WYZSZA SZKOLA BANKOWA W POZNANIU

BACHELOR THESIS

The Mango Messenger

Authors:
Petro Kolosov, Serhii Holishevskyi, Illia Zubachov, Arslanbek Temirbekov

Supervisor:
Dr. Szymon Murawski

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Computer Science

in the

Wyzsza Szkola Bankowa w Poznaniu
Department of Computer Science

November 2, 2021

Partner Details

Mentor's details

First name and surname	Szymon Murawski
Degree	
Date and signature	

Team members' details

First name and surname	Petro Kolosov
Course of study	
Type of study program	
Date and signature	

First name and surname	Serhii Holishevskyi
Course of study	
Type of study program	
Date and signature	

First name and surname	Illia Zubachov
Course of study	
Type of study program	
Date and signature	

First name and surname	Arslanbek Temirbekov
Course of study	
Type of study program	
Date and signature	

Contents

Chapter 1

Introduction

Nowadays, instant messaging systems achieve a great success and became the main mean of communication between people via an internet. Thanks to the simplicity and quickness of the message exchanging more and more people over the world start to use instant messengers on daily basis. However, such a great attention forces us to discuss another aspect of these systems, an aspect of the information security and user privacy. The main aim of this thesis is to design and implement an instant messaging system that copes with the required functionalities and satisfies the defined security requirements. More precisely, the following steps

1. Chapter 1 provides the functional and non-functional requirements for instant messaging system.
2. Chapter 2 eliminates security and user privacy vulnerabilities of the instant messaging system by means of threat modelling.
3. In chapter 3 we design and discuss instant messaging system, going into details
    - To propose web service architecture that fits the requirements.
    - To propose an authorization mechanism that fits the requirements.
    - To propose mitigations for security and user privacy vulnerabilities discussed in chapter 2.
    - To design database structure.
    - To propose planned technologies to be used during implementation of the system.
4. To discuss and apply E2E Encryption to the system.
5. To design user interface that fits the designated functional requirements.
6. To implement following modules
    - **Web Service (API).** Application Programming Interface that allows developers to create their own clients. Web service to be implemented using latest for the moment of writing of this thesis .NET 5 platform.

- **Web Client** — Web client of the Mango Messenger. Web client to be implemented using Angular front-end framework with TypeScript programming language.
- **Mobile Client** — Mobile client of the Mango Messenger for target platforms: Android, IOS.
- **Desktop Client** — Desktop client of the Mango Messenger for target platforms Windows, Linux, MacOS. Desktop client to be implemented by means of existing web client and ElectronJS framework.

System Requirements

In previous sections we have briefly discussed Instant Messaging System, mainly from security and user privacy aspects. Prior to software module implementation, it is essentially important to define the functionality module will obtain. In this section we discuss functional and non-functional requirements of secure instant messaging system from customer's prospective.

Generally, there are three forms of software product requirements: business, functional, and non-functional. Business requirements [Dilworth and Kochhar, 2007] typically answer how the product will address the needs of your company and its users. They also reveal the business model of the app and what problems it can solve. Functional requirements [Malan, Bredemeyer, et al., 2001] are about functionalities that will be implemented in the application. Non-functional requirements [Chung et al., 2012] describe how these functionalities will be implemented.

1.1 Functional Requirements

Mostly common and simple way to define software product's functional requirements is User Stories. User stories [Cohn, 2004] should be understandable both to developers and to you as the client, and should be written in simple words. The most popular way of writing a user story is with the following formula:

```
"As a <user type>, I want <goal> so that <reason>."
```

Now, let's group the main features of the application as follows

- Registration
- Authentication
- Managing contacts
- Sending messages and media to individuals
- Creating and managing groups
- Sending messages and media to groups

- Viewing messages history
- Managing profile settings

1.1.1 Registration user stories

- As an unregistered user, I want to tap “Register” so that I see the registration form and register myself.
- As an unregistered user, I want to use my phone number to register so that my account is tied to my phone number.
- As an unregistered user, I want to use my e-mail address to register so that my account is tied to my e-mail address.
- As an unregistered user, I want to add a display name during registration so that other users can find me using it.
- As an unregistered user, I want to choose how to receive the registration confirmation via SMS or e-mail so that notification is sent to me via SMS or e-email.
- As an unregistered user, I want to receive the registration confirmation via SMS or Email so that I can activate my account.
- As a registered user, I want to confirm my email address so that I get confirmation link via email I provided.
- As a registered user, I want to confirm my phone number so that I use specified form to do it.

1.1.2 Authentication user stories

- As a registered user, I want to authenticate myself using both combinations email-password and phone-password so that I use the specified form with two inputs.
- As a registered user, I want to restore my password if I forget it so that I use specified form and restore my password.
- As an authenticated user, I want my session on each device to least 7 days so that after 7 days of inactivity device will be logged out automatically.

1.1.3 Managing contacts user stories

- As an authorized user, I want to see my contact list so that there is a list of users who are my contacts.
- As an authorized user, I want to search users so that I write user display name or phone number of e-mail address to specified input, click "Search user" button and see results.
- As an authorized user, I want to add other user to my contact list so that I click "Add contact" button on user profile and add him to my contact list.
- As an authorized user, I want user search input to accept empty or whitespace queries so that all users displayed as search result.
- As an authorized user, I want to remove the user from my contact list so that I click "Remove contact" button on user profile and remove him from my contact list.
- As an authorized user, I want to navigate to private chat with the user from my contact list so that I click "Message" button at user profile and get navigated to the private chat with him.

1.1.4 Sending messages and media to individuals user stories

- As an authorized user, I want to send a text message so that another user sees my message.
- As an authorized user, I want to add an attachment to the message so that another user sees the message with attachment.
- As an authorized user, I want to add an emoji to the message so that another user sees the message with emoji.
- As an authorized user, I want to tap "Edit" on my message so that message I edited is changed immediately in the chat.
- As an authorized user, I want to tap "Delete" on my message so that message immediately disappears from the chat.
- As an authorized user, I want to share secret messages with users from my contact list so that our are messages encrypted for anyone else including system administrators.
- As an authorized user, I want each new message in private chats I participate to be displayed immediately in real-time so that I do not reload page.

1.1.5 Creating and managing groups user stories

- As a registered user, I want to tap "Create channel" so that I create a new channel of the one of the types: Private channel, Public channel, Readonly channel.
- As a registered user, I want to tap "Start direct chat" so that I create a new direct chat with specified user.
- As a registered user, I want to tap "Start secret chat" so that I create a new secret chat with specified user.
- As a registered user, I want to join public groups so that I click button "Join group" to join the group.
- As a registered user, I want to tap "Archive" so that I archive the specified chat or channel.
- As a registered user, I want to tap "Un-archive" so that I un-archive the specified archived chat or channel.
- As a registered user, I want my secret chats to be device-specific so that I can see a secret chat only on the device that I used to start this chat.

1.1.6 Sending messages and media to groups user stories

- As an authorized user, I want to send a text message so that other members of the group see the message I sent.
- As an authorized user, I want to add an attachment to the message so that other members of the group see the message with attachment I sent.
- As an authorized user, I want to add an emoji to the message so that other members of the group see the message with emoji I sent.
- As an authorized user, I want to tap "Edit" on my message so that other members of the group see the message I edited.
- As an authorized user, I want to tap "Delete" on my message so that my message is deleted for all members of the group.
- As an authorized user, I want to search public groups by title so that I enter display name to specified field, click button "Search chats" and see results.
- As an authorized user, I want each new message in groups I participate to be displayed immediately in real-time so that I do not reload page.

1.1.7 Viewing messages history user stories

- As an authorized user, I want to view a message history of particular chat or group so that I see a list of my active chats on the UI.
- As an authorized user, I want to search messages in particular chat so that I see the results in messages window of the chat.

1.1.8 Managing profile settings user stories

- As an authorized user, I want to update my personal information in profile settings so that other users my updated personal information.
- As an authorized user, I want to update my social network links in profile settings so that other users my updated social media.
- As an authorized user, I want to change my profile picture so that all other users will see updated one.
- As an authorized user, I want reset password, so that my password will change.
- As an authorized user, I want to tap "Logout" button so that current device will be logged out from the system.
- As an authorized user, I want to tap "Logout all" button so that all my authorized devices will be logged out from the system.

1.2 Non-Functional Requirements

- **NFR01.** Graphic user interface of the system should be well organized. To fulfill this requirement, we follow an ISO 9241–161:2010 (en) Ergonomics of human-system interaction standard [ISO and STANDARD, 2010].
- **NFR02.** The system should have well performance, which meant to respond it at least 1 second. User should have a device with at least 6 GB RAM and CPU with 1.8 GHZ, 100 Mbps internet connection. Server must have the following hardware: Intel 2.4 GHz 8 Cores server processor, 64GB DDR4 (4x16GB) memory, NVME or SAS server disk with a minimum capacity of 1.6 TB.
- **NFR03.** The unique, unambiguous identifier of users in the system is the username. It is set in the profile settings.
- **NFR04.** The UI must be well displayed with the following browsers, in the versions current at the date of receipt of the system or, depending on technical possibilities, with the latest versions that support correct operation of the system:

  - Google Chrome 72.0.36.
  - Mozilla Firefox 64.0.2.
  - Microsoft Edge 17.17134.

- **NFR05.** The system shall force users to use passwords with a minimum length of 8 characters and using at least one capital letter and one number and one special symbol.

