ICEBREAKER

CHAT APP

Be ready to break the ice

ABSTRACT

This Document summarizes how we were able to develop a real-time chat application with a generative AI feature

Bilal Alali, Utkarsh Fulara, Andrej Sum-Shik

Course “Informatik Projekt” with a Focus on AI held by Prof. Dr. -Ing. Markus Miettinen

Table of Content

Table of figures

1. Introduction

Nowadays the term AI is discussed in nearly every corner of this world. The Term AI refers to artificial intelligence. But what that actually means is hard to tell, as we humans do not have a decisive definition of the term intelligence. Mainly because our own intellect is still a mystery to us. The term artificial seems a lot more straightforward, as it refers to something that is man-made and not naturally produced. (Although as every component in every machine is made out of particles, you cannot definitely prove that there is not a planet in the universe where this particular machine was made by nature). To still derive a general definition artificial intelligence refers to a man-made machine which exhibits cognitive skills similar to that of a human. In recent years AI has made further progress and was able to sophisticate itself in everyday life, that is to escape from the purely scientific world and acquiring general relevance in the world of all human beings. ChatGPT made a significant contribution to this change, as it allowed every human being to come into contact with the scientific and mathematical world of AI, without having to understand any of the complex structure behind it. For most of us ChatGPT has become an irreplaceable companion that helps us in our everyday tasks. Especially for university students this technology became a brand-new way of learning and understanding complex concepts in every branch of research. But as popular as AI is, it is also a highly controversial topic. Many fear the rise of AI, as it has the potential to become a lot smarter than us. Although, there a valid reasons for this fear, a bigger problem right now stems from the fact that most people discussing this topic do not actually understand how AIs like ChatGPT work[1]. ChatGPT does not actually understand us, what it does instead is predict the most likely next word for an input. It then concatenates the input and the predicted word as a new input and repeats this process. This is resumed until there is a response. In this project we want to leverage this "skill" ChatGPT has, we take user profiles as an input and try to produce an output for that. We call the desired output "Icebreakers" and as the name implies, we want this system to generate good first messages for a given user profile. All of this is part of a real-time chat application. How we realised this task and what problems and obstacles we faced will be discussed in this report. Furthermore, this report serves as a first dive into the strange and scientific world of AI, that we as team began diving into since the start of this semester (October 2024).

[1] Vuong, Q., La, V., Nguyen, M., Jin, R., La, M. & Le, T. (2023b). How AI's Self-Prolongation Influences People's Perceptions of Its Autonomous Mind: The Case of U.S. Residents. *Behavioral Sciences*, *13*(6), 470. https://doi.org/10.3390/bs13060470

2. What is our app?

Our application is a real-time chatting website. Here users have accounts and can see all other users of this application. Each user has a profile with a description. When a new user: Laura logs in, she can select another user: Bilal. Then our AI analyses the description that Bilal wrote about himself and suggests Laura a few potential first messages she can send to Bilal. For instance: Bilal wrote about himself that he likes boxing, so a potential first message may be: “Boxing is so nerve-wrecking. Who is your favourite Boxer?” Laura selects this Icebreaker and sends it as a first message. A new chat now exists between Laura and Bilal. Here they can spend as much time texting each other as they want and best of all, only they know the content of their conversation. Because we value user privacy, we want this chatting feature to be end-to-end encrypted, this means only Laura and Bilal have the means to decrypt the exchanged messages.

2.1 High level requirements.

The most important requirements this app must fulfill are the following:

- The generated Icebreakers reflect the user’s description
- The chat feature is real time, meaning instantaneous messaging and sending messages without waiting for the recipient to “acknowledge” them
- The user’s keys and passwords are securely stored
- The UI design is intuitive and easily accessible
- A secure login system
- All chat messages are stored
- A user can adjust his profile to his/her liking
- The website can handle multiple users using the website at once
- Chat messages are stored in an encrypted form
- Chat messages are encrypted while in transit
- A user can search for specific users
- A user sees only the users in the overview component, whom he does not have a chat with
- Automatically logout inactive users
- Encryption is robust enough to prevent brute force attacks
- Only authenticated users can use the website
- Make something to distract users while icebreakers are loading
- Find a balance between security and accessibility
- Users can mark themselves as “inactive” so to not be disturbed
- Each message is accompanied by a timestamp
- Users can logout manually
- There shall be a minimum character number for passwords

2.2 High level design

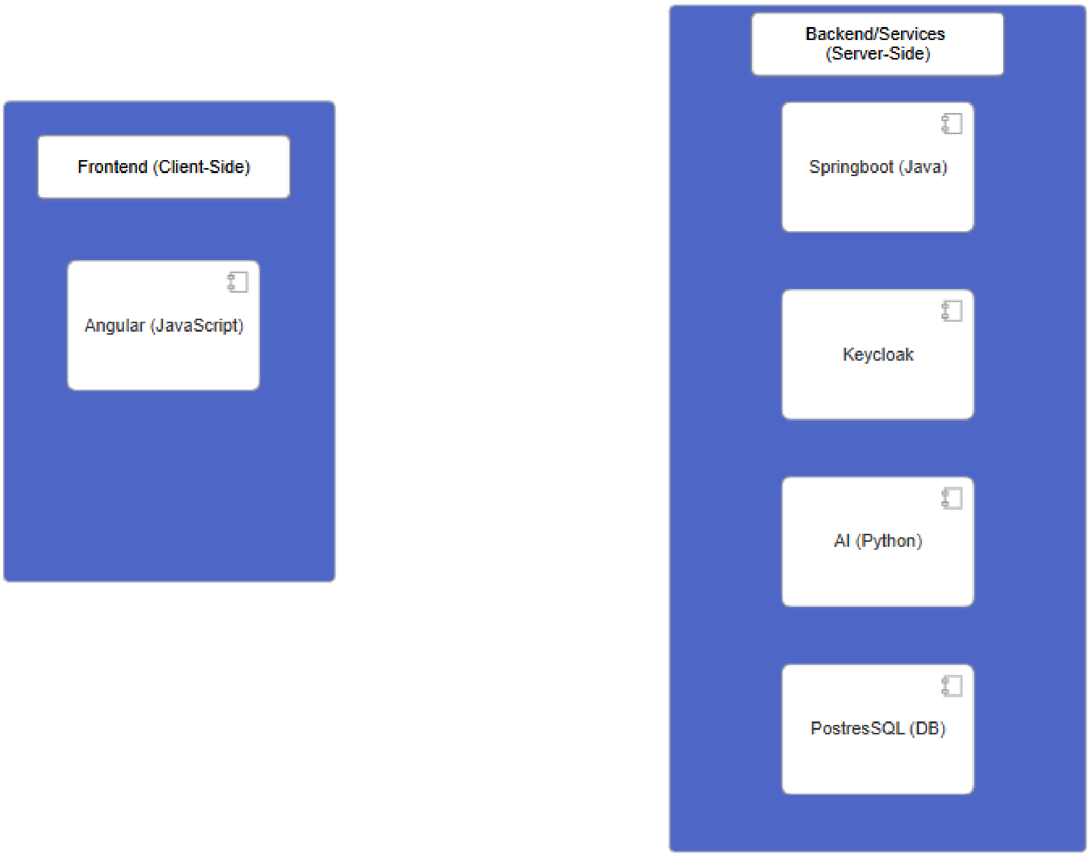
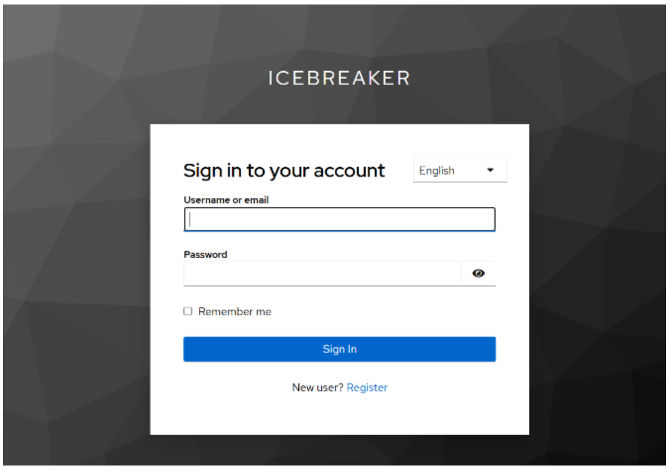


