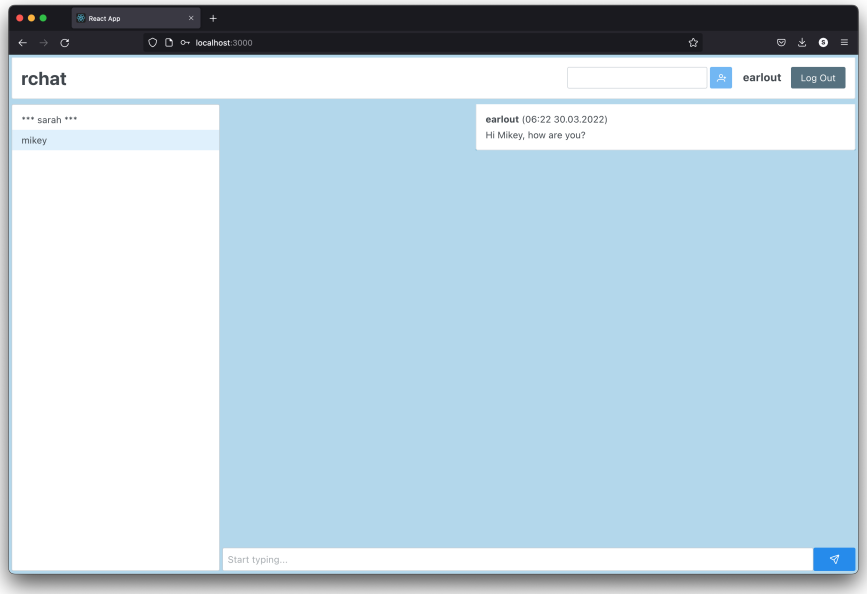

Figure 6.7: Receiving the new message

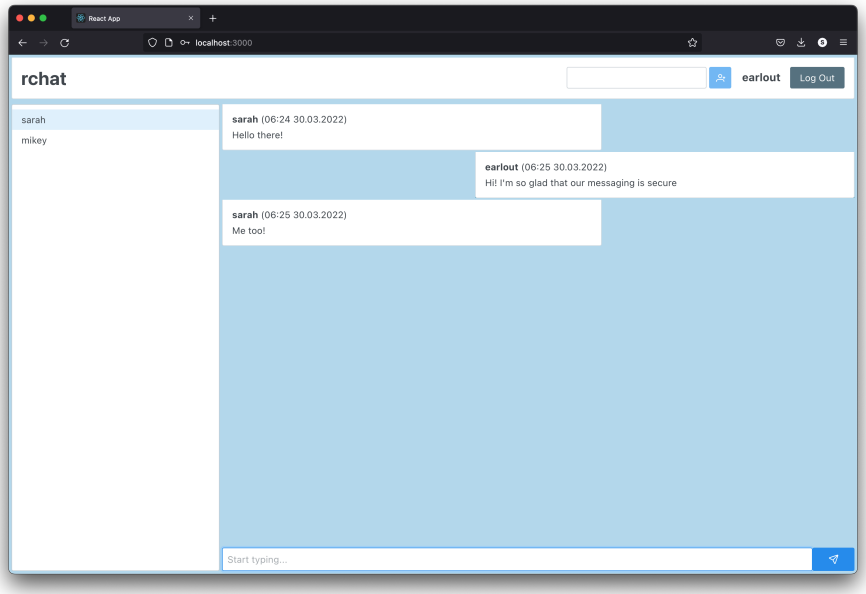

Figure 6.8: Successful conversation after the initial message received

Chapter 7

Evaluation

7.1 IM Platform Evaluation

The implemented Instant Messaging Platform prototype uses the microservices architecture and supports scalability and extensions by design as described in Analysis and Design chapter. The extensions can be done differently based on the purpose. In the case of the chatbots, it may be reasonable to add a new service handling it because of the different types of the load. Group chats can be implemented by adding a new conversation type.