- **NFR06.** The UI must be compatible to use on mobile device screens with a minimum width of 600 pixels.

- **NFR07.** The UI must be compatible to use on desktop or laptop device screens with a minimum display width of 1024 pixels.

Chapter 2

Security and User Privacy Vulnerabilities of Instant Messaging System

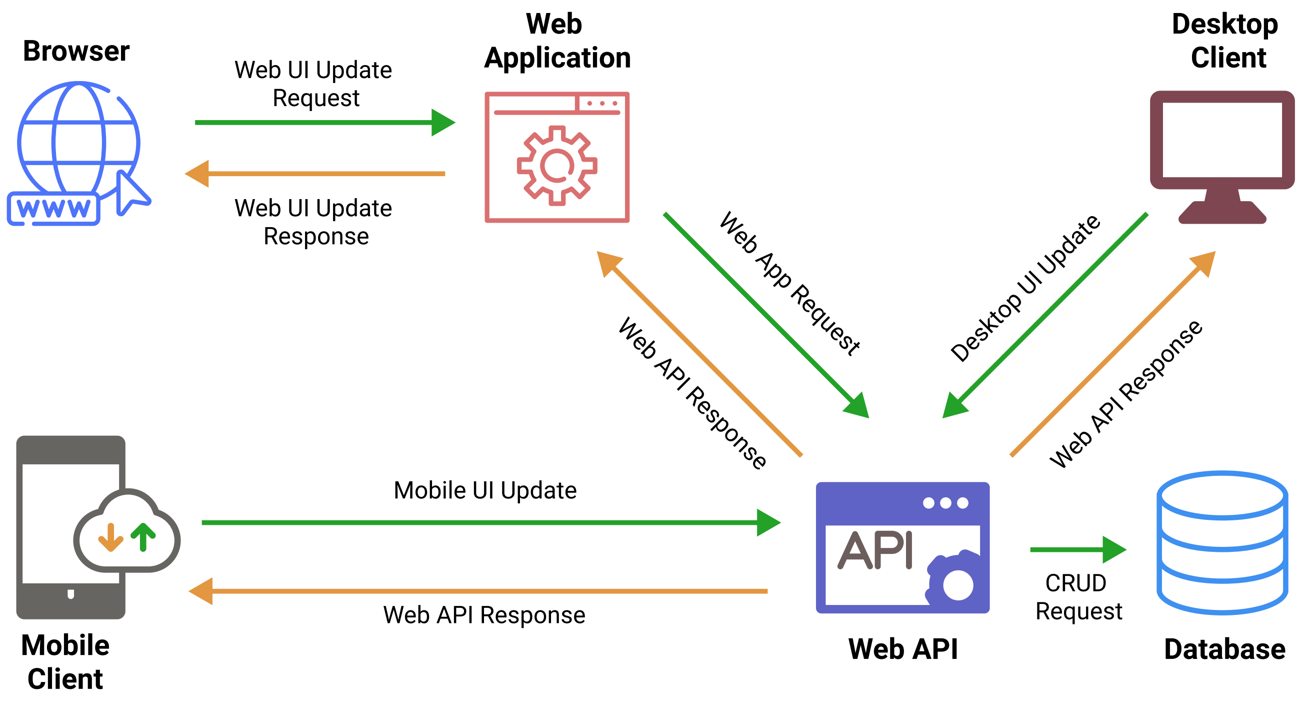


FIGURE 2.1: Database diagram.

1. **Browser UI Update Request**

   - **Treat 1.1.** An adversary can perform action on behalf of other user due to lack of controls against cross domain requests.
   - **Treat 1.2.** An adversary may bypass critical steps or perform actions on behalf of other users (victims) due to improper validation logic.

- **Treat 1.3.** An adversary can reverse weakly encrypted or hashed content.
- **Treat 1.4.** An adversary may gain access to sensitive data from log files.
- **Treat 1.5.** An adversary can spoof the target web application due to insecure TLS certificate configuration.
- **Treat 1.6.** An adversary can steal sensitive data like user credentials.
- **Treat 1.7.** An adversary can gain access to sensitive data stored in Web App's config files.
- **Treat 1.8.** An adversary can gain access to sensitive data by performing SQL injection through Web App.
- **Treat 1.9.** An attacker steals messages off the network and replays them in order to steal a user's session.
- **Treat 1.10.** An adversary can deface the target web application by injecting malicious code or uploading dangerous files.
- **Treat 1.11.** An adversary may spoof Desktop Web Browser (Chrome) and gain access to Web Application.
- **Treat 1.12.** An adversary can create a fake website and launch phishing attacks.
- **Treat 1.13.** Attackers can steal user session cookies due to insecure cookie attributes.
- **Treat 1.14.** An adversary can get access to a user's session due to insecure coding practices.
- **Treat 1.15.** An adversary can get access to a user's session due to improper logout and timeout.
- **Treat 1.16.** Attacker can deny the malicious act and remove the attack foot prints leading to repudiation issues.
- **Treat 1.17.** An adversary may gain access to sensitive data from uncleared browser cache.
- **Treat 1.18.** An adversary can gain access to sensitive information through error messages.
- **Treat 1.19.** An adversary can gain access to sensitive data by sniffing traffic to Web Application.
- **Treat 1.20.** An adversary can gain access to certain pages or the site as a whole.
- **Treat 1.21.** An adversary may gain access to unmasked sensitive data such as credit card numbers.

2. **Web App Request**

- **Treat 2.1.** An adversary may gain unauthorized access to Web API due to poor access control checks.
- **Treat 2.2.** An adversary can gain access to sensitive information from an API through error messages.
- **Treat 2.3.** An adversary can gain access to sensitive data by sniffing traffic to Web API.
- **Treat 2.4.** An adversary can gain access to sensitive data stored in Web API's config files.
- **Treat 2.5.** Attacker can deny a malicious act on an API leading to repudiation issues.
- **Treat 2.6.** An adversary may spoof Mango Web Application and gain access to Web API.
- **Treat 2.7.** An adversary may inject malicious inputs into an API and affect downstream processes.
- **Treat 2.8.** An adversary can gain access to sensitive data by performing SQL injection through Web API.

3. **Web API Response**

   - **Treat 3.1.** An adversary can reverse weakly encrypted or hashed content.
   - **Treat 3.2.** An adversary may gain access to sensitive data from log files.
   - **Treat 3.3.** An adversary can gain access to sensitive information through error messages.
   - **Treat 3.4.** Attacker can deny the malicious act and remove the attack foot prints leading to repudiation issues.
   - **Treat 3.5.** An adversary can spoof the target web application due to insecure TLS certificate configuration.
   - **Treat 3.6.** An adversary can steal sensitive data like user credentials.
   - **Treat 3.7.** An adversary can create a fake website and launch phishing attacks.

4. **CRUD Request**

   - **Treat 4.1.** An adversary can gain unauthorized access to database due to loose authorization rules.
   - **Treat 4.2.** An adversary can gain access to sensitive PII or HBI data in database.
   - **Treat 4.3.** An adversary can gain access to sensitive data by performing SQL injection.

- **Treat 4.4.** An adversary can deny actions on database due to lack of auditing.
- **Treat 4.5.** An adversary can tamper critical database securables and deny the action.
- **Treat 4.6.** An adversary may leverage the lack of monitoring systems and trigger anomalous traffic to database.
- **Treat 4.7.** An adversary can gain unauthorized access to database due to lack of network access protection.

5. **Mobile UI Update**

   - **Treat 5.1.** An adversary can gain access to sensitive data by performing SQL injection through Web API.
   - **Treat 5.2.** An adversary can reverse engineer and tamper binaries.
   - **Treat 5.3.** An adversary may inject malicious inputs into an API and affect downstream processes.
   - **Treat 5.4.** An adversary may spoof Mobile App (IOS, Android) and gain access to Web API.
   - **Treat 5.5.** An adversary obtains refresh or access tokens from Mobile App (IOS, Android) and uses them to obtain access to the Mango Web API.
   - **Treat 5.6.** Attacker can deny a malicious act on an API leading to repudiation issues.
   - **Treat 5.7.** An adversary can gain access to sensitive data stored in Web API's config files.
   - **Treat 5.8.** An adversary can gain sensitive data from mobile device.
   - **Treat 5.9.** An adversary can gain access to sensitive data by sniffing traffic to Web API.
   - **Treat 5.10.** An adversary can gain access to sensitive data by sniffing traffic from Mobile client.
   - **Treat 5.11.** An adversary can gain access to sensitive information from an API through error messages.
   - **Treat 5.12.** An adversary may gain unauthorized access to Web API due to poor access control checks.
   - **Treat 5.13.** An adversary may jail break into a mobile device and gain elevated privilege.

Chapter 3

System Design

3.1 Web Service Architecture

3.1.1 Motivation

Implementing the instant messenger system, we consider applying a well-known N-tier Monolithic Architecture [Bucchiarone et al., 2018], which provides a time-proven model that allows software developers to create flexible and reusable applications.

However, during the implementation of monolith it is very important to avoid the cases of crucial over-engineering of the system. For the developers, it is a vital point to follow the KISS [Alwin and Beattie, 2016] and YAGNI [Da Silva et al., 2018] software development principles in order not to reach thousands lines of code in a single class.