Figure 1 High Level Component Overview

When accessing the website users first meet a login screen, here they can choose to create a new account or sign in with an existing one. Users can also choose which language they prefer: German or English.



ICEBREAKER

Sign in to your account

English

Username or email

Password

☐ Remember me

New user? Register

Figure 2: Keycloak Login Terminal

When the login was successful new users have to create a passphrase and a magic number, “old” users will only have to confirm their passphrase.

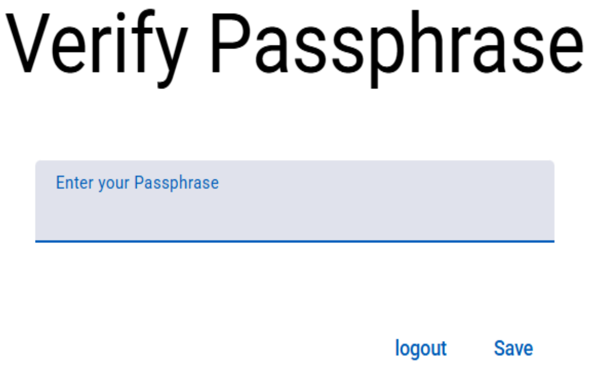


Figure 3: Pass Phrase Input

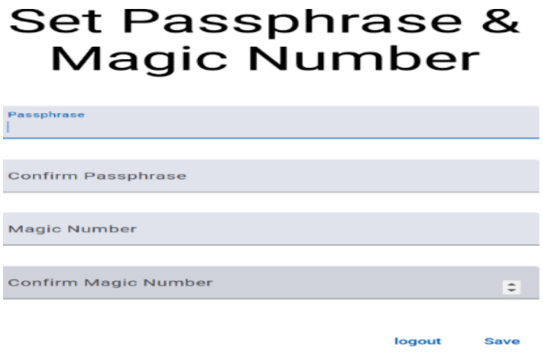
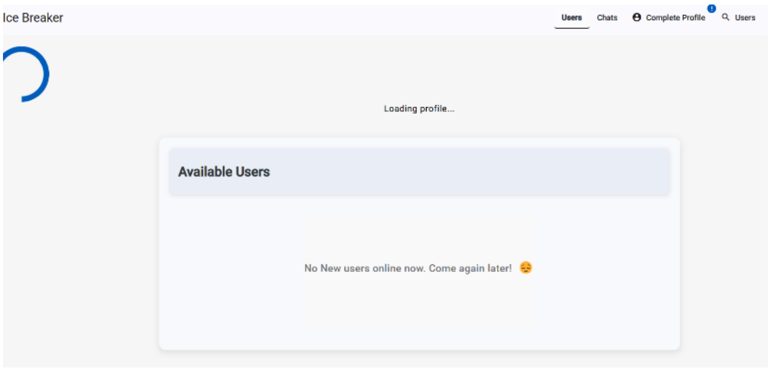


Figure 4: Passphrase Setup

After this if a new user will be prompted to “complete” his profile.

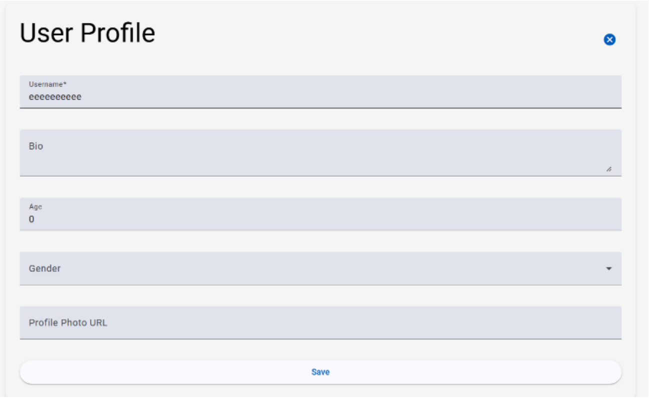


Loading profile...

No New users online now. Come again later! 😔

Figure 5: Available Users-Home Screen

When the user then clicks on complete profile, he will need to write a description about him/herself disclose their age and their gender, additionally, users can choose to use a profile photo.



User Profile

Username*
eeeeeeeeee

Bio

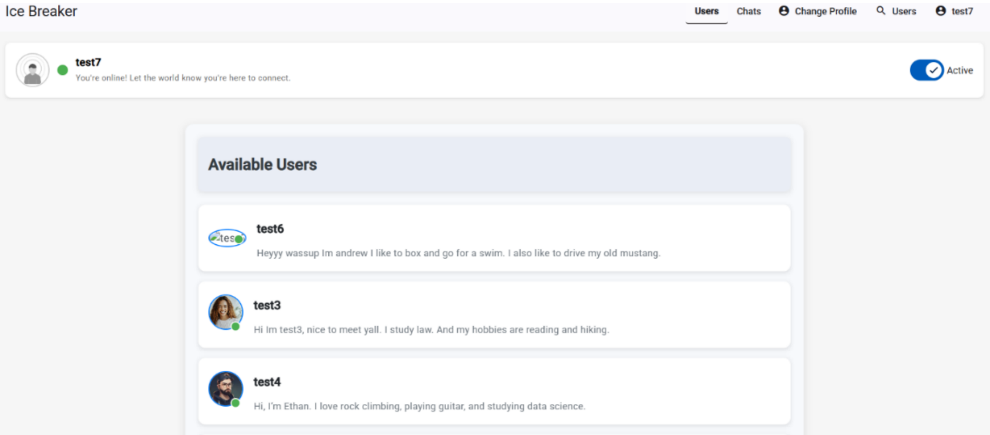
Age
0

Gender

Profile Photo URL

Figure 6: Public User Profile

After that the user is greeted with the following screen



test7
You're online! Let the world know you're here to connect.

Available Users

test6
Heyyy wassup Im andrew I like to box and go for a swim. I also like to drive my old mustang.

