Table of contents

SafeInfo documentation

Safeinfo

What is SafeInfo?

SafeInfo is a messaging app project developed to explore and address the challenges of building a secure communication platform. While it includes the essential features of any messaging app, its primary focus is on robust security, ensuring that messages remain private and protected from any prying eyes.

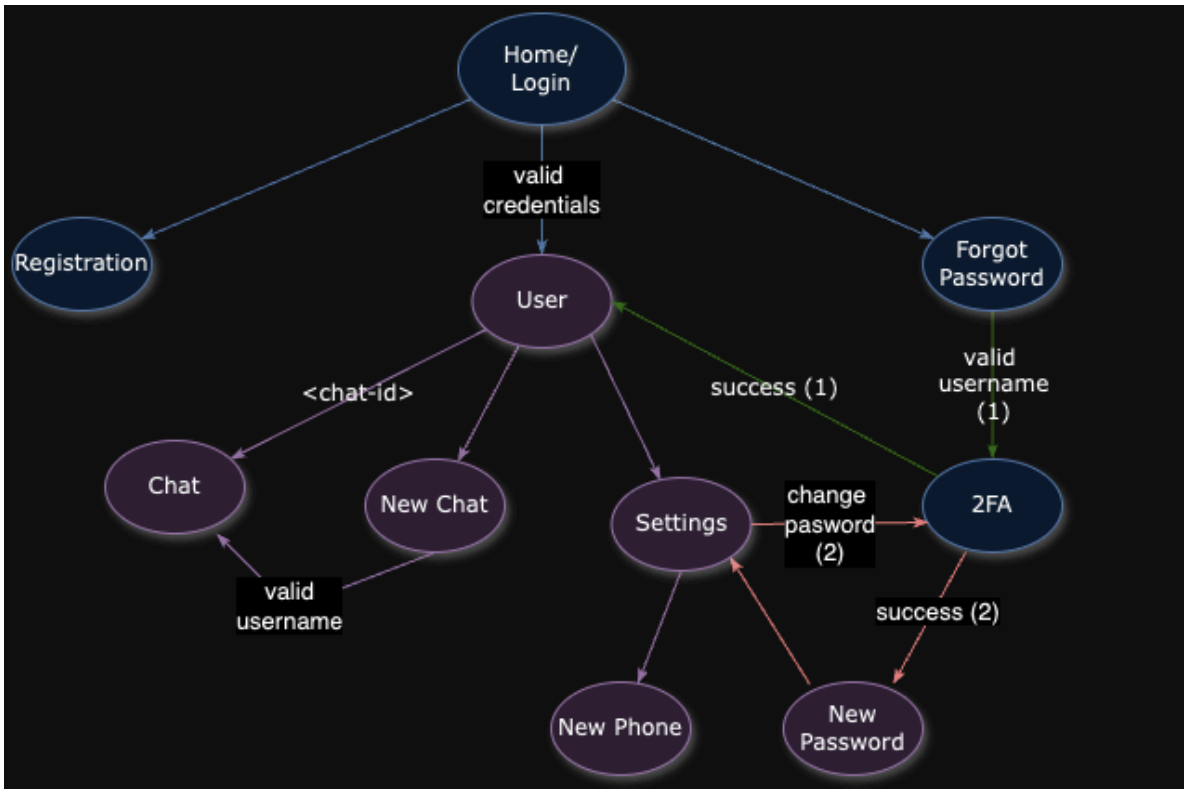
Features of SafeInfo

- Text, image, and emoji messaging

- Server communication via custom APIs
- End-to-end (E2E) encryption
- Encrypted chat history storage
- Two-factor authentication (2FA) via SMS verification

Overview of Client-Side Architecture

Only one panel can be open at a time. When switching panels, the previous panel is removed from memory, and the new panel is initialized (lazy loading). Here's a quick overview of how panels are managed within the client-side application:



Client panels

The format for saving messages in the chat history file is:

```
{
    "<chat-id1>": {
        "<user>": [
            {"m": "<m1>", "t": "<t1>"},
            {"m": "<m2>", "t": "<t2>"},
```

```
            ...
            ],
        "<otherUser>": [
            {"m": "<m1>", "t": "<t1>"},
            {"m": "<m2>", "t": "<t2>"},
            ...
            ]
    },
    "<chat-id2>": {
        ...
    },
    ...
}
```

In this structure, "m" represents the encrypted message, and "t" is the timestamp associated with each message.