The different sendable content can be implemented by adding a new service responsible for storing it. However, the platform already supports it because it does not restrict the message content field allowing the frontend to store everything it can handle and display. All extensions require further analysis, but the platform's architecture supports them.

The author designed the backend services to support the features like multiple devices by storing the end-to-end encrypted messages on the server. Delivery Service supports multiple devices by having different message channels for different devices. However, the Web Application prototype needs to provide a way to share all the keys from the primary device to the device that logs in, which was not implemented during this thesis.

The application supports all basic features required for the Instant Messaging Platforms. Furthermore, it respects the security by implementing end-to-end encryption. Another benefit is respecting users' privacy — the system does not store any sensitive or private information. It does not require the username to be linked or linkable information.

The essential feature is also the easy and reliable deployments. Thanks to the containerized architecture, there is no dependence on the developer's machine system during the development, which decreases the probability of unexpected hidden errors in production.

7.2 Possible Future Steps

Since the Instant Messaging Platform implemented in this thesis is the prototype, it lacks some feature important for IM to be competitive in the market.

One of the mentioned features is, for example, the support of multiple devices on the frontend. The possible implementation is by getting the QR code with encoded secret keys. Alternatively, to provide more security — posting the way of communication with the public key to encrypt all shared data, which can use the QR code to simplify the process. However, it requires further investigation of possible risks to find the best solution.

Another critical improvement may be to add the ability to backup the secrets. If the user loses access to the account or all devices, they can still access the application and messages on another device. One of the possible implementations may be to store it in another microservice encrypted with the key shared with the user, so they can store it in a secure place and use it for the next login. This implementation will also resolve a problem with supporting multiple devices by allowing the user to provide this key when logging in.

One of the most significant improvements is storing the messages in the queue to deliver the messages for the currently offline user when they are back online.

Storing the messages in queues will also allow the users to initiate the communication when the recipient is offline. Because, in the current implementation, the user does not have any chance to get the shared key for the initial message, not from the Delivery Service. The shared key problem may also be resolved by another microservice, which will temporarily store the shared key until all devices receive it.

Conclusion

The goal of this thesis was based on the analysis to design and implement an open source Instant Messaging Platform that will be possible to deploy on-premise and adjust security mechanisms. It was essential to design the application using the microservices architecture supporting the scalability and extensions.

The output of this thesis is the Instant Messaging Platform prototype. Its implementation uses the microservices architecture with three services separated by the purpose: Users Service, Chat Service, and Delivery Service. As mentioned in the thesis chapters, this implementation supports scalability and extensions.

The author analyzed some of the popular Instant Messaging Platforms present on the market and primary security mechanisms related to IM, defined the functional and non-functional requirements, and described the use cases. In addition, the author described the selected technologies, documented the output, and evaluated the solution. The author implemented the Unit Tests to test the complex logic and tested the API using the Postman tool.

The designed prototype supports the end-to-end encryption to maintain the users' conversations security and respects the privacy by not collecting the personal information. It stores the end-to-end encrypted messages on the server to support the multiple devices, which the delivery logic also supports. However, the prototype still has the ways for improvement, and the Evaluation chapter covers it.

Bibliography

[1] GeeksforGeeks. Client-Server Model. *GeeksforGeeks Tutorials [online]*, November 2019, [accessed on 2022-03-27]. Available from: `https://www.geeksforgeeks.org/client-server-model/`

[2] Imperva. Man in the middle (MITM) attack. *Imperva Learning Center [online]*, 2022, [accessed on 2022-03-20]. Available from: `https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/`

[3] Kemp, S. Digital 2022 Global Overview Report (January 2022) v05. *DataReportal*, January 2022: p. 99. Available from: `https://www.slideshare.net/DataReportal/digital-2022-global-overview-report-january-2022-v05`

[4] LLC, W. Download. *WhatsApp Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: `https://www.whatsapp.com/download`

[5] handlerug. Telegram macOS screenshots picture. *GitHub [online]*, April 2019, [accessed on 2022-04-03]. Available from: `https://github.com/overtake/TelegramSwift/blob/master/images/tg.png`

[6] Signal. Get Signal. *Signal Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: `https://signal.org/download/`

[7] Slack. Slack Features. *Slack Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: `https://slack.com/features`

[8] Abhinav Asthana. Postman. Available from: `https://www.postman.com`

[9] Maina, T. Instant messaging an effective way of communication in workplace. 10 2013.