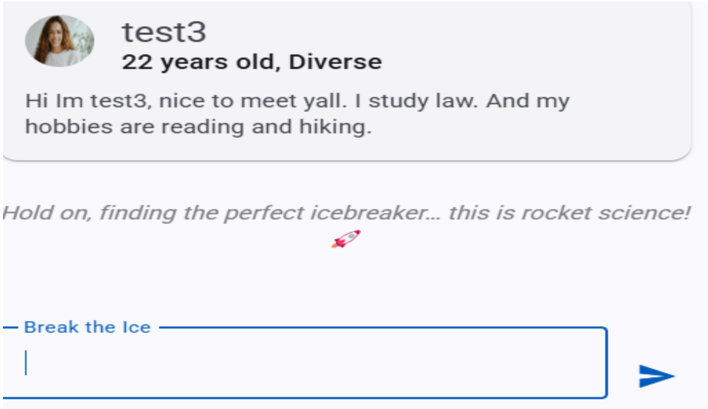
test3
Hi Im test3, nice to meet yall. I study law. And my hobbies are reading and hiking.

test4
Hi, I'm Ethan. I love rock climbing, playing guitar, and studying data science.

Figure 7: Online available Users

Here the user can see all other “available” users. The term “available” in this context means active users, whom this user does not yet have a chat with. The user can see the description, username and picture (if provided) of every “available” user. Furthermore, a user can set himself as inactive, change his profile and search for users.

When clicking on a user profile the user can send a first message to this person, there the icebreakers are also generated, this takes some time, so in the mean time the user sees various messages as to distract him/her from the passing time.



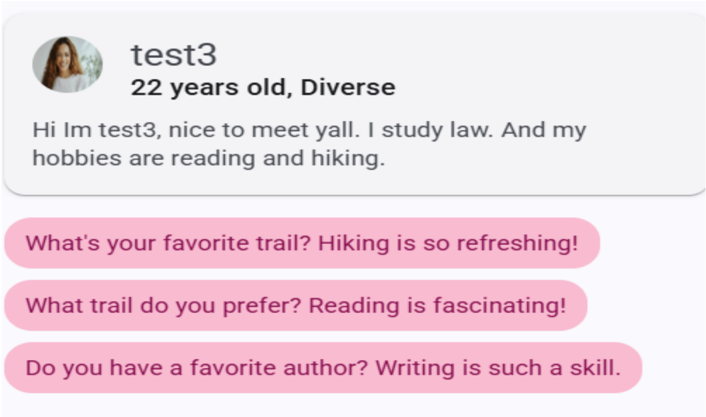
test3

22 years old, Diverse

Hi Im test3, nice to meet yall. I study law. And my hobbies are reading and hiking.

Hold on, finding the perfect icebreaker... this is rocket science! 🚀

Figure 8: One particular 'Waiting Time Slogan'



test3

22 years old, Diverse

Hi Im test3, nice to meet yall. I study law. And my hobbies are reading and hiking.

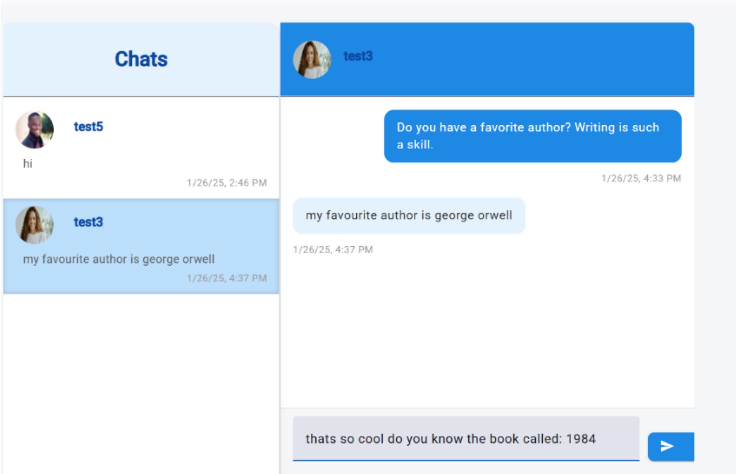
What's your favorite trail? Hiking is so refreshing!

What trail do you prefer? Reading is fascinating!

Do you have a favorite author? Writing is such a skill.

Figure 9: Generated Icebreakers for this Profile

When a user sends a first message a chat is created, this chat can be viewed when the user clicks on chat overview.



test5

hi

1/26/25, 2:46 PM

test3

my favourite author is george orwell

1/26/25, 4:37 PM

test3

Do you have a favorite author? Writing is such a skill.

1/26/25, 4:33 PM

my favourite author is george orwell

1/26/25, 4:37 PM

thats so cool do you know the book called: 1984

Figure 10: Chat Screen with E2EE

This is what the user sees afterwards, here all of his chats are displayed. When he/she selects a particular chat (like here with test3) he/she can see all exchanged messages along with timestamps for each message. The messages on the right-hand side are the ones the user wrote him/herself, the messages on the left-hand side are the messages of the other participant.

2.3 High level implementation of the main components

2.3.1 AI feature

The feature is a fine-tuned version of the pretrained gpt2-medium model. In the following it will be described step-by-step how generative AIs work and how we can leverage that for our own purposes.

Machine learning and neural networks
Most of the modern AI technology revolves around so called “neural networks”. The idea is that the initial input is split into much smaller features. For example, a picture may be split into pixels. These features are then multiplied by so called “weights”, at first the values for these weights are random, this process can also be described as a matrix multiplication. Because many inputs (for example multiple pictures each with 1000 pixels for instance) are multiplied by a variety of weights. (The height of the input matrix may be the number of pictures and the width the number of pixels). Each output from the matrix multiplication (weights × inputs) is called a pre-activation. A bias term is added to each pre-activation, acting as a learnable offset that adjusts the neuron’s sensitivity. The result is then passed through a non-linear activation function (e.g., ReLU), which returns 0 for negative inputs and the input itself for positive values. This final output is called the neuron’s activation, loosely inspired by biological neurons that ‘fire’ when their input crosses a threshold. This described process is then repeated x times. At last, there is an output, also called a “prediction”, (for example there may be 10 outputs representing the numbers 0-9, the neuron with the highest number at the end represents the predicted number by the neural network). As expected at first this output will not make any sense but that is precisely why we need to train the AI. In the training process we know the desired output for each input. We then compare each predicted output with the actual output. We can use functions like mean squared error, to estimate how bad a certain prediction actually is and then build an estimate of all predictions. But since the prediction depends on the weights which can be changed, the weights can be seen as variables and the average, the so called “loss” is the outcome of a function with these variables. Since now the whole system can be described as multi-dimensional function, we can also find a minimum for it (could also be a local minimum). But in this context a minimum would correspond to a system which makes the (mostly) correct prediction, this is our desired outcome. But how do we find this minimum, in calculus we learned that functions have derivatives which describe the slope of the function, and a multi-dimensional function has a gradient, the derivatives with respect to all variables, this gradient points into the direction of the steepest slope, if we were to go into the opposite direction of said gradient we can find a minimum. This is what backpropagation is all about. We calculate the derivatives for all

the weights and subtract that from the current weights (often multiplied by a learn rate). This is the basis for basically all modern AI models.