One would suggest to use nowadays popular Microservices Architecture, thinking about scalability [Brataas and Hughes, 2004], an ability of the system to handle large numbers of users distributed over geographically large areas without notably affecting the overall performance of the system. However, the effect of Microservices is being felt only for quite large and complex systems, not the case of our yet simple application. Following plot demonstrates this relation

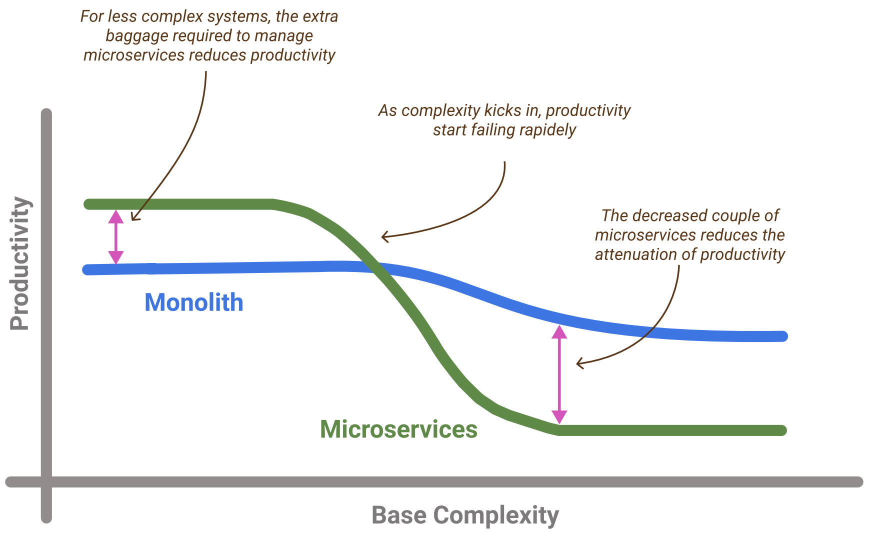


FIGURE 3.1: Relation between system complexity and architectures.
Source: Martin Fowler.

A layered architecture usually consists of Presentation layer, Business logic layer, Data access layer. By segregating the project into layers, developers reach the options to modify or add a specific layer without reworking the entire application.

- **Presentation Layer.** Graphic user interface or API gateway.
- **Application Logic.** Encapsulates the means of interaction with user, E-mail or SMS notifications.
- **Business Logic.** Directly handles previously validated request.
- **Data Access Layer.** Responsible for logging, database access among other services required to support Business Logic layer.

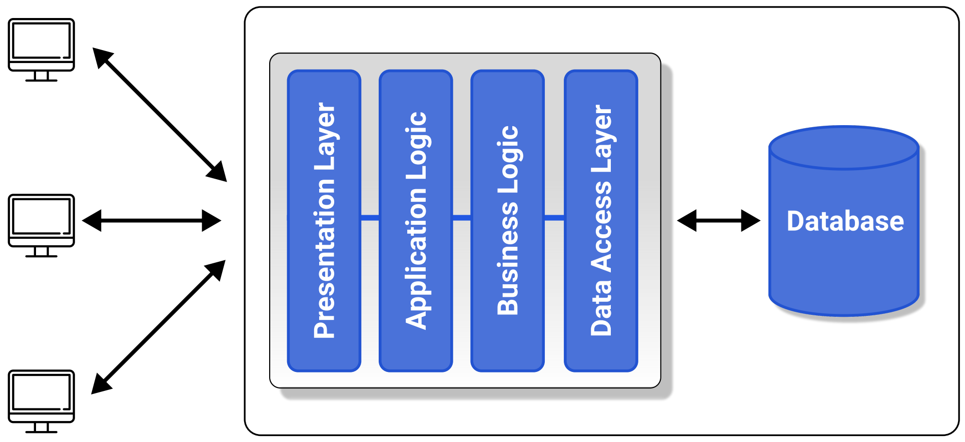


FIGURE 3.2: Monolithic architecture diagram.

3.1.2 Monolithic Architecture: Cons and Props

A monolith is built as a large system with a single code base and deployed as a single unit, usually behind a load balancer. Monoliths offer several advantages, particularly when it comes to operational overhead requirements. Here are some of those basic benefits:

- **Simplicity.** Monolithic architectures are simple to build and deploy. These applications can scale horizontally, by running several copies of the application behind a load balancer. With a single codebase, monolithic apps can easily handle cross-cutting concerns, such as logging, configuration management and performance monitoring. Another advantage associated with the simplicity of monolithic applications is easier deployment. When it comes to monolithic applications, you do not have to handle many deployments but just one.

- **Performance.** Components in a monolith typically share memory which is faster than service-to-service communications using IPC [Proctor, 1999] or other mechanisms.

- **Easier debugging and testing.** In contrast to the microservices, monolithic applications are much easier to debug and test. Since that monolithic application is a single indivisible unit the process of end-to-end testing is much faster.

- **Easier development.** As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

However, the drawback of monolithic architectures hides in their tight coupling. Over time, monolithic components and layers become tightly coupled and entangled, effecting management, scalability and continuous deployment. Another disadvantages of the monoliths include:

- **Understanding.** When a monolithic application's code base grows up, it becomes too complicated to understand. Obviously, huge code base of monolithic app is hard to manage therefore.
- **Reliability.** Entire application down may be caused by an error in every single component.
- **Updates.** Single and large code base causes the needs to redeploy an application on every single update.
- **Technology stack.** Technology stack of the monolithic app is limited by the technologies and providers used from the beginning of development. It makes technology stack changes to be expensive in terms of finances and time.
- **Scalability.** Application's components cannot be scaled independently, an entire application should be scaled.

3.1.3 Minimization of services coupling

As we see, monolith has its own disadvantages, like for instance: understanding the project structure, reliability concerns, technology stack limitations, scalability limitations. Obviously, some of these disadvantages cannot be mitigated. However, the complexity and coupling problem can be minimized applying certain approaches. Frequent violation of the single-responsibility principle of SOLID during implementing service components in business logics layer causes the over-complication of codebase over the time. The reason is that service components keep the huge number of methods in order to handle all possible CRUD requests to database, without any bounded context. Schematically it is as follows

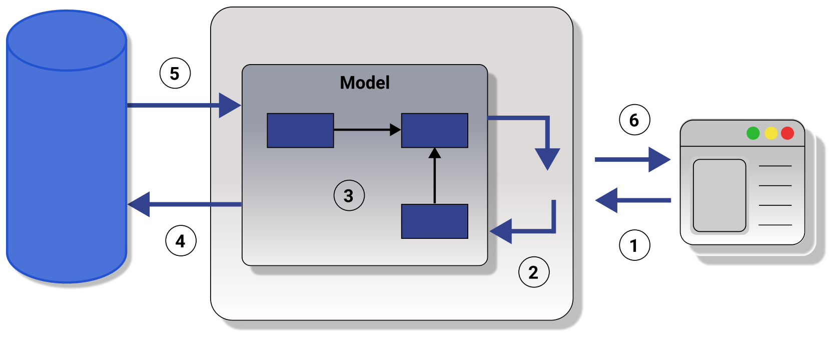


FIGURE 3.3: Service Conceptual diagram. Source: Martin Fowler.

Where the steps are

1. User makes a change in UI.
2. Change forwarded to model.
3. Model executes validation and business logic.
4. Model updates the database.
5. Model reads from database.
6. Service updates presentation from query model.

To minimize the natural disadvantages of the monolithic architecture like complexity and high tight coupling of the components we have to recall the design patterns [Rising, 1998]. In particular, the mediator pattern helps to decouple the components. Mediator – is a behavioral design pattern [Rasche et al., 2016] that allows the communication between two entities, such that entities doesn't know each other. Therefore, the program components depend only on a single mediator instance instead of being coupled to multiple of their colleagues. In context of .NET platform there are many implementations of the Mediator, the most widely known and used is the MediatR, which we use in our project.

Another mindset we are going to use in order to minimize complexity and coupling of monolith is Command-Query Responsibility Segregation (CQRS) principle. In brief, it stands that read (query) and write (command) requests should be segregated by their responsibilities. Using CQRS and Mediator together greatly simplifies the project structure and minimizes coupling between business logic layer components.

CQRS is a pattern that first described by Greg Young [Young, 2010] and its conceptual diagram as follows

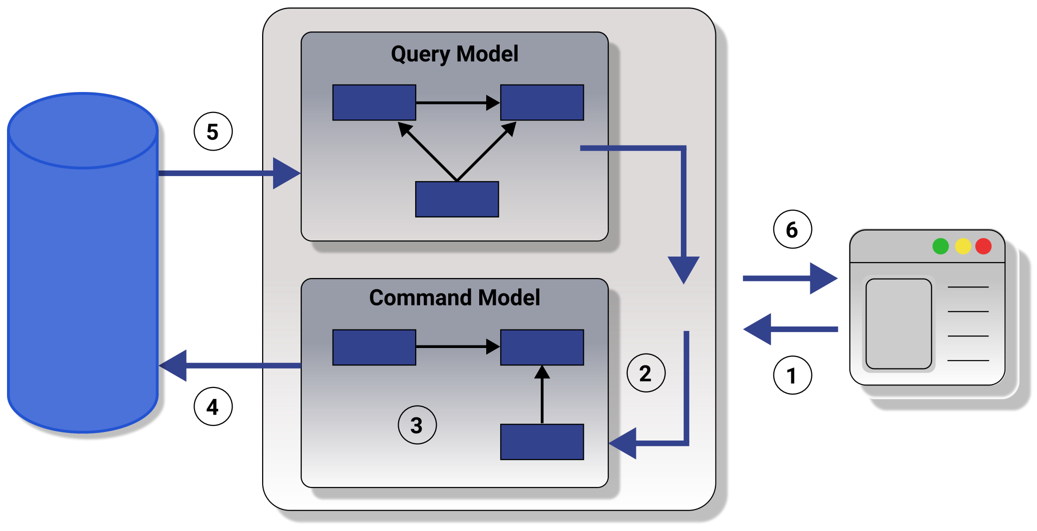


FIGURE 3.4: CQRS Conceptual diagram. Source: Martin Fowler.