[10] WhatsApp LLC. WhatsApp. Available from: `https://www.whatsapp.com`

[11] Tencent Holdings Limited. WeChat. Available from: https://www.wechat.com

[12] Meta Platforms. Messenger. Available from: https://www.messenger.com

[13] Shenzhen Tencent Computer System Co., Ltd. Tencent QQ. Available from: https://im.qq.com

[14] Snap Inc. Snapchat. Available from: https://www.snapchat.com

[15] Telegram Messenger Inc. Telegram. Available from: https://telegram.org

[16] BBC. Network hardware. *Bitesize [online]*, February 2022: p. 7, [accessed on 2022-03-27]. Available from: https://www.bbc.co.uk/bitesize/guides/zh4whyc/revision/7

[17] Sarangam, A. What Is Client Server Architecture? An Overview. *Jigsaw Academy [online]*, December 2020, [accessed on 2022-03-27]. Available from: https://www.jigsawacademy.com/blogs/cyber-security/what-is-client-server-architecture/

[18] Avron, S. Advantages and Disadvantages of a Peer-to-Peer Network. *Flevy Blog [online]*, May 2021, [accessed on 2022-03-27]. Available from: https://flevy.com/blog/advantages-and-disadvantages-of-a-peer-to-peer-network/

[19] Dictionary, C. Definition of 'security'. *Collins Dictionary [online]*, March 2022, [accessed on 2022-03-18]. Available from: https://www.collinsdictionary.com/dictionary/english/security

[20] Schuster, S.; Berg, M.; et al. Mass Surveillance and technological policy options: Improving security of private communications. *Computer Standards & Interfaces*, 09 2016, doi:10.1016/j.csi.2016.09.011, [accessed on 2022-03-20].

[21] Cloudflare. What is SSL? — SSL definition. *Cloudflare Learning [online]*, 2022, [accessed on 2022-03-20]. Available from: https://www.cloudflare.com/en-gb/learning/ssl/what-is-ssl/

[22] Olenski, J. SSL vs TLS - What's the Difference? *GlobalSign Blog [online]*, February 2020, [accessed on 2022-03-20]. Available from: https://www.globalsign.com/en/blog/ssl-vs-tls-difference

[23] Munteanu, R. SSL Certificates vs. Man-in-the-middle attacks. *Medium [online]*, October 2019, [accessed on 2022-03-20]. Available from: https://medium.com/@munteanu210/ssl-certificates-vs-man-in-the-middle-attacks-3fb7846fa5db

[24] Nohe, P. The difference between Encryption, Hashing and Salting. *The SSL Store Blog [online]*, December 2018, [accessed on 2022-03-20]. Available from: https://www.thesslstore.com/blog/difference-encryption-hashing-salting/

[25] Open Whisper Systems. Signal Protocol. Available from: https://signal.org/docs/

[26] Dion van Dam. *Analysing the Signal Protocol*. Master's thesis, Radboud University, Houtlaan 4, 6525 XZ Nijmegen, Netherlands, 2019.