Generative AI Models

New models took this idea and refined it for their needs, as computational power rose, the models could also become more complex and nuanced, the so-called transformer model used by ChatGPT was born. How does that work? ChatGPT takes an input of words, it then maps these words or tokens (tokens do not necessarily have to be words) into vectors. These vectors go through many layers of layers of multiplications and at last these vectors are used to predict a subsequent vector; this is the quintessence of generative AI models like ChatGPT. This new sentence will then form the next input to then predict the yet next vector, this is repeated until a response is formed. Especially notable is the "Attention" model[2], used in between mapping the tokens into vectors and then using the final vector to predict the next. To put it simply each of the sentence's token's embeddings (the vectors) is mapped into query, key and value vectors using matrices. Then the dot product of all key and query vectors is computed and then divided by the square root of the dimension size of the vectors (for scaling), this is put into softmax (a function scaling all values between 0 and 1) to find out the probability distribution of the results. (a high probability means a strong relevance between tokens). This in turn is multiplied by the value vectors; by learning how closely each of the embedding vectors are, the AI can learn the structure of the sentences. If the AI models is fed with a lot of sentences and information, it could thus, learn its structure and reproduce it. This is the magic behind how ChatGPT is able to predict the next word for an input. (A few important aspects to mention are generative AI models often use Gelu instead of Relu and a cross-entropy loss instead of the mean squared error; masked self-attention is also important so that no "future" tokens are used when predicting the next token during training. Lastly, the model uses positional embeddings so that vectors have their initial position in the sentence encoded).

Our fine-tuned model

With this knowledge our use of the gpt2-medium model can be explained. We wanted the model to learn the structure of user profiles and corresponding "icebreakers", thus, we had to create 200-300 of these examples. (For professional purposes a lot more a needed). These are then fed into the gpt2-medium model to learn the relationship between the profiles and the icebreakers. We used a pretrained model, because, without a pretrained model, gpt2-medium would have no knowledge of grammar or sentence structure and a lot more than 200-300 examples would have been needed to train the model. Now that we understood how the model works it also makes sense why feeding the model the following structure for learning would not bring any result: "Profile

[2] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017). Attention is All you Need. *arXiv (Cornell University)*, *30*, 5998–6008. https://arxiv.org/pdf/1706.03762v5

A- Icebreaker 1; Profile A- Icebreaker 2…". For someone not understanding how the attention mechanism works this may seem like a way of teaching gpt2-medium the different possible icebreakers for a given user profile. But now we know that this will not work as the model will learn contradicting relationships between the tokens and would not be able to produce valid output. Thus, a better structure would be "Profile A- Icebreaker 1- Icebreaker 2…. Profile B- Icebreaker 1….". With this structure the model will learn all icebreakers corresponding to one user profile. This learned knowledge is then captured in its updated weights, these are used to then predict several icebreakers for new profiles written by users of our app. This works pretty well; the limitations of our fine-tuned model are discussed in the section "Challenges we have faced".

2.3.2 End-to-end encryption

During the discussion, as to how this app shall look like, we derived that it is extremely important that each chat is end-to-end encrypted. The first reason is, that users may not want the server or the provider of the app to know the contents of their conversations and secondly the messages are saved on servers, if they are stored as plaintext, an attacker hijacking the servers may find out all contents of each conversation. Therefore, we needed a way to prevent this.

Upon creating an account, the user creates a passphrase and a magic number. The passphrase alongside the magic number is put into a hash function and only the hash value alongside the magic number (also called salt) is saved on the server. A hash function (SHA-256) is used so that hash(p, m) = x is easy but the reverse is computationally infeasible.

Then in the frontend a new public/private key pair is generated. The private key is encrypted with passphrase to the power of magic number modulo 10^12+33 (a modulo value large enough, so that brute force attacks are infeasible). Thus, in the end it is encrypt(privkey, p^m % 10^12+33) with the encryption algorithm (AES-256 in CBC mode with PKCS7 padding and a 16-byte random IV). The keypair is then uploaded to the server.

When starting a new chat, the frontend retrieves the public key of the recipient (which is not encrypted) and generates a symmetric key (AES-256, returned as a Base64 string). It then encrypts the first message with the symmetric key (using AES-256 in CBC mode with PKCS7 padding and a 16-byte random IV) and then in turn encrypts the symmetric key twice (using RSA-OAEP (2048-bit, SHA-256)), once with the sender's public key and once with the public key of the recipient. These encrypted symmetric keys are uploaded to the server.

When the recipient logs into the website he/she has to type in the passphrase (the magic number is also fetched from the server), the output hash is compared to the one stored on the server; if they match, he/she is permitted to enter. His/her encrypted private key is fetched from the server, and in the frontend the passphrase is used to decrypt it. Now he/she can go to the new chat. For this, the symmetric key (encrypted with this user's public key) is fetched, the decrypted private key is used to decrypt the symmetric key, then the symmetric key is used to decrypt the new message. This lengthy process is only needed in the beginning; from the moment the user fetched the symmetric key for this chat, the key is saved in the local cache and used for encrypting/decrypting all messages during the session (referring to the period where the user uses our website; a session terminates if a user is inactive or logs out. For more details on the implementation please refer to our code, especially E2EEComponents in the Frontend).

2.3.3 Real-time chatting

The idea behind the chat feature was to create a seamless experience where there's no lag during conversations, and users don't have to click any button to refresh the chat. When someone sends a message, the recipient should see it instantly, without doing anything. This made it clear that a traditional request-response architecture wouldn't work.

We started exploring other options and came across the publish-subscribe architecture. In this setup, an event automatically sends notifications to everyone subscribed to it. However, since we were working across two different platforms, implementing this wasn't straightforward. After some research, we found a great solution using WebSocket.

Here's how it works: every time a user starts a chat with someone, a queue is created specifically for the user, and the user is subscribed to it. The queues follow a standard structure, like user/queue/{username}/messages. This consistency allows the server to automatically create queues even if they don't already exist.

When a message is sent, it goes to the server, which then publishes it to the relevant queue of recipient without him needing to make a request. This way, the server acts as a middleman, simply forwarding the messages in real time while also saving them for future use.

This approach gave us a real-time chat system that works smoothly and could even support notifications in the future. Unfortunately, we had to skip adding notifications this time due to time constraints.

2.3.4 Login and authentication

The microservices architecture was adopted throughout the development of our entire project. When implementing authentication and authorization, we had two options:

1. Develop a custom authentication service from scratch and package it as a microservice.

2. Leverage an open-source authentication microservice to handle these tasks efficiently.

After evaluating our options, we decided to integrate Keycloak, an open-source identity and access management solution. We containerized Keycloak using Docker, configuring it to listen on port 8081.

Implementation Overview

The core functionality we implemented involves checking for an authentication token in every outgoing request to the backend. If a request lacks a valid token, the user is automatically redirected to the login page.

To handle this process efficiently, we used an interceptor in our code. If a user is logged in, the interceptor appends an authentication token to each request, ensuring the structure looks like:

request? Auth = token

The backend then retrieves this token and forwards it to the Keycloak microservice for validation. If the token is verified as legitimate, Keycloak sends a green flag to the backend, which then processes the request and returns the appropriate response to the frontend.

Overview of Authentication Architecture

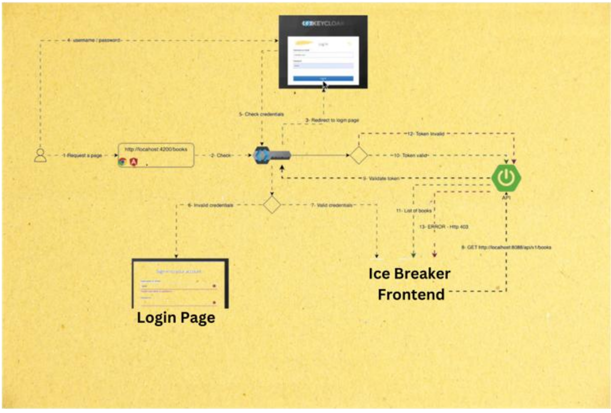


Figure 11: System Architecture for Authentication and Authorization Using Keycloak

2.3.5 Search users

We aimed to enhance the user experience by allowing users to quickly find and connect with other online users. This feature was designed with the same principles of Object-

Oriented Programming (OOP) that guided the development of the "Available Users" overview, focusing on modularity, reusability, and efficiency.