1. User makes a change in UI.
2. Application routes information to command model.
3. Command model executes validation and business logic.
4. Command model updates the database.
5. Query model reads from database.
6. Query service update presentation from query model.

Despite these benefits, you should be very cautious about using CQRS. Many information systems fit well with the notion of an information base that is updated in the same way that it's read, adding CQRS to such a system can add significant complexity. I've certainly seen cases where it's made a significant drag on productivity, adding an unwarranted amount of risk to the project, even in the hands of a capable team. So while CQRS is a pattern that's good to have in the toolbox, beware that it is difficult to use well and you can easily chop off important bits if you mishandle it.

As a short conclusion, we may state that CQRS and Mediator pattern will not entirely solve the coupling problems the monolith, however will make project much more simplistic and intuitively understood. It is worth to keep is simple, even relatively simple project may grow to the sizes of universe without proper architectural solutions.

3.2 Authorization Mechanism

3.2.1 Motivation

In this section we describe the processes of Authentication and Authorization in the system. It is worth to remember the meaning of Authentication and Authorization definitions. Authentication – is the process of ascertaining that somebody really is who they claim to be [Burrows, Abadi, and Needham, 1989]. Authorization refers to rules that determine who is allowed to do what [Fagin, 1978]. For example, Adam may be authorized to create and delete databases, while Catherine is only authorized to read. The two concepts are completely orthogonal and independent, but both are central to security design, and the failure to get either one correct opens up the avenue to compromise. In terms of web apps, very crudely speaking, authentication is when you check login credentials to see if you recognize a user as logged in, and authorization is when you look up in your access control whether you allow the user to view, edit, delete or create content. Currently, there are two widely-known authentication methods, that are cookie authentication and JWT authentication. We discuss JWTs in next section.

3.2.2 JWT Tokens

What is JSON Web Token? JSON Web Token (JWT) is an open standard [Jones, Bradley, and Sakimura, 2015] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object [Jones, Campbell, and Mortimore, 2015]. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret with the HMAC [Wang et al., 2004] algorithm or a public/private key pair using RSA [Wiener, 1990] or ECDSA [Johnson, Menezes, and Vanstone, 2001].

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

When should you use JSON Web Tokens?

Here are some scenarios where JSON Web Tokens are useful:

- **Authorization.** This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.

- **Information Exchange.** JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed – for example, using public/private key pairs – you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

What is the JSON Web Token structure? In its compact form, JSON Web Tokens consist of three parts separated by dots, which are Header, Payload, Signature. Therefore, a JWT typically looks like

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I
kpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQ
ssw5c
```

Let's break down the different parts.

- **Header.** Typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA. For example,

  ```
  {
      "alg": "HS256",
      "typ": "JWT"
  }
  ```

  Then, this JSON is Base64Url encoded to form the first part of the JWT.

- **Payload.** The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

  – **Registered claims.** These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), and others. Notice that the claim names are only three characters long as JWT is meant to be compact.

  – **Public claims.** These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

  - **Private claims.** These are the custom claims created to share information between parties that agree on using them and are neither registered or public claims.

An example payload could be:

```
{
    "sub": "1234567890",
    "name": "John Doe",
    "admin": true
}
```

The payload is then Base64Url encoded to form the second part of the JSON Web Token. Do note that for signed tokens this information, though protected against tampering, is readable by anyone. Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

- **Signature.** To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

- **Putting all together.** The output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML. The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJqdGkiOiJmZDNjNjdjNS1jNmZmLTRhNWQtY
TE2Ni05OGVjZTFiNzc1MmIiLCJyb2xlIjoiVX
NlciIsIm5iZiI6MTYzMTU1MjQ5NiwiZXhwIjo
xNjMxNTUyNzk2LCJpYXQiOjE2MzE1NTI0OTYs
```

```
ImlzcyI6Imh0dHBzOi8vbWFuZ28tbWVzc2VuZ
2VyLWFwcC5oZXJva3VhcHAuY29tIiwiYXVkIj
oiaHR0cHM6Ly9tYW5nby1tZXNzZW5nZXItYXB
wLmhlcm9rdWFwcC5jb20vYXBpIn0.
locHt8ow1lFnGGZ_aFFvXI09dD4y1r594XQF2
-6YxCw
```

As to the projects concerns, we should handle multiple client applications, e.g desktop, web, mobile etc. Therefore, HTTP cookie authorization doesn't fit our requirements, however the JWT one surely passes.

3.2.3 JWT Authorization

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Since tokens are credentials, great care must be taken to prevent security issues. In general, you should not keep tokens longer than required. You also should not store sensitive session data in browser storage due to lack of security. Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

```
Authorization: Bearer <token>
```

This can be, in certain cases, a stateless authorization mechanism. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. If the JWT contains the necessary data, the need to query the database for certain operations may be reduced, though this may not always be the case. If the token is sent in the Authorization header, Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies. Generally, the workflow is as follows

1. User provides credentials in order to authenticate to the system.
2. Server verifies user's authentication, fetches the login and password in database.
3. If authentication is successful, server creates session then writes this session to the database, see table session in [REFERENCE_DATABASE_SCHEMA].
4. Server generates a pair of access token (JWT) and refresh token (GUID).
5. Server sends to client access token and refresh token.
6. Client saves the pair of access and refresh tokens.

7. User requests resource using received token passed to the request header.

8. The server check user's claims and proceeds or declines request.

The eighth point is the authorization. As a result, token stored on the client and used when it is necessary to authorize the requests. When a hacker tries to replace the data in the header or payload the token will become invalid, therefore the signature will not match the original values. So, the hacker hasn't any possibility to generate a new signature since that encryption secret key stored on the server. Access token (JWT) is used for request authorization and for storing the additional information about user like identifier, display name and others. Refresh Token (GUID) issued by server based on successful authentication results and used for get new access/refresh token pair. Also, it is worth to add a few basic rules about JWT secure usage [Degges, 2019]

- JWT should have a short lifetime, since it cannot be revoked.

- JWT should be used in a single time, e.g JWT per request.

Therefore, we consider access token's lifetime to be 5 minutes and refresh token's 7 days.

For each request client preliminarily checks access token's lifetime. If access token it expired, client sends request for updating a pair of access/refresh tokens. For more confidence, we can update tokens a few seconds earlier. That is, the case when the API receives an expired access token is practically excluded. However, we are able to consider the case of interception of the request on 401 http response code, that is unauthorized. The following diagram demonstrates the process of requesting the resource

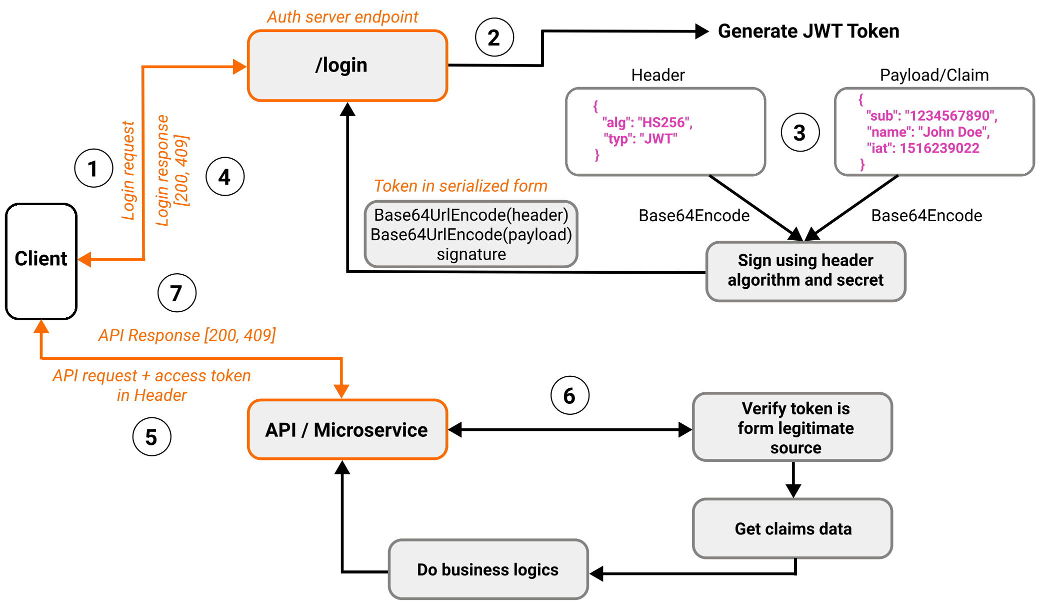


FIGURE 3.5: JWT Authentication concept diagram.

