Professor: Massimo Villari

Project Advisor: Mario Colosi

Student: Arya Khosravirad

Matricola: 534 170

Università degli Studi di Messina

End-to-End Encrypted Real-Time Chat Application

Table Of Contents

Introduction

The goal of this project is to create a secure and real time chat application where users can send and receive messages. The system uses client-side end-to-end encryption with AES to ensure that only the sender and receiver can read the messages, while the server stores only encrypted data. Key features are a simple user interface, user authentication for secure logins, and real time message updates. The application is designed to run in a containerized environment using Docker, which simplifies deployment and ensures consistent performance across different setups. This project prioritizes security and user privacy, providing a safe platform for private communication.

System Architecture

This chat application is built with three main parts: the client, the server, and the database.

Client: The client is responsible for the user interface and the operations that take place on the user's device. It is built using HTML, CSS, and JavaScript. The client performs tasks such as encrypting messages using AES before they are sent to the server and decrypting messages when they are received. This ensures that the project uses client-side encryption, protecting messages so that only the sender and receiver can access the content. It also provides a simple and interface where users can log in, send, and view messages in real time. The client communicates with the server using the Fetch API to ensure smooth and fast updates.

Server: The server is designed to handle back end operations such as user authentication, session management, and communication with the database. It does not perform any encryption or decryption of messages, ensuring that the server only stores and processes encrypted data. This approach enhances security by limiting access to plaintext messages. The server is developed using PHP and runs in a Dockerized environment.

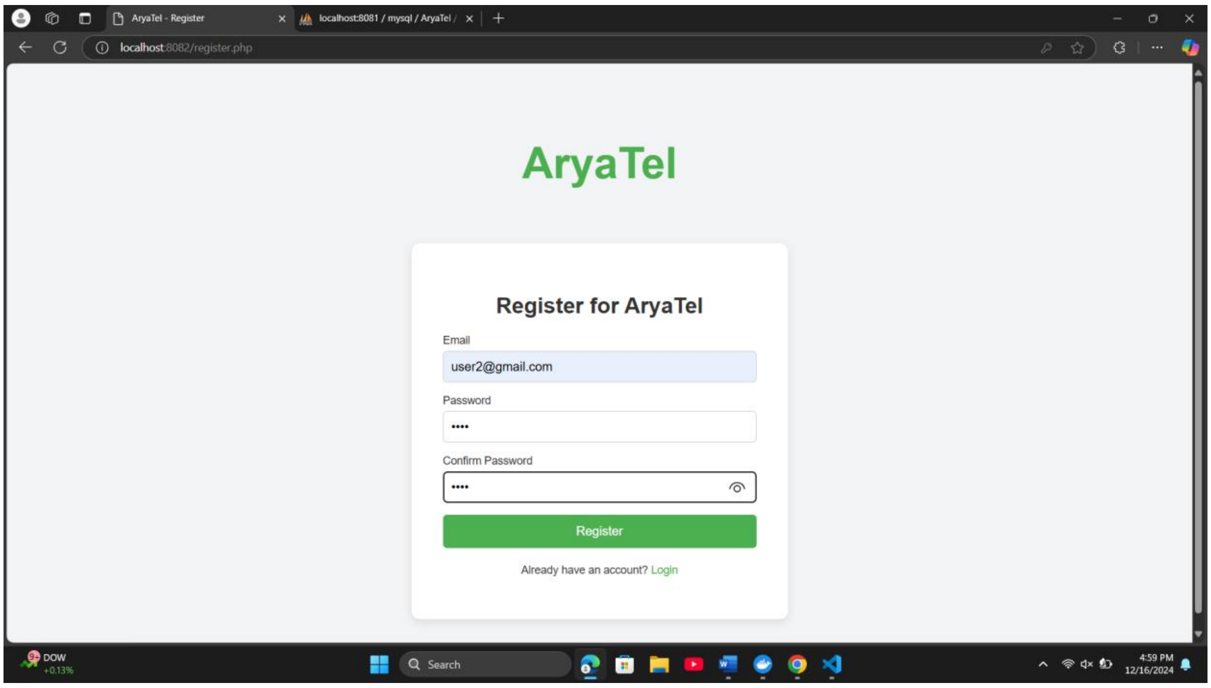
Database: The database is used to store and manage user credentials and encrypted messages. Using MySQL, it ensures that sensitive data remains secure and consistent. By maintaining relationships between users and their messages, the database supports efficient data retrieval. Docker volumes are employed to persist the database content, ensuring reliability across different setups.