The search functionality interacts with the backend to filter users based on the search query entered by the logged-in user.

When a user enters a name into the search bar, the frontend transmits a request to the backend, which processes the query and returns a list of users whose usernames match the search criteria.

```
@Query("SELECT p FROM PublicUser ProfileEntity p WHERE p.username LIKE %:username%
AND p.keycloakUser Id <> :currentUser Id")
```

Importantly, the backend not only returns users who are currently online and who have not previously engaged in a chat with the logged-in user, it returns every user who has been saved in the DB, ensuring that the results are relevant and actionable.

To ensure optimal performance, we implemented a mechanism to avoid unnecessary data retrieval. The backend utilizes a dedicated search method that leverages the existing user data structure**.** This ensures that only the relevant user information is transmitted. This approach minimizes the load on the server and enhances the responsiveness of the application.

The search results are displayed using a similar Data Transfer Object (DTO) structure as the "Available Users" feature.

The search results are dynamically rendered in a user-friendly card format, similar to the available users' display. Each card presents the user's profile picture, username, and status, allowing users to quickly identify potential contacts.

2.3.6 “Available” user overview

For the See All Users component, we have been following the five main principles of Object-Oriented Programming (OOP). The key features we adhered to are modularity, non-repetitiveness, and extensibility, while ensuring that modifications are minimized.

The backend is responsible for providing a list of all users, excluding the currently logged-in user. Specifically, it returns only those online users whom the logged-in user has not yet chatted with. A crucial principle we followed was not requesting any unnecessary data from the backend, ensuring efficiency and optimized performance.

To achieve this, we used a Data Transfer Object (DTO) structure for displaying users:

AvailableUserDTO= [id; username ; profilePhoto ; bio; status]

This DTO contains only the essential information needed for the frontend, avoiding unnecessary data transmission. Then, each user’s details are compiled into a card component, which is then dynamically rendered on the screen using a loop.

We also implemented pagination by fetching and displaying a maximum of 10 users at a time. This allows users to navigate through the list of available users easily, using "Previous" and "Next" buttons to view the next or previous set of available users. This approach enhances the user experience by preventing overwhelming amounts of information from being displayed at once and allows for smoother navigation through the available user profiles.

This approach eliminates the need to manually create individual user cards, saving both development time and effort while maintaining scalability.

3. Architecture

3.1 General overview

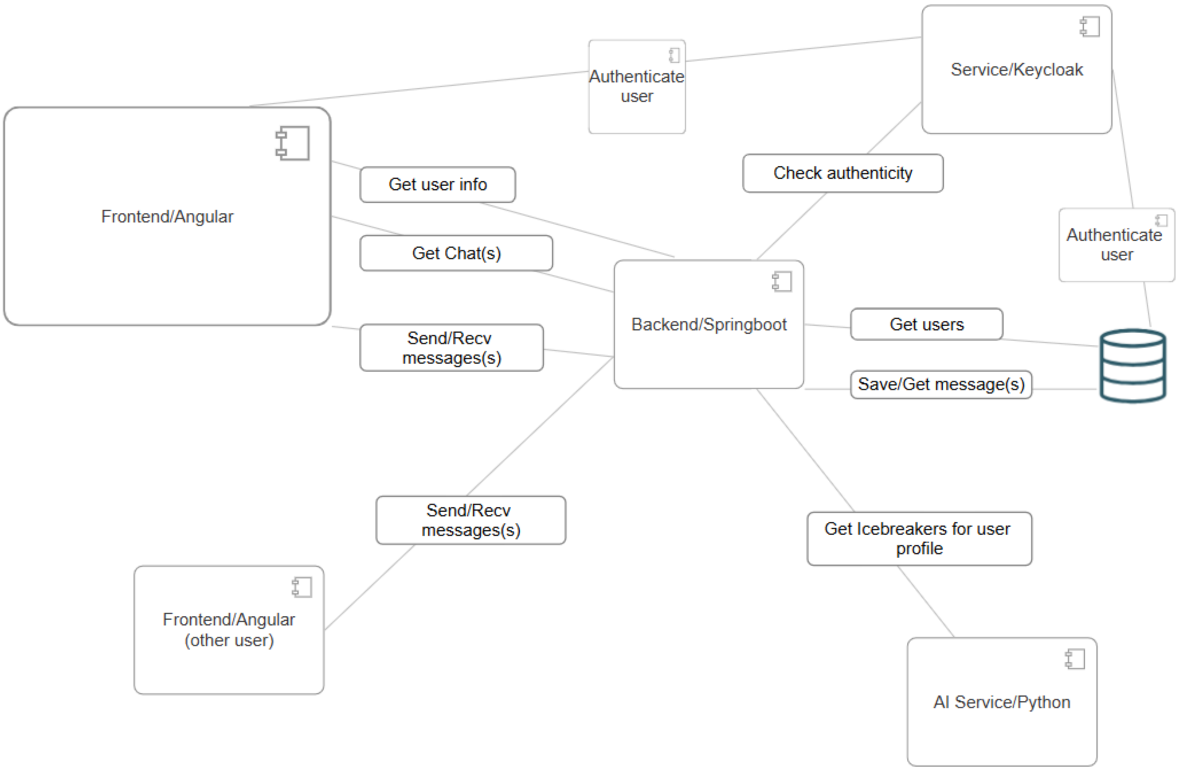


Figure 12: General Overview of our Architecture

Here the general ideas with the core components are illustrated. (Most of the shown steps are an over-simplification of the actual processes). Furthermore, the diagram illustrates a main user (Frontend/Angular) using our systems and all components and connections relevant for the general use cases, the other user is “only there” to illustrate the chat functionality between different end users.