By steps, the process is

- **Step 1.** **Client** application sends **POST** authentication request to the **Auth server endpoint** provided user credentials in request body.
- **Step 2.** **Auth server endpoint** responses to the **Client** with the following HTTP response codes:
  - **409Conflict**: Invalid credentials.
  - **200Success**: Returns a pair of access and refresh tokens.
    - **Step 3.** Server generates a pair of access and refresh tokens
      - API fetches user data and claims.
      - Server creates new session instance in database.
      - Access token's **Header** Base64 encoded.
      - Access token's **Payload** with user claims is Base64 encoded.
      - Access token's **Signature** is generated using encoded token's Header and Payload signed by means of the algorithm (HMAC-SHA256) from the header and secret:

```
                        Signature = HMACSHA256(
                        base64UrlEncode(header) + "."
+
                        base64UrlEncode(payload),
                        secret)
```

- **Step 4.** Access token in serialized form and refresh token (GUID) returned in response with HTTP **200Success** code to the **Client**.
- **Step 5. Client** queries the **API / Microservice** providing access token as **Bearer** in request header.
- **Step 6. API / Microservice** validates the token claims in order to authorize user
  - If authorized: **API / Microservice** does business logics and returns response **(Step 7)**.
  - Otherwise: returns **401Unauthorized** response code.
- **Step 7. API / Microservice** returns response **200Success** or **409Conflict** to the client according to backend business logic.

3.3 Security and User Privacy Vulnerabilities Mitigations

3.4 Database Structure

As a next topic to uncover goes the one related to the database structure. Proper database structure is to be of very high priority, since it influences on the project as a whole, starting from performance point of view and many other aspects. When we say performance, we mean such a design of database that there is an opportunity to include required indexes, depending on regularity of any particular request. So, as the main topic of current thesis is implementation of Instant messaging system, we can list the following general entities to be added to database schema.

- **Users.** Table stores information about user. Table contains following columns:
  - Id VARCHAR(36) – Id of the user, primary key, GUID.
  - UserName VARCHAR(50) – Unique Username.
  - NormalizedUserName VARCHAR(50) – Unique Username in upper case.
  - DisplayName VARCHAR(50) – Name of the user, displayed to others.
  - Bio VARCHAR(250) – User's short biography, visible to others.
  - Image VARCHAR(36) – User's profile picture, visible to others.
  - Email VARCHAR(120) – User's email address, not public.
  - EmailConfirmed BOOLEAN – Flag that indicates if user has confirmed his email address.

  - PhoneNumber VARCHAR(50) – User's phone number.
  - PhoneNumberConfirmed BOOLEAN – Flag that indicates if user has confirmed his phone number.
  - PhoneVerificationCode INTEGER – Code sent to user in order to confirm phone number.
  - PasswordHash VARCHAR(60) – Hashed password.
  - CreatedAt DATETIME – Indicates the date and time user has been registered.
  - UpdatedAt DATETIME – Indicates the date and time user has updated his record.

- **User Personal Information.** Table stores additional but not required info about user. Relation one-to-one with Users, foreign key is UserId, GUID. Table contains following columns:
  - UserId VARCHAR(36) – Foreign key to the Users table, GUID.
  - FirstName VARCHAR(120) – First name of user.
  - LastName VARCHAR(120) – Last name of user.
  - BirthDay DATETIME – Birth day of user.
  - WebSite VARCHAR(120) – Web site of user.
  - Address VARCHAR(120) – Residence address of user.
  - Facebook VARCHAR(120) – Facebook nickname of user.
  - Twitter VARCHAR(120) – Twitter nickname of user.
  - Instagram VARCHAR(120) – Instagram nickname of user.
  - LinkedIn VARCHAR(120) – LinkedIn nickname of user.
  - ProfilePicture VARCHAR(36) – Avatar of user.

- **UserContacts.** Table stores the contacts of current user. Relation between tables Users and UserContacts is one-to-many, foreign key UserId. Table contains following columns:
  - ContactId VARCHAR(36) – Id of current user's contact, GUID.
  - UserId VARCHAR(36) – Id of current user, GUID.

- **Chats.** In order to communicate with other people it is essentially to have a chat room. Our implementation provides chat rooms of the four types. Direct chat – chat room between only two members. Public channel – chat room for multiple members, each member can send and read messages. It displayed in search results. Readonly channel – channel for multiple members, however only the owner can send messages. It displayed in search results. Private channel – channel for multiple members, can be joined only by invite link. Each user can have a numerous various chats, however, each chat has a multiple members, at least 2 as the case of direct chat. Therefore, we consider a many-to-many relation between user and chats via intermediate table UserChats. We discuss UserChats relation in foregoing part. Continuing with Chats table, it contains the following columns:

  - Id VARCHAR(36) – Id of the chat, primary key, GUID.
  - ChatInfoId VARCHAR(36) – Since we have a four types of channels, which has a common subset of data, the different data is moved to another table, so it could be joined depending on chat type. For instance, any chat type except direct one would require to join additional data in order to display the chat properly.
  - Title VARCHAR(50) – Simply, the title of the chat.
  - Image VARCHAR(36) – Picture of the chat. Displayed in search results etc.
  - ChatType ENUM – The type of the chat, e.g direct chat, public channel, readonly channel, private channel.
  - CreatedAt DATETIME – Indicates the date and time chat has been created.
  - UpdatedAt DATETIME – Indicates the date and time chat has been updated.

- **UserChats.** Table that considered as composite key of the many-to-many relation between Users and Chats tables. Over that, contains an enum value that indicates user's role in the chat. We assume the following user roles in the system:

  - Owner – the creator of the chat. Has an ultimate privileges.
  - Administrator – designated by the owner user, which has adjustable privileges.
  - Moderator – designated by administrator user, which has adjustable privileges.
  - User – default role assigned to the user on join the chat.

  Despite that, the UserChats table contains the following columns:

  - ChatId VARCHAR(36) – Foreign key to the Chats table, GUID.
  - UserId VARCHAR(36) – Foreign key to the Users table, GUID.

- RoleId ENUM – Indicates the user role in the chat, e.g Owner, Administrator, Moderator, User.

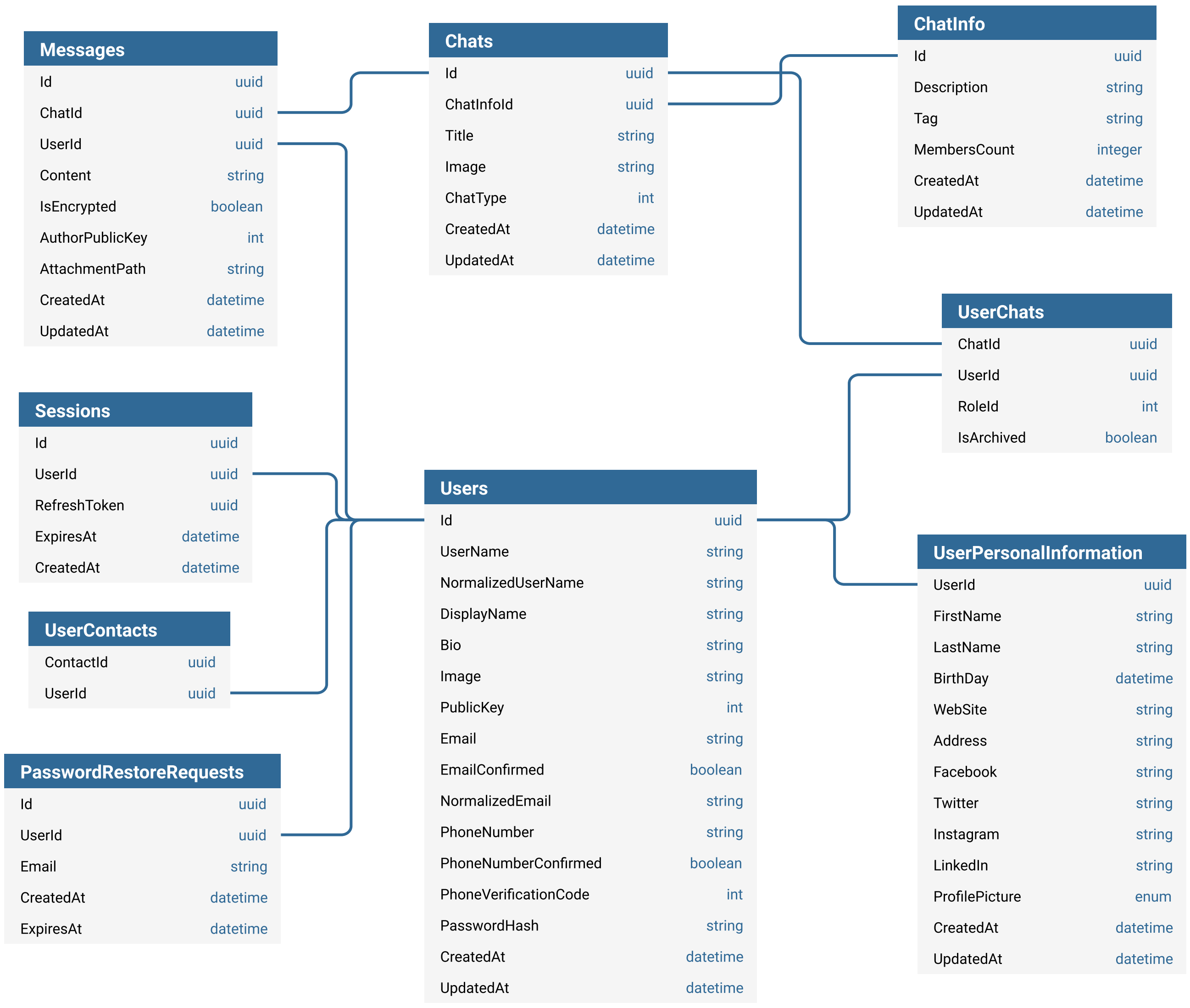
- **Chat Info.** Table contains additional data related to the chat. This table is created since that we have four types of chats, namely, direct chat, public channel, readonly channel, private channel. These four types has a common data between each other. Common data between chat types is Chats table itself. One would advise to store the chats in a single table per chat type, however it is very costly approach, since there would be at least four joins per request. Note that every chat type except direct chat requires an additional data to be displayed, that is Chat Info. Contains following columns:
  - Id VARCHAR(36) – Id of the chat information, primary key, GUID.
  - Description VARCHAR(120) – Description of the chat.
  - Tag VARCHAR(20) – Unique identifier of the chat.
  - MembersCount INTEGER – Count of members in the chat.