The keys is used in the cryptographic operations are stored in secrets.txt. More about how these values are generated in the How Are Cryptographic Features Implemented? section.

```
*---------------------*
|ENCRYPTED PRIVATE KEY|
*---------------------*
|BASE64 ENCODED IV    |
*---------------------*
|ENCRYPTED MASTER KEY |
*---------------------*
```

Overview of Server-Side Architecture

The server operates on an Apache2 Tomcat server within a Docker container, where it manages database communications for user credentials and the logic for communication channels between users.

Each communication channel has a unique chat ID. Messages from clients are stored in separate files based on their chat ID, with each file containing messages in JSON format (mirroring the chat history stored on the client-side):

```
{
    "<first-user>": [
        {"m": "<m1>", "t": "<t1>"},
        {"m": "<m2>", "t": "<t2>"},
        ...
        ],
    "<second-user>": [
        {"m": "<m1>", "t": "<t1>"},
        {"m": "<m2>", "t": "<t2>"},
        ...
        ]
}
```

Database Structure

1. **User Authentication Table (`Users`):**

The Users table stores authentication-related data with the following structure:

```
CREATE TABLE Users (
id INT PRIMARY KEY,
username VARCHAR(20),
passwordHash VARCHAR(65),
phoneNumber VARCHAR(14),
publicKey TEXT NOT NULL
);
```

- `id`: Unique identifier for each user, reserved for future use.
- `passwordHash`: Stores the SHA-256 hash of the user's password.
- `publicKey`: Initially set to 0 and updated to the actual public key when an RSA key pair is generated upon authentication.

2. Chat Logic Table (Chat):

The Chat table manages information for each conversation:

```
CREATE TABLE Chat {
chat_id VARCHAR(64) PRIMARY KEY,
user1 VARCHAR(20),
ua1 BOOLEAN,
lastReqTs1 VARCHAR(30),
user2 VARCHAR(20),
ua2 BOOLEAN,
lastReqTs2 VARCHAR(30)
}
```

- `chat_id`: Unique UUID for each communication channel.
- `user1` and `user2`: Represent the two users in the chat.
- `ua1` and `ua2`: "User aware" flags for user1 and user2, initially set to 0. Once a user accesses the chat, the server sets their flag to 1, indicating the chat is no longer "new" for that user.
- `lastReqTs1` and `lastReqTs2`: Track the last time each user polled for messages, helping avoid duplicate messages in future requests.

For example, if user1 initiates a chat with user2, the server assigns a chat_id but initially sets ua2 to 0 since user2 hasn't yet seen the chat. When user2 logs in and the server detects the new chat, it sets ua2 to 1, indicating the user is aware of this chat. This ensures that chat_id isn't repeatedly flagged as "new" for future requests.

How Are Messages Handled?

When a client decides to communicate with another registered user, the messages are encrypted before being sent to the server. They are also stored locally on the client's file system, allowing the user to view sent and received messages offline.

Each time a user opens a chat with another user, a background worker thread initiates polling to the server every 3 seconds. Upon receiving a request, the server compares the timestamp of the last polling request with the timestamps of any messages linked to the current chat ID. Messages with timestamps more recent than the previous request are added to an array of new messages, which is then sent back to the user in the response body.

```
{"messages":[{"m": "<m1>", "t": "<t1>"}, {"m": "<m2>", "t": "
<t2>"},...]}
```

Once all new messages are identified, the server generates a new timestamp for the current request and updates it in the database, establishing it as the reference time for the next polling request.

To poll messages from the server, send a `GET` request with the user's token and the chat ID included in the URL.

To send messages to the server, send a `POST` request with the user's token and a message object: `{"m": "<message>","t": "<timestamp>"}`

How Are Authentication and Authorization Managed?