3.2 AI functionality

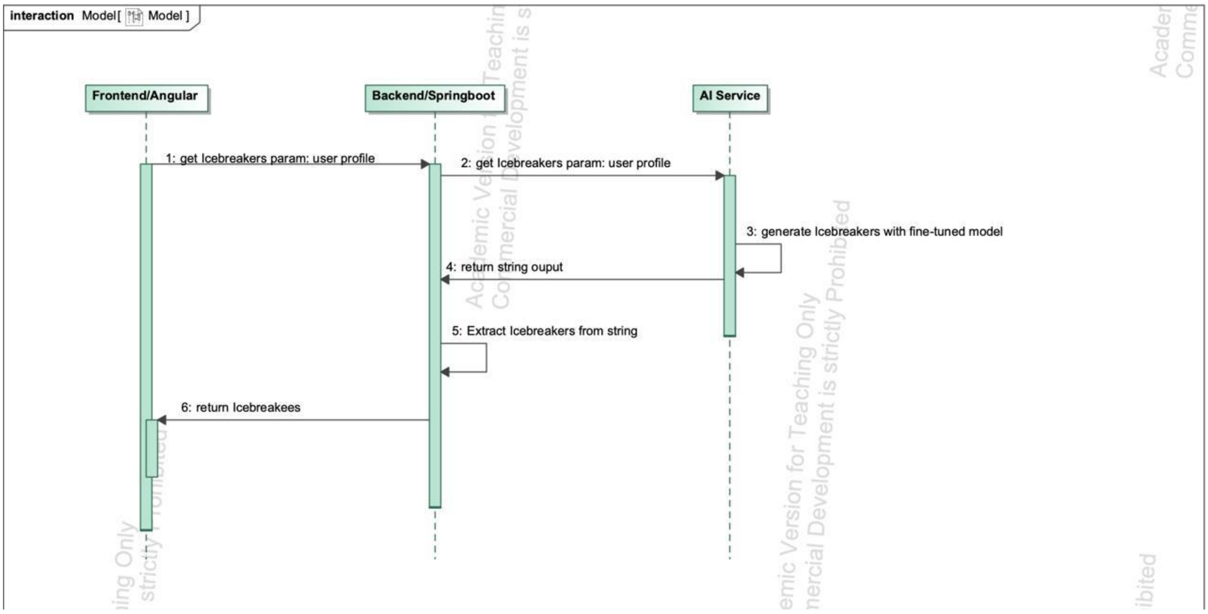

interaction Model[Model]
Frontend/Angular
Backend/Springboot
AI Service
1: get Icebreakers param: user profile
2: get Icebreakers param: user profile
3: generate Icebreakers with fine-tuned model
4: return string ouput
5: Extract Icebreakers from string
6: return Icebreakees

Figure 13: This sequence diagram illustrates how the different components interact when new Icebreakers for a profile shall be generated.

3.3 End-to-end encryption

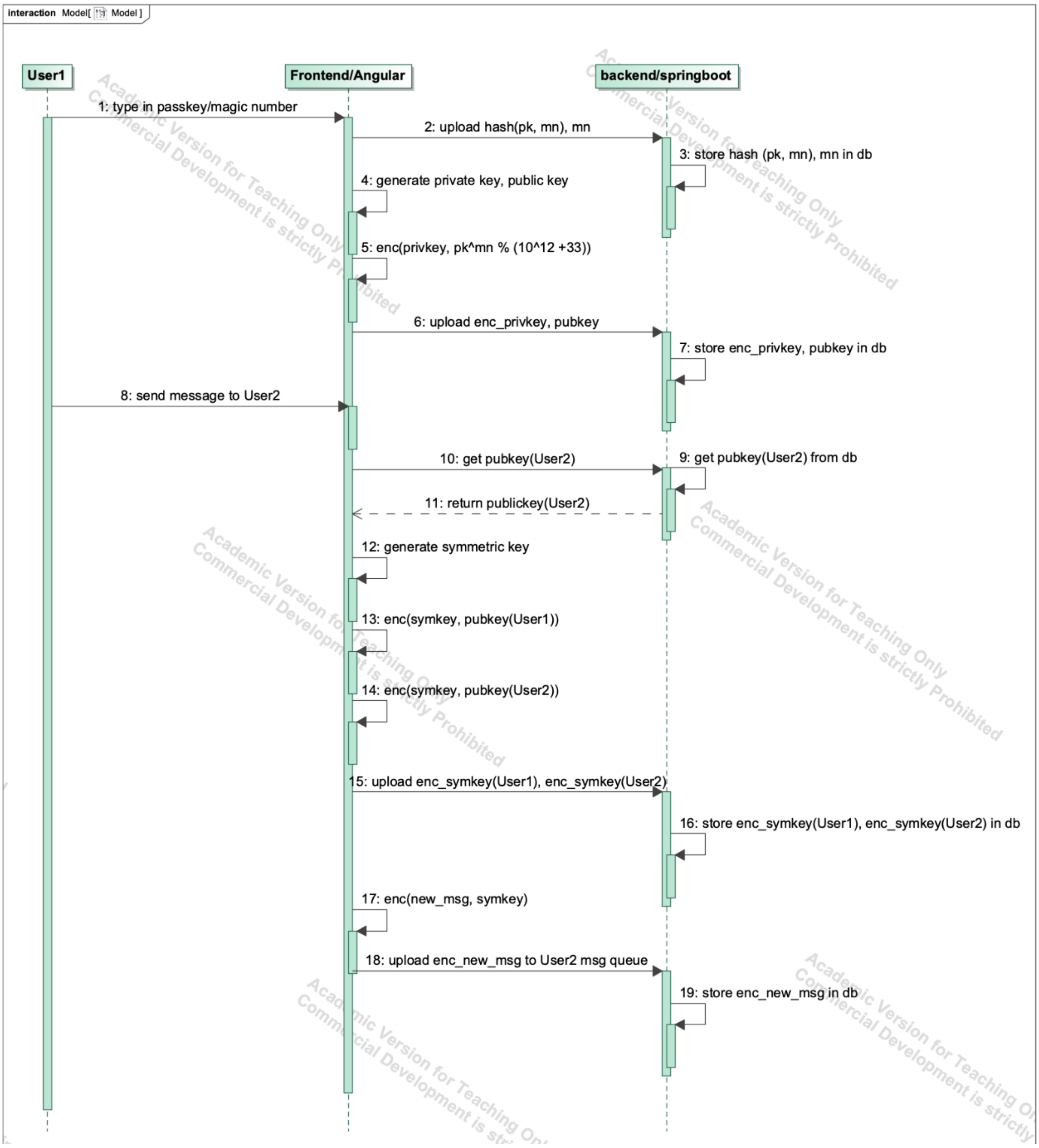

interaction Model[Model]
User1
Frontend/Angular
backend/springboot
1: type in passkey/magic number
2: upload hash(pk, mn), mn
3: store hash (pk, mn), mn in db
4: generate private key, public key
5: enc(privkey, pk^mn % (10^12 +33))
6: upload enc_privkey, pubkey
7: store enc_privkey, pubkey in db
8: send message to User2
9: get pubkey(User2) from db
10: get pubkey(User2)
11: return publickey(User2)
12: generate symmetric key
13: enc(symkey, pubkey(User1))
14: enc(symkey, pubkey(User2))
15: upload enc_symkey(User1), enc_symkey(User2)
16: store enc_symkey(User1), enc_symkey(User2) in db
17: enc(new_msg, symkey)
18: upload enc_new_msg to User2 msg queue
19: store enc_new_msg in db

Figure 14: Sequence Diagram for E2EE(a)

This sequence diagram illustrates how end-to-end encryption is realized, through illustrating the flow when a new user registers and sends a new message to another (existing) user. Important abbreviations: hash – hashing algorithm, pk – passkey, mn – magic number, enc – encryption algorithm, enc_variable - encrypted variable, pubkey – public key, privkey – private key, symkey – symmetric key, msg – message, db - database.