- **Messages.** Table that keeps messages and related data. Each chat has a multiple messages, however one message must belong only to single and defined chat, therefore relation between Chats and Messages is one-to-many, foreign key ChatId. From other side, each User has a multiple messages, however single message should belong to single author, therefore, we consider one-to-mane relation between Users and Messages, foreign key UserId. Messages table contains following columns:
  - Id VARCHAR(36) – Id of the message, primary key, GUID.
  - ChatId VARCHAR(36) – Foreign key to the Chats table, GUID.
  - UserId VARCHAR(36) – Foreign key to the Users table, GUID.
  - Content VARCHAR(300) – Content of the message.
  - IsRead BOOLEAN – Indicates whenever message has been read by another user.
  - CreateAt DATETIME – Time when the message has been created.
  - UpdatedAt DATETIME – Time when the message has been updated.

- **Refresh Tokens.** Table stores refresh tokens.
  - Id VARCHAR(36) – Id of the token, primary key, GUID.
  - UserId VARCHAR(39) – Foreign key to the Users table, GUID.
  - RefreshToken VARCHAR(60) – Refresh token itself.
  - Expires DATETIME – Expiration date of refresh token.

– CreatedAt DATETIME – Date when token has been created.

Following diagram demonstrates the database structure.



Messages	
Id	uuid
ChatId	uuid
UserId	uuid
Content	string
IsEncrypted	boolean
AuthorPublicKey	int
AttachmentPath	string
CreatedAt	datetime
UpdatedAt	datetime

Chats	
Id	uuid
ChatInfoId	uuid
Title	string
Image	string
ChatType	int
CreatedAt	datetime
UpdatedAt	datetime

ChatInfo	
Id	uuid
Description	string
Tag	string
MembersCount	integer
CreatedAt	datetime
UpdatedAt	datetime

UserChats	
ChatId	uuid
UserId	uuid
RoleId	int
IsArchived	boolean

Sessions	
Id	uuid
UserId	uuid
RefreshToken	uuid
ExpiresAt	datetime
CreatedAt	datetime

UserContacts	
ContactId	uuid
UserId	uuid

PasswordRestoreRequests	
Id	uuid
UserId	uuid
Email	string
CreatedAt	datetime
ExpiresAt	datetime

Users	
Id	uuid
UserName	string
NormalizedUserName	string
DisplayName	string
Bio	string
Image	string
PublicKey	int
Email	string
EmailConfirmed	boolean
NormalizedEmail	string
PhoneNumber	string
PhoneNumberConfirmed	boolean
PhoneVerificationCode	int
PasswordHash	string
CreatedAt	datetime
UpdatedAt	datetime

UserPersonalInformation	
UserId	uuid
FirstName	string
LastName	string
BirthDay	datetime
WebSite	string
Address	string
Facebook	string
Twitter	string
Instagram	string
LinkedIn	string
ProfilePicture	enum
CreatedAt	datetime
UpdatedAt	datetime

FIGURE 3.6: Database diagram.

3.5 Planned Technologies

- **SDK**: .NET Core 5.0
- **Backend:** ASP.NET Web API
- **Database**

  - SQL Database: PostgreSQL 13
  - ORM: Entity Framework Core 5.0.7
  - PostgreSQL Provider: Npgsql.EntityFrameworkCore.PostgreSQL 5.0.7

- **Authorization**
  - JWT Library: System JWT 6.8.0
  - JWT Library: System Tokens 6.11.1
  - JWT Bearer: Microsoft Jwt Bearer 5.0.7

- **Business Logic**
  - MediatR 9.0.0
  - Fluent Validation 10.2.3
  - AutoMapper 10.1.1

- **Presentation**
  - Documentation: Swashbuckle 6.1.4
  - Realtime Communication: SignalR 2.4.2
  - Frontend Development: Angular 11.2.7
  - Desktop Development: ElectronJS framework.
  - Mobile Development:

- **Unit and Integration Testing**
  - Testing Framework: NUnit 3.13.1
  - Testing Auxiliary: Moq 4.16.1
  - Testing Auxiliary: FluentAssertions 6.0.0

- **Static Code Analysis**: SonarQube 8.9.2 LTS Community Edition

- **Containerization**: Docker 3.6.0

- **Continuous Integration**: GitHub Actions, Heroku, Azure

- **Programming languages**: C#, SQL, TypeScript

- **Tools**: Microsoft Visual Studio, JetBrains Rider, Visual Studio Code, WebStorm.

Chapter 4

End-to-End Encryption

4.1 Diffie-Hellman Key Exchange

Diffie-Hellman key exchange [Li, 2010] is a method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitefield Diffie and Martin Hellman. DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key. Traditionally, secure encrypted communication between two parties required that they first exchange keys by some secure physical means, such as paper key lists transported by a trusted courier. The Diffie-Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. This key can then be used to encrypt subsequent communications using a symmetric-key cipher. Although Diffie-Hellman key agreement itself is a non-authenticated key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to provide forward secrecy in Transport Layer Security's ephemeral modes, referred to as EDH or DHE [Ahirwal and Ahke, 2013] depending on the cipher suite.

Diffie-Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:

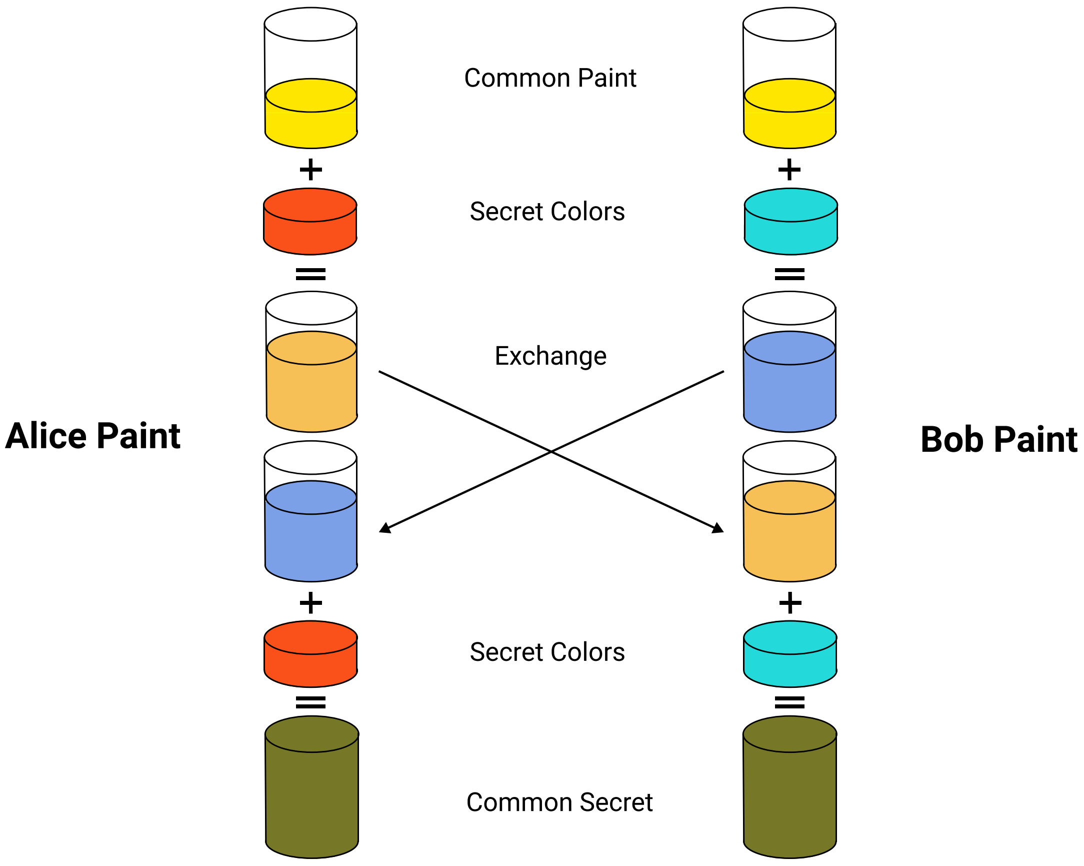


FIGURE 4.1: Illustration of the concept behind Diffie–Hellman key exchange. Source:

The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that does not need to be kept secret (but should be different every time). In this example, the color is yellow. Each person also selects a secret color that they keep to themselves - in this case, red and blue-green. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to the partner's final color mixture. If a third party listened to the exchange, it would only know the common color (yellow) and the first mixed colors (orange-tan and light-blue), but it would be difficult for this party to determine the final secret color (yellow-brown). Bringing the analogy back to a real-life exchange using large numbers rather than colors, this determination is computationally expensive. It is impossible to compute in a practical amount of time even for modern supercomputers.