Authentication

After successfully registering an account, a user must log in with a username and password. The SHA-256 hash of the provided password is generated and compared to the stored hash on the server. If the username exists and the password is correct, the user is logged into their account.

If a user forgets their password, they can authenticate via an alternative method. After entering their username, they are redirected to a 2FA panel where they must provide a one-time passcode (OTP) sent to the phone number registered with the account. If the OTP is correct, the user is logged in, but they must change their password in settings to re-enable password-based access to the account.

Authorization

Once logged in, the user requests a JWT token, valid for 30 minutes, to verify their identity for any user-related chat or API requests. The JWT is signed with an HMAC SHA-256 algorithm, using a server-only key to ensure session integrity. The session token is stored locally on the client side for reuse and is deleted when the user logs out.

How Are Cryptographic Features Implemented?

Most cryptographic security features are handled client-side within the CryptoProvider class, which applies different algorithms based on the type of encryption needed.

Chat History Encryption

To protect messages from potential malicious third parties, message content is encrypted using AES in CBC mode. This process uses a random 16-byte IV and a 256-bit random master key for encryption security.

End-to-End (E2E) Encryption

Before messages are sent to the server, they are encrypted by the client using the public key of the intended recipient (identified by the associated chat ID) with RSA encryption. This ensures that only the intended recipient can read the message, as the server cannot decrypt it. When a user receives messages from the server, they are encrypted and must be decrypted locally with the user's private key.

Key Storage

The master key and private key are critical to the app's cryptographic security. To keep them confidential, they are stored locally with added protection. The private key is encrypted using the master key, and the master key is further XORed with a "data key" for security. This data key is derived from the user's password hash combined with a known salt using PBKDF2 with HMAC SHA-256, allowing secure re-encryption of the master key when needed.

To learn more about client-server communication, check the API documentation

API documentation

- Login
- Registration
- Forgot Password
- Verify Phone Number
- Verify OTP
- New Chat IDs
- Get Public Key
- Send Public Key
- Get Session
- Get Chat ID
- Change Phone
- Change Password

Login

Endpoint: /api/login

- **Method**: POST
- **Description**: Authenticates a user with their username and password.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.
`passwordHash`	String	Yes	The password hash for the account.

Example Requests and Responses

Request Example

```
POST /api/login
Content-Type: application/json

{
    "username": "exampleUser",
    "passwordHash": "examplePasswordHash"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".
`phoneNumber`	String	Phone number associated with the account.

Response Example Successful Login:

```
{
    "status": "success",
    "phoneNumber": "+0123456789101"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Invalid username or password."
}
```

Registration

Endpoint: /api/registration

- **Method**: POST
- **Description**: Registers a new user with username, password and phone number.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.
`passwordHash`	String	Yes	The password hash for the account.
`phone`	String	Yes	The phone number for the account.

Example Requests and Responses

Request Example

```
POST /api/registration
Content-Type: application/json

{
    "username": "exampleUser",
    "password": "examplePasswordHash",
```

```
    "phone":"examplePhoneNumber"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".

Response Example

Successful Login:

```
{
    "status": "success"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Invalid username or password."
}
```

Forgot-Password

Endpoint: /api/forgotPassword

- **Method**: POST
- **Description**: Authenticates a user by sending a verification code to the associated phone number account.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.

Example Requests and Responses

Request Example

```
POST /api/forgotPassword
Content-Type: application/json

{
    "username": "exampleUser"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Invalid username."
}
```

Verify-Phone-Number

Endpoint: /api/verifyPhoneNumber

- **Method**: POST
- **Description**: Sends a verification code to user's phone number.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.
`hashedPassword`	String	Yes	The password hash for the account.

Example Requests and Responses

Request Example

```
POST /api/verifyPhoneNumber
Content-Type: application/json

{
    "username": "exampleUser",
    "hashedPassword": "examplePasswordHash"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".
`phoneNumber`	String	Phone Number associated with user's account

Response Example Successful Login:

```
{
    "status": "success",
    "phoneNumber": "+123456789101"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Invalid username or password."
}
```

Verify-OTP

Endpoint: /api/verifyOTP

- **Method**: POST
- **Description**: Validates the OTP code, returning "success" upon successful login.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.
`otp`	String	Yes	The OTP code receives through SMS.

Example Requests and Responses

Request Example

```
POST /api/verifyOTP
Content-Type: application/json

{
    "username": "exampleUser",
    "otp": "123456"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".
`passwordHash`	String	The password hash of the user.