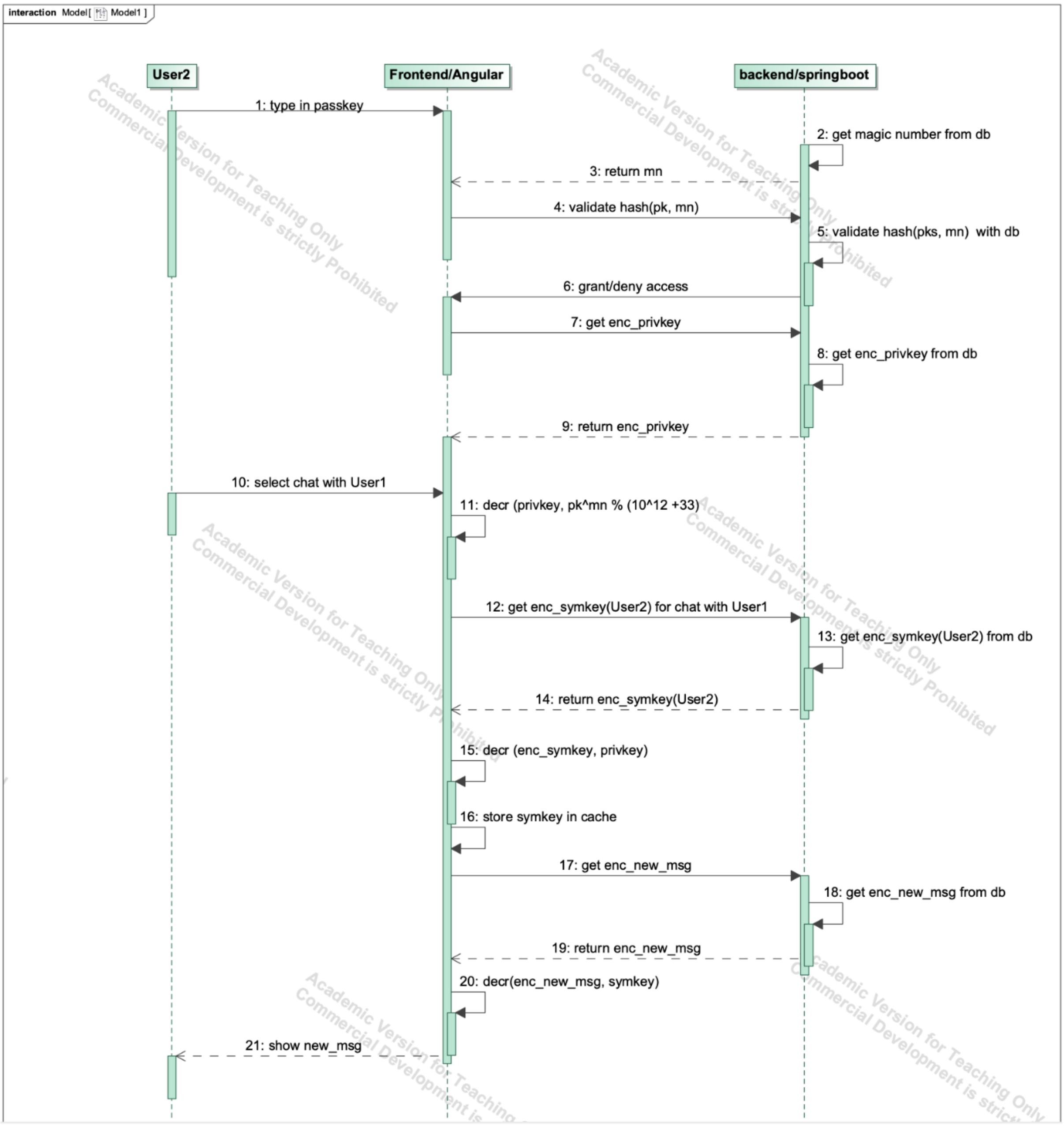

interaction Model[Model1]
User2
Frontend/Angular
backend/springboot
1: type in passkey
2: get magic number from db
3: return mn
4: validate hash(pk, mn)
5: validate hash(pks, mn) with db
6: grant/deny access
7: get enc_privkey
8: get enc_privkey from db
9: return enc_privkey
10: select chat with User1
11: decr (privkey, pk^mn % (10^12 +33)
12: get enc_symkey(User2) for chat with User1
13: get enc_symkey(User2) from db
14: return enc_symkey(User2)
15: decr (enc_symkey, privkey)
16: store symkey in cache
17: get enc_new_msg
18: get enc_new_msg from db
19: return enc_new_msg
20: decr(enc_new_msg, symkey)
21: show new_msg

Figure 15: Sequence Diagram for E2EE(b)

This sequence diagram illustrates how the other user logs in and receives the new message from the new user. Important abbreviations: hash – hashing algorithm, pk – passkey, mn – magic number, decr – decryption algorithm, enc_variable - encrypted variable, pubkey – public key, privkey – private key, symkey – symmetric key, msg – message, db - database.

3.4 Chat functionality

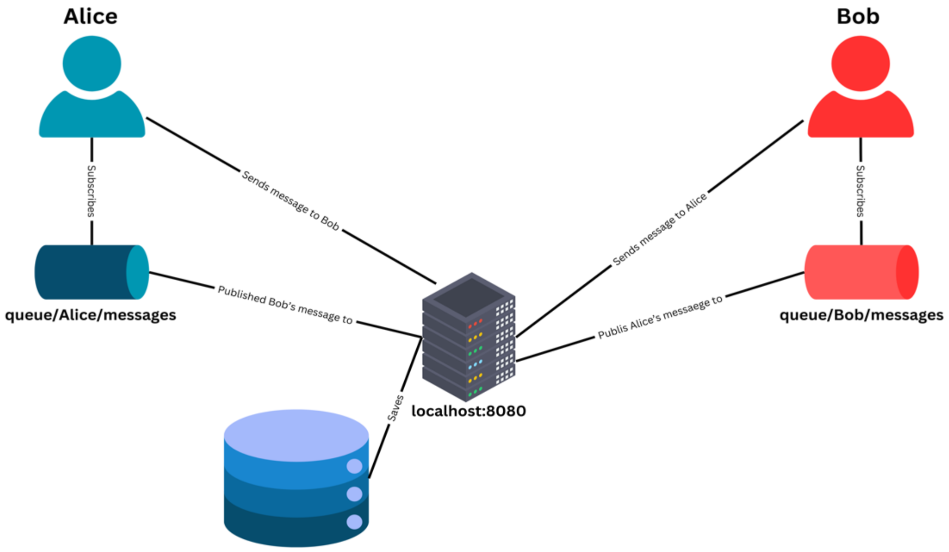


Figure 16: Real Time Chatting - Architecture

4. Challenges we have faced

4.1 AI feature

Data gathering

Initially we wanted to write 20 Profiles + Icebreakers ourselves and let ChatGPT generate 5000 more, but we soon noticed that ChatGPT was not able to do this at all (the quality of the generated examples was extremely bad, ChatGPT essentially wrote an algorithm to generate them, meaning each profile and Icebreaker was nearly the same). We then created 50 Profiles + Icebreaker ourselves and through countless tries were able to let ChatGPT create 150-200 more sufficient ones. This meant that we had an extremely much smaller dataset to work with, furthermore, the provided examples were extremely similar in how their structure was. An example, profile: “Hey, I’m Maya, I like singing, learning, languages, and eating sweets. I’m currently studying computer science.” Each profile starts with a hello and the name, then hobbies and often the current occupation(s) is/are listed. An Icebreaker for this profile would be: “Singing is really nice. What kind of songs do you really like singing the most?” Each Icebreaker has two sentences, one sentence reaffirming the interest of the person and the other asking a question about it. We believe such a structure makes the most sense for first messages but also due to limitation in the number of datasets that we have, all Icebreakers follow this structure. Thus, the generated Icebreakers do the same, the real problem comes from when the structure of the profile is deviated from. We first trained the gpt2-small model, here minor changes in the profile structure often meant illogical Icebreakers. With the gpt2-medium model this is a little better, even if the provided profile is significantly different from the learnt structure, the model is still able to generate mostly “good” Icebreakers. But the lack of datasets and the fact that all datasets follow a similar structure for the profiles still means, that when giving the model a profile that completely deviates from the learned structure the generated Icebreakers cannot be considered “good”. Here we want to demonstrate an example for a generated Icebreaker with the fine-tuned gpt2-medium model. Profile:” Hey, I’m Lucas. I love playing guitar, coding games, and studying astrophysics.” One of the generated Icebreakers: “Astrophysics is mind-blowing! Do you have a favorite celestial phenomenon?”