[27] Warren, S. D.; Brandeis, L. D. The Right to Privacy. *Harvard Law Review*, volume 4, no. 5, 1890: pp. 193–220, ISSN 0017811X. Available from: http://www.jstor.org/stable/1321160

[28] Stone, E. F.; Gueutal, H. G.; et al. A field experiment comparing information-privacy values, beliefs, and attitudes across several types of organizations. *Journal of Applied Psychology*, volume 68(3), 1983: pp. 459–468. Available from: http://www.jstor.org/stable/1321160

[29] Durnell, E.; Okabe-Miyamoto, K.; et al. Online Privacy Breaches, Offline Consequences: Construction and Validation of the Concerns with the Protection of Informational Privacy Scale. *International Journal of Human–Computer Interaction*, volume 36, no. 19, 2020: pp. 1834–1848, doi:10.1080/10447318.2020.1794626. Available from: https://doi.org/10.1080/10447318.2020.1794626

[30] Matuszewska, K.; Lubowicka, K.; et al. What is PII, non-PII, and personal data? *Piwik Pro Blog [online]*, April 2021, [accessed on 2022-03-21]. Available from: https://piwik.pro/blog/what-is-pii-personal-data/#what-is-personally-identifiable-information-(pii)?

[31] United States Government Accountability Office. PRIVACY Alternatives Exist for Enhancing Protection of Personally Identifiable Information. *Report to Congressional Requesters*, May 2008, [accessed on 2022-03-29]. Available from: https://www.gao.gov/assets/gao-08-536.pdf

[32] Proton Technologies AG. Art. 4 GDPR - Definitions. *General Data Protection Regulation [online]*, May 2016, [accessed on 2022-03-29]. Available from: https://gdpr.eu/article-4-definitions/

[33] Red Hat, Inc. What is open source? *Red Hat Topics [online]*, October 2019, [accessed on 2022-03-29]. Available from: https://www.redhat.com/en/topics/open-source/what-is-open-source

[34] Open Source Initiative. The Open Source Definition. *Open Source Initiative website [online]*, March 2007, [accessed on 2022-03-29]. Available from: https://opensource.org/osd

[35] Signal Foundation and Signal Messenger LLC. Signal. Available from: https://signal.org

[36] Kraus, R. What is Signal? The basics of the most secure messaging app. *Mashable [online]*, July 2021, [accessed on 2022-04-02]. Available from: https://mashable.com/article/what-is-signal-app

[37] Slack Technologies. Slack. Available from: https://slack.com

[38] Olson, P. Exclusive: The Rags-To-Riches Tale Of How Jan Koum Built WhatsApp Into Facebook's New $19 Billion Baby. *Forbes [online]*, February 2014, [accessed on 2022-04-03]. Available from: https://thehackernews.com/2021/01/whatsapp-will-delete-your-account-if.html

[39] Lakshmanan, R. WhatsApp Will Disable Your Account If You Don't Agree Sharing Data With Facebook. *The Hacker News [online]*, January 2021, [accessed on 2022-04-03]. Available from: https://thehackernews.com/2021/01/whatsapp-will-delete-your-account-if.html

[40] of Android, C. Use a third-party WhatsApp client and you could be banned for life. *Cult of Mac [online]*, March 2015, [accessed on 2022-04-03]. Available from: https://www.cultofmac.com/314343/use-a-third-party-whatsapp-client-and-you-could-be-banned-for-life/

[41] Lawler, R. WhatsApp multi-device beta allows four devices at once even without a phone. *The Verge [online]*, July 2021, [accessed on 2022-04-03]. Available from: https://www.theverge.com/2021/7/14/22577594/whatsapp-multi-device-e2e-facebook

[42] LLC, W. About linked devices. *WhatsApp Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: https://faq.whatsapp.com/general/download-and-installation/about-linked-devices/?lang=en

[43] Inc., T. M. Telegram Applications. *Telegram Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: https://telegram.org/apps

[44] Telegram Messenger Inc. Telegram Database Library. Available from: https://core.telegram.org/tdlib

[45] Open Whisper Systems. RedPhone.

[46] Open Whisper Systems. TextSecure.