Response Example Successful Login:

```
{
    "status": "success",
    "passwordHash": "992feb10b..."
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to get password hash from database"
}
```

New-Chat-Ids

Endpoint: /api/newChatIds

- **Method**: POST
- **Description**: Returns a list of new chat ids.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.

Example Requests and Responses

Request Example

```
POST /api/newChatIds
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`token`	String	JWT token for session authentication.
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success",
    "newChatIds" : ["<chat-id1>","<chat-id2>",...]
}
```

```
{
    "status": "success",
    "message": "No new chat ids found"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to find new chat ids in database"
}
```

Get-Public-Key

Endpoint: /api/getPubKey

- **Method**: POST
- **Description**: Requests the RSA public key of another user.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.
`otherUser`	String	Yes	The username of the other user.

Example Requests and Responses

Request Example

```
POST /api/getPubKey
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    "otherUser": "exampleOtherUser"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".
`pubKey`	String	Base64 encoded public key for the other user.

Response Example Successful Login:

```
{
    "status": "success",
    "pubKey": "MIIBIjANBgkqhkiG9w0BAQEFAAO..."
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to get pubKey from database"
}
```

Send-Public-Key

Endpoint: /api/sendPubKey

- **Method**: POST
- **Description**: Sends the base64 encoded public key of current user to server.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.
`pubKey`	String	Yes	The public key for the current user.

Example Requests and Responses

Request Example

```
POST /api/sendPubKey
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "pubKey": "MIIBIjANBgkqhkiG9w0BAQEFAAO..."
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to update pubKey field"
}
```

Get-Session

Endpoint: /api/getSession

- **Method**: POST
- **Description**: Retrieves a session token with the provided credentials.

Parameters

Field	Type	Required	Description
`username`	String	Yes	The username of the user.
`passwordHash`	String	Yes	The password hash for the account.

Example Requests and Responses

Request Example

```
POST /api/getSession
Content-Type: application/json

{
    "username": "exampleUser",
    "passwordHash": "examplePasswordHash"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".
`token`	String	JWT token for session authentication.

Response Example Successful Login:

```
{
    "status": "success",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Wrong username or password"
}
```

Get-Chat-Id

Endpoint: /api/getChatId

- **Method**: POST
- **Description**: Retrieves an existing chat id for a specified user.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.
`otherUser`	String	Yes	The username of the other user.

Example Requests and Responses

Request Example

```
POST /api/getChatId
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "otherUser": "exampleOtherUser"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`token`	String	JWT token for session authentication.
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success",
    "chatId": "84c10f60-f1cf-4045-b018-113e9dd40a49"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to get chat id from database"
}
```

Change-Phone

Endpoint: /api/changePhone

- **Method**: POST
- **Description**: Requests a new phone number.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.
`phone`	String	Yes	The new phone number for the account.

Example Requests and Responses

Request Example

```
POST /api/changePhone
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "phone": "+0123456789101"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success"
}
```

Failed Login:

```
{
    "status": "error",
```

```
    "message": "Failed to change phone number; user may not exist"
}
```

Change-Password

Endpoint: /api/changePassword

- **Method**: POST
- **Description**: Requests a new password.

Parameters

Field	Type	Required	Description
`token`	String	Yes	JWT token for session authentication.
`newPassword`	String	Yes	The new password for the account.

Example Requests and Responses

Request Example

```
POST /api/changePassword
Content-Type: application/json

{
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "newPassword": "examplePassword"
}
```

Response

Field	Type	Description
`status`	String	"success" or "error".
`message`	String	Error details, if the status is "error".

Response Example Successful Login:

```
{
    "status": "success"
}
```

Failed Login:

```
{
    "status": "error",
    "message": "Failed to change password; user may not exist"
}
```