Training the model

Another issue that we faced was that none of our GPUs had enough VRAM to train the small (117 million parameters) or the medium (345 million parameters) model. Thus, we used Google Collab and bought 100 computational units, with these we could select a GPU which had approximately 40 GB of VRAM. With this we could train the model, after assessing that we had sufficient results we could download the weights and use them in our application.

4.2 Compatibility, Versioning and Dependencies

Our team has faced significant challenges because we worked on different machines. When Utkarsh for example set up the login functionalities and parts of the frontend, we could not immediately run the program as we for example had older or newer versions of node.js, java, etc. Or we did not have it at all and had to download the correct version. In this regard dependencies were a huge problem as well, often they needed to be downloaded separately, as here too versions mattered this could become a lengthy process of trial and error. We tried to mitigate this problem using docker, both keycloak and PostgreSQL would for example run in docker. Docker tries to solve the described problem, by making the creator of a program able to include all dependencies, libraries, etc. in so called docker images. However, due to the project's complexity and numerous components, many dependency issues had to be resolved manually. Especially, when trying to run our program on a separate machine to test it, we again noticed how much versions of node.js, java, etc. mattered. We changed the “README” file in accordance, so that it still works, but we could not guarantee with this setup that our program would run on all devices. Lastly, our AI feature proved to be another major issue, when executing it throughout development, we used PyCharm, this proved to be fatal for one of our teammates, as executing any program on PyCharm was not possible on his machine (The problem laid with python). Unfortunately, until the end of our project he could not make python work on his machine and therefore, he was not able to execute the AI Api.

4.3 Real Time Chat Application

With all the knowledge we gathered up to the 4th semester, we felt fully equipped to understand how a real-time chat application could be designed—at least in theory. However, we underestimated the effort required to transition from theory to reality.

During the early planning phase, one of the biggest challenges was evaluating individual frameworks and understanding how certain components, such as message queues, could be implemented. For instance, we struggled with designing a common queue structure that wouldn't require storing user details on the server.

Additionally, many of the YouTube tutorials we found were outdated and incompatible with the latest versions of the technologies we were using. Syntax changes and missing documentation for our specific framework further complicated the process. Since there were no up-to-date video tutorials available, we had to rely entirely on official documentation. This turned out to be a significant challenge, as the necessary information was scattered across multiple web pages, requiring extensive time and effort to piece everything together. Apart from that, connecting the backend and frontend was another major challenge. A direct POST request was not a viable option in this scenario, which required substantial syntax modifications. Adapting to these changes and ensuring proper communication between both ends of the application took considerable effort.

Despite these challenges, overcoming them gave us a new level of confidence. We now understand how to approach larger problems and find solutions even when resources like YouTube tutorials are not available. This experience has strengthened our ability to tackle complex technical issues and build solutions from the ground up.

4.4 UI Designing

Since our UI knowledge was quite limited, we only designed basic user interfaces for essential components. However, we lacked a well-defined UI structure, particularly for features like Icebreakers and the chat application. As a result, the developer had to spend additional time first visualizing the concept and then implementing it in code. Moreover, since the developer was not a professional UI/UX designer, we acknowledge that the visual experience of the application may not be as polished as it could be. However, if we decide to scale the application in the future, we plan to redesign the UI to ensure a more structured, user-friendly, and visually appealing experience.

4.5 Bugs

Despite our best efforts, we encountered some unexpected bugs that we were unable to resolve within the given time constraints.

The first issue involves our real-time chat window, where messages sometimes appear twice. Although the system is correctly subscribed to the message queue, a sender's message occasionally gets replicated in the chat window. We had plans to address this but resolving it would have required significant changes across both the frontend and backend. Given the time limitations, we decided to defer this fix for now, as the application still serves its core purpose effectively.

Another issue we faced was that refreshing the chat screen resulted in a blank window. To resolve this, users currently need to switch to a different screen and then return to the chat screen. Although we implemented code to handle this scenario, it did not work as expected. Again, due to time constraints, we were unable to conduct detailed line-by-line debugging.

While these issues exist, the application remains functional, and we plan to address these bugs in future iterations.

4.6 HTTPS Connection

Because we prioritized the security of our application, we implemented an end-to-end encryption. Additionally, we chose to secure our HTTP connections even in the local development environment by introducing a certificate from a certification authority, enabling us to use HTTPS instead of HTTP.

To achieve this, we generated a self-signed certificate on our local machine. However, implementing this required significant backend configuration changes, as it was initially designed to handle only HTTP requests. Since HTTPS is typically managed by the hosting service during deployment, we decided that fully implementing this configuration at this stage was unnecessary.

.

5. Who has done what

Utkarsh Fulara was responsible for the login and authentication, for the chat overview component and for the real time chatting feature. Bilal Alali was responsible for the search feature, the profile overview component and for creating data for our AI feature. Andrej Sum-Shik was responsible for conceptualizing the AI, training the AI and integrating the AI into the Website. Furthermore, he was also responsible for the end-to-end encryption of the messages. All members worked on creating architectures for the application. Especially conceptualizing the database where login and chat data is stored.

6. Conclusion

This was our favourite project of our young career so far, mainly because we were given the freedom to create what we want and the fact that we could work on one of the most important topics in computer science of this day. Through this project we were able to understand neural networks which are the basis for all modern AI. Furthermore, we were able to work with and understand the complex models of today's age which extend the classic neural networks, namely generative AI models like ChatGPT. We had a lot of fun understanding how AIs like ChatGPT work and how we can fine-tune these models to our demands. Furthermore, we were able to sophisticate and enlarge our knowledge of creating complex websites, which incorporate a java driven backend (in our case springboot), a JavaScript driven frontend (in our case angular) and a database structure (in our case PostgreSQL). Especially conceptualizing and implementing the chat functionality demanded a lot of our knowledge acquired in courses like: Distributed Systems or Computer Networks. Thus, we could bring a lot of the learned concepts into practice. The same can be said for the end-to-end encryption. Here the knowledge acquired in the IT Security lectures was of vital importance to realize our ideas. As the mentioned courses were among some of our favourites in our journey in this university, we were able to sophisticate our interests and get a first idea of what to do after acquiring our bachelor's degree. (If all goes well all of us are finishing this year or at the beginning of next year). In conclusion, on behalf of the entire team: Utkarsh Fulara, Bilal Alali and Andrej Sum-Shik we would like to thank Prof. Dr. -Ing. Markus Miettinen for giving us this wonderful opportunity. We are very satisfied with this course and are looking forward to the presentation where we can present our work to all the other students.