The simplest and the original implementation of the protocol uses the multiplicative group of integers modulo P, where P is prime, and the generator G which is a primitive root modulo P. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to $P-1$. Here is an example of the protocol

1. Given modulus P and generator G.
2. Alice chooses her secret a.
3. Alice sends to Bob $A,\ A = G^a \bmod P$.
4. Bob chooses his secret b.
5. Bob sends to Alice $B,\ B = G^b \bmod P$.
6. Alice computes common secret $s,\ s = B^a \bmod P = (G^b \bmod P)^a \bmod P$.
7. Bob computes common secret $s,\ s = A^b \bmod P = (G^a \bmod P)^b \bmod P$.
8. Alice and Bob have arrived to the same value

   $$A^b \bmod P = G^{ab} \bmod P = G^{ba} \bmod P = B^a \bmod P,$$

   more specially,

   $$(G^a \bmod P)^b \bmod P = (G^b \bmod P)^a$$

However, to reach a satisfactory level of security through DH key exchange a few rules have to be satisfied. More precisely, Diffie-Hellman works in a multiplicative subgroup of integers modulo a given prime p. To do some DH, you use some DH parameters which are:

- p – a big prime, called the "modulus"
- q – a divisor of $p-1$, called the "subgroup order".
- g – an integer modulo p of order q, this means that the smallest integer $k > 0$ such that $g^k = 1 \bmod p$ is $k = q$.

For DH to be safe, you need the following:

- Prime p must defeat attempt at discrete logarithm through Index Calculus. This means that p must be large enough, and also must not have any "special structure" such as being very close to a power of 2, because such structures allow for improvements in Index Calculus. It so happens that size requirements for

DH are about the same as the size requirements for RSA, though the underlying reason for that is intricate and partly coincidental. So, basically, use a random p of 2048 bits, and you will be fine.

- Number q should be prime or have a prime divisor whose size is enough to defeat generic algorithms for discrete logarithm. If the size (in bits) of the largest prime divisor of q is z, then generic algorithms have a cost in $2^{z/2}$. For best results, arrange for q to be a prime of 256 bits or more.

- Systems that use the parameters to perform a DH key exchange must generate a random integer between 1 and $q - 1$ uniformly, using a cryptographically strong source of randomness, of course. If q is prime and larger than 256 bits, it suffices to choose a 256-bit random value to achieve 128-bit of security. However, if q is not prime, things are more complex: if q has size r bits, and the largest prime divisor of q has size e bits, and $e \geq 256$, then one may choose a random value x of size $r - (e - 256)$ bits to get the usual "128-bit security".

- When p is a so-called "safe prime", then $p = 2r + 1$ for a prime r, so for any generator g that is not 1 or $p - 1$, the order of g will be either r or $2r$, so it suffices to generate DH secret keys x as random 257-bit values. The "safe primes" are not actually any safer than other primes, except for that point: they tolerate the choice of relatively small DH secret keys for any generator.

- Last but not least, DH is a key exchange algorithm that does not, inherently, provide authentication or confidentiality. DH is "safe" only when used within a protocol that uses DH and other algorithms with proper integration to achieve such sought after characteristics as data confidentiality and integrity.

Speaking of which, some (many) SSL/TLS implementations did things improperly, in that they gladly accepted to do DH with weak parameters, in particular a 512-bit modulus. The protocol itself is suboptimal in its handling of DH because the `ServerKeyExchange` message allows the server to send the DH parameters p and g to the client, but not q, leaving the client a bit in the dark. Thus, the client must either "play safe" and generate its key in the full $1, \ldots, p - 1$ range, or try to use a shorter exponent (say, 256 bits, not 2048) for a reduced computational cost, but possibly at risk of weakness in case the subgroup order q is not prime. A better design would have allowed the server to send the value of q and the size of the biggest prime divisor of q. In that respect, the ECDHE cipher suites of SSL/TLS (DH translated to elliptic curves) have a better design.

For a practical answer if you are configuring your SSL/TLS server: you should use a modulus of at least 2048-bit, and a generator g such that the order of g is a prime q of at least 256 bits; alternatively, you may use a modulus p which is a "safe prime", the order of g will then be either a very big prime, or twice a very big prime, which is almost as good. Some people feel safer when they generate their DH parameters "themselves" instead of reusing existing values; if that's what it takes to allow you to sleep at night, then do it.

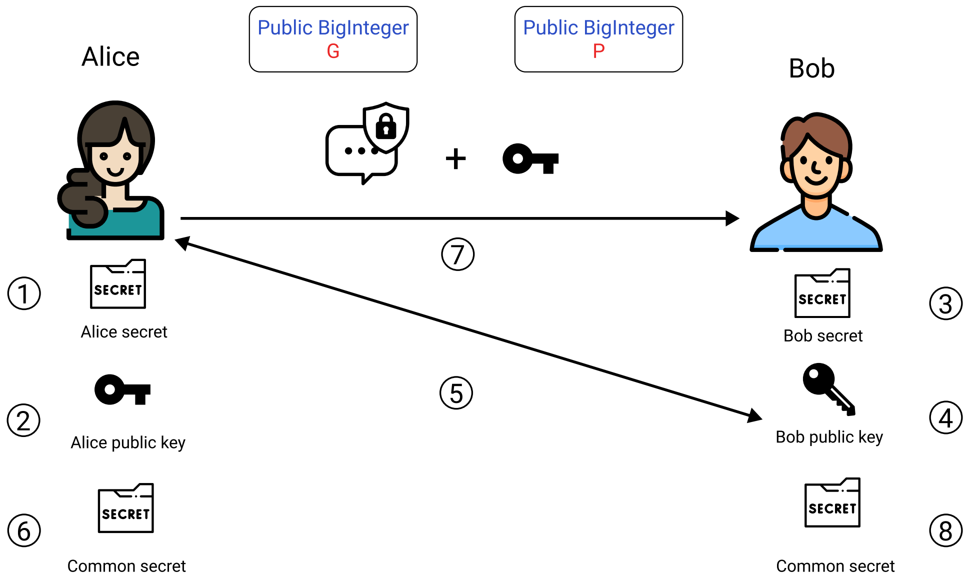


FIGURE 4.2: Secret chat encryption concept diagram. Source:

Assume that Alice wants to write a secret message to the Bob. The secret chat encryption implemented as follows

1. Given two public constants: P, G.
2. Alice generates her secret a.
3. Alice generates public key A as $A = G^a \mod P$ and shares it public.
4. Bob generates his secret b.
5. Bob generates public key B as $B = G^b \mod P$ and shares it public.
6. Alice reads Bob's public key B.
7. Alice calculates Common secret s as $s = B^a \mod P$.
8. Alice encrypts message using AES256 algorithm, then sends it to Bob along her public key A.
9. Bob calculates Common secret s as $s = A^b \mod P$ and decrypts message from Alice.

Although, DH fits the key exchange concerns fine, the secret might be shared via RSA approach as well. We discuss it in next section.

Chapter 5

User Interface

Here to be filled sections with screenshots, after functional requirements updated.

Chapter 6

Conclusions

Bibliography

Ahirwal, Ram Ratan and Manoj Ahke (2013). "Elliptic curve diffie-hellman key exchange algorithm for securing hypertext information on wide area network". In: *International Journal of Computer Science and Information Technologies* 4.2, pp. 363–368.

Alwin, Duane F and Brett A Beattie (2016). "The KISS principle in survey design: question length and data quality". In: *Sociological methodology* 46.1, pp. 121–152.

Brataas, Gunnar and Peter Hughes (2004). "Exploring architectural scalability". In: *Proceedings of the 4th international workshop on Software and performance*, pp. 125–129.

Bucchiarone, Antonio et al. (2018). "From monolithic to microservices: An experience report from the banking domain". In: *Ieee Software* 35.3, pp. 50–55.

Burrows, Michael, Martin Abadi, and Roger Michael Needham (1989). "A logic of authentication". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 426.1871, pp. 233–271.

Chung, Lawrence et al. (2012). *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media.

Cocks, Clifford C (1973). "A note on non-secret encryption". In: *CESG Memo*.

Cohn, Mike (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.

Da Silva, Tiago Silva et al. (2018). "The evolution of agile UXD". In: *Information and Software Technology* 102, pp. 1–5.

Degges, Randall (2019). *JWTs Suck*. https://speakerdeck.com/rdegges/jwts-suck. [Online; accessed 15-August-2021].

Dilworth, John and AK Kochhar (2007). "Creation of an e-business requirements specification model". In: *Journal of Manufacturing Technology Management*.

Fagin, Ronald (1978). "On an authorization mechanism". In: *ACM Transactions on Database Systems (TODS)* 3.3, pp. 310–319.

ISO, BSEN and BRITISH STANDARD (2010). "Ergonomics of human-system interaction". In.

Johnson, Don, Alfred Menezes, and Scott Vanstone (2001). "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1, pp. 36–63.