[47] McCall, V.; Smith, B. What is Signal? How the popular encrypted messaging app keeps your texts private. *Business Insider [online]*, October 2021, [accessed on 2022-04-03]. Available from: https://www.businessinsider.com/signal-app

[48] Greenberg, A. Your iPhone Can Finally Make Free, Encrypted Calls. *Wired [online]*, July 2014, [accessed on 2022-04-03]. Available from: https://www.wired.com/2014/07/free-encrypted-calling-finally-comes-to-the-iphone/

[49] Slack. Slack Downloads. *Slack Web Pages [online]*, April 2022, [accessed on 2022-04-03]. Available from: https://slack.com/downloads

[50] MongoDB Inc. MongoDB. Available from: https://www.mongodb.com/

[51] go-chi. chi. Available from: https://github.com/go-chi/chi

[52] go chi. chi. *GitHub [online]*, April 2022, [accessed on 2022-04-28]. Available from: https://github.com/go-chi/chi

[53] go-chi. CORS net/http middleware. Available from: https://github.com/go-chi/cors

[54] go-chi. render. Available from: https://github.com/go-chi/render

[55] MongoDB Inc. MongoDB Go Driver. Available from: https://www.mongodb.com/docs/drivers/go/current/

[56] dgrijalva. jwt-go. Available from: https://github.com/dgrijalva/jwt-go

[57] dgrijalva. jwt-go. *GitHub [online]*, April 2022, [accessed on 2022-04-28]. Available from: https://github.com/dgrijalva/jwt-go

[58] Google. Go Cryptography. Available from: https://pkg.go.dev/golang.org/x/crypto#section-readme

[59] OpenBSD. CVS log for src/lib/libc/crypt/bcrypt.c. *OpenBSD CVS [online]*, February 1997, [accessed on 2022-04-28]. Available from: https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/crypt/bcrypt.c

[60] Meta Platforms. React. Available from: https://reactjs.org/

[61] PrimeFaces. PrimeReact. Available from: https://www.primefaces.org/primereact/

[62] remix-run. react-router. Available from: https://github.com/remix-run/react-router

[63] w3schools. React Router. *w3schools [online]*, April 2022, [accessed on 2022-04-28]. Available from: `https://www.w3schools.com/react/react_router.asp`

[64] Mozilla.org. SubtleCrypto. *developer.mozilla.org [online]*, February 2022, [accessed on 2022-04-28]. Available from: `https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto#browser_compatibility`

[65] Docker Inc. Docker. Available from: `https://www.docker.com/`

Appendix A

Acronyms

API Application Programming Interface

CCPA California Consumer Privacy Act

CORS Cross-Origin Resource Sharing

DOM Document Object Model

EU European Union

GDPR General Data Protection Regulation

HTTP Hypertext Transfer Protocol

IM Instant Messaging

IP Internet protocol

IRC Internet Relay Chat

JSON JavaScript Object notation

JWT JavaScript Web Tokens

MAC Media access control

P2P Peer-to-Peer

PII Personally Identifiable Information

SSE Server-Sent Events

SSL Secure Sockets Layers

TLS Transport Layer Security

TTL Time to Live

UI User Interface

URL Uniform Resource Locator

US United States

Appendix B

Contents of enclosed SD card

- `readme.txt` ..... the file with SD card contents description
- `docs` ..... the directory of documentation
  - `users-service` ..... the documentation of Users Service
    - `api.pdf` ..... the API documentation
    - `manual.pdf` ..... the instructions to build and run the service
  - `chat-service` ..... the documentation of Chat Service
    - `api.pdf` ..... the API documentation
    - `manual.pdf` ..... the instructions to build and run the service
  - `delivery-service` ..... the documentation of Delivery Service
    - `api.pdf` ..... the API documentation
    - `manual.pdf` ..... the instructions to build and run the service
- `src` ..... the directory of source codes
  - `impl` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `DP_Volodin_Vladyslav_2022.pdf` ..... the thesis text in PDF format