Jones, M, J Bradley, and N Sakimura (2015). "Rfc 7519: Json web token (jwt)". In: *IETF. May*.

Jones, Michael, Brain Campbell, and Chuck Mortimore (2015). “JSON Web Token (JWT) profile for OAuth 2.0 client authentication and authorization Grants”. In: *May-2015.{Online}. Available: https://tools. ietf. org/html/rfc7523.*

Li, Nan (2010). “Research on Diffie-Hellman key exchange protocol”. In: *2010 2nd International Conference on Computer Engineering and Technology*. Vol. 4. IEEE, pp. V4–634.

Malan, Ruth, Dana Bredemeyer, et al. (2001). “Functional requirements and use cases”. In: *Bredemeyer Consulting*.

Proctor, Frederick M (1999). “Linux, Real-Time Linux, & IPC.” In: *Dr. Dobb’s Journal: Software Tools for the Professional Programmer* 24.11, pp. 32–36.

Rasche, Peter et al. (2016). “Building and Exchanging Competence Interdisciplinarily: Design Patterns as Domain Mediator”. In: *Proceedings of the International Symposium on Human Factors and Ergonomics in Health Care*. Vol. 5. 1. SAGE Publications Sage CA: Los Angeles, CA, pp. 19–24.

Rising, Linda (1998). “Design patterns: Elements of reusable architectures”. In: *The Patterns Handbook: Techniques, Strategies and Applications*, pp. 9–13.

Rivest, Ronald L, Adi Shamir, and Leonard Adleman (1978). “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2, pp. 120–126.

Wang, Mao-Yin et al. (2004). “An HMAC processor with integrated SHA-1 and MD5 algorithms”. In: *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*. IEEE, pp. 456–458.

Wiener, Michael J (1990). “Cryptanalysis of short RSA secret exponents”. In: *IEEE Transactions on Information theory* 36.3, pp. 553–558.

Young, Greg (2010). “CQRS and Event Sourcing”. In: *URL: http://codebetter. com/gregyoung/2010/02/13/cq and-event-sourcing*.

Appendix A

RSA Algorithm comments

A.1 Details

A.1.1 One way functions

One way function – is a function that is easy to compute on every input, but hard to invert given the image of a random input. For instance, the function

$$f(m) = m^e \bmod N \equiv C$$

where e, N are public constants is one-awy function, because it is easy to compute C given m, however it is hard to compute m given C.

A.1.2 Euler's totient theorem

Given a number N and its prime factorization $p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, then Euler's totient function $\phi(N)$ is defined as

$$\phi(N) = (p_1^{e_1} - p_1^{e_1 - 1}) \cdot (p_2^{e_2} - p_2^{e_2 - 1}) \cdots (p_k^{e_k} - p_k^{e_k - 1})$$

In particular, for positive number M such that its factorization is $p1 \cdot p2$, the $\phi(M)$ is

$$\phi(M) = (p_1 - 1) \cdot (p_2 - 1)$$

Euler's theorem relates the modular division and exponent as follows, given number m, then

$$m^{\phi(N)} = 1 \bmod N$$

It means that reminder of division $m^{\phi(N)}$ by N is always 1. By the equality $1^K = 1$

$$M^{K \cdot \phi(N)} = 1 \bmod N$$

If we multiply both parts by M, we get

$$M \cdot M^{K \cdot \phi(N)} = M^{K \cdot \phi(N)+1} = M \bmod N$$

A.2 RSA Encryption algorithm

The RSA algorithm is named after Ron Rivest, Adi Shamir and Len Adleman, who invented it in 1977 [Rivest, Shamir, and Adleman, 1978]. The basic technique was first discovered in 1973 by Clifford Cocks [Cocks, 1973] of CESG (part of the British GCHQ) but this was a secret until 1997. The patent taken out by RSA Labs has expired.

Historically, the process of encryption is considered to be symmetric one. That means that prior the communication, the sides conclude on the common key to be used in encryption. This process is similar to the first sharing keys and only after that the locked chest with the message. Such approach is highly cost since it requires to share the defined keys between each actor if the number of actors is greater than 2. Much more simpler is to think about secured communication channel that in terms of asymmetric encryption. The real life example would be if Alice shares with all actors an opened lock having key. So that Bob receives an opened lock, writes letter to Alice, puts letter to the chest, locks this chest with received from Alice lock. This way, only Alice will be able to open the chest and to read the letter. This is an idea of the asymmetric encryption. However, such a simple from first glance idea requires complex number theory approach. A concept of opened lock may be interpreted in terms of one-way functions. One way function – is a function that is easy to compute on every input, but hard to invert given the image of a random input. Thus, it is much simpler to close the lock without key, but very difficult to open lock trying the combinations of the key. For instance, the function

$$f(m) = m^e \bmod N = C$$

where e, N are public constants is one-awy function, because it is easy to compute C given m, however it is hard to compute m given C. So, assume that Alice defines two positive integer constants e, N and sends it to Bob. Bob encrypts the secret message m using $f(m)$

$$f(m) = m^e \bmod N = C$$

Then Bob sends encrypted message C to the Alice. Given C Alice must fetch the Bob's message m. In order to decrypt C, Alice has to compute

$$C^d \bmod N = m^{ed} \bmod N \equiv m,$$

where e for encryption and d for decryption. Now the problem is to define such d that it is hard to the listener to fetch it. In order to define the secret d, Alice chooses two enough big prime numbers: $P,\ Q$, let's say around 150 digits both. Then Alice multiplies these two prime numbers in order to get N

$$N = P \cdot Q$$

The N is around 300 digits. Now Alice can share N with anyone, since it takes decades to find its prime factorization by the fundamental problem of prime factorization. Next, it is very important to know such a function, which depends on the knowledge of factorization of N. Such function is an Euler's totient function. Given a number N and its prime factorization $p_1^{e_1} \cdot p_2^{e_2} \cdots p_k^{e_k}$, the Euler's totient function $\phi(N)$ is defined as

$$\phi(N) = (p_1^{e_1} - p_1^{e_1-1}) \cdot (p_2^{e_2} - p_2^{e_2-1}) \cdots (p_k^{e_k} - p_k^{e_k-1})$$

In particular, for positive number M such that its factorization is $p1 \cdot p2$, the $\phi(M)$ is

$$\phi(M) = (p_1 - 1) \cdot (p_2 - 1)$$

Euler's theorem relates the modular division and exponent as follows, given number m, then

$$m^{\phi(N)} = 1 \bmod N$$

It means that reminder of division $m^{\phi(N)}$ by N is always 1. By the equality $1^K = 1$

$$M^{K \cdot \phi(N)} = 1 \bmod N$$

If we multiply both parts by M, we get

$$M \cdot M^{K \cdot \phi(N)} = M^{K \cdot \phi(N)+1} = M \bmod N$$

It follows that Alice is able to define the secret d as follows

$$e \cdot d = K \cdot \phi(N) + 1$$
$$d = \frac{K \cdot \phi(N) + 1}{e}$$

The following image demonstrates the concept of RSA approach

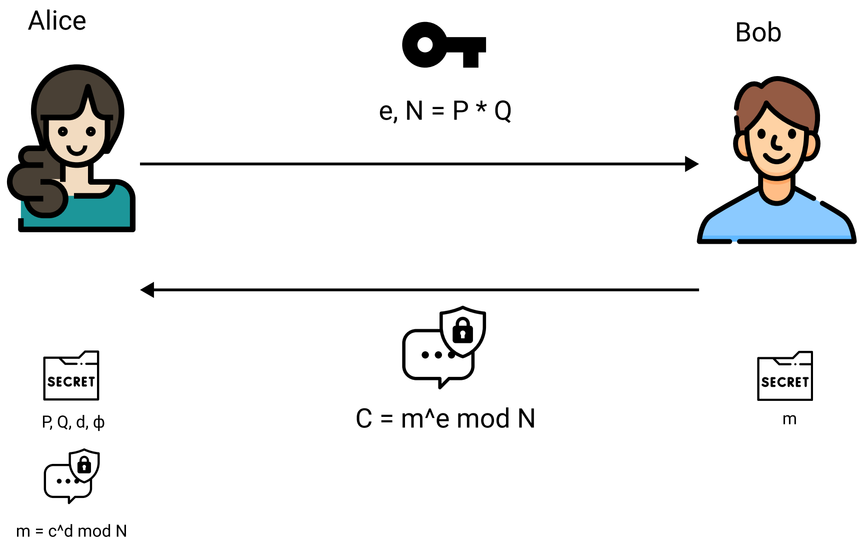


FIGURE A.1: Secret chat encryption concept diagram. Source:

To summarize, the process by the steps is as follows

- Alice defines the large secret prime numbers $P,\ Q$.
- Alice computes $N = P \cdot Q$ and $\phi = (P-1)(Q-1)$
- Alice chooses an integer e, $1 < e < \phi$ such that $\gcd(e, \phi) = 1$.
- Alice computes secret exponent d, $1 < d < \phi$ such that $ed \equiv 1 \bmod \phi$.
- Alice shares public key (N, e) with Bob and keeps private key (d, p, q) is secret.
- Bob defines the message m, encrypts it as $C = m^e \bmod N$.
- Bob sends C to Alice.
- Alice decrypts C using her secret d, so she gets m

$$m = C^d \bmod N$$

Security of the RSA approach is based on the complexity of fundamental problem of prime factorization, which takes decades to solve having enough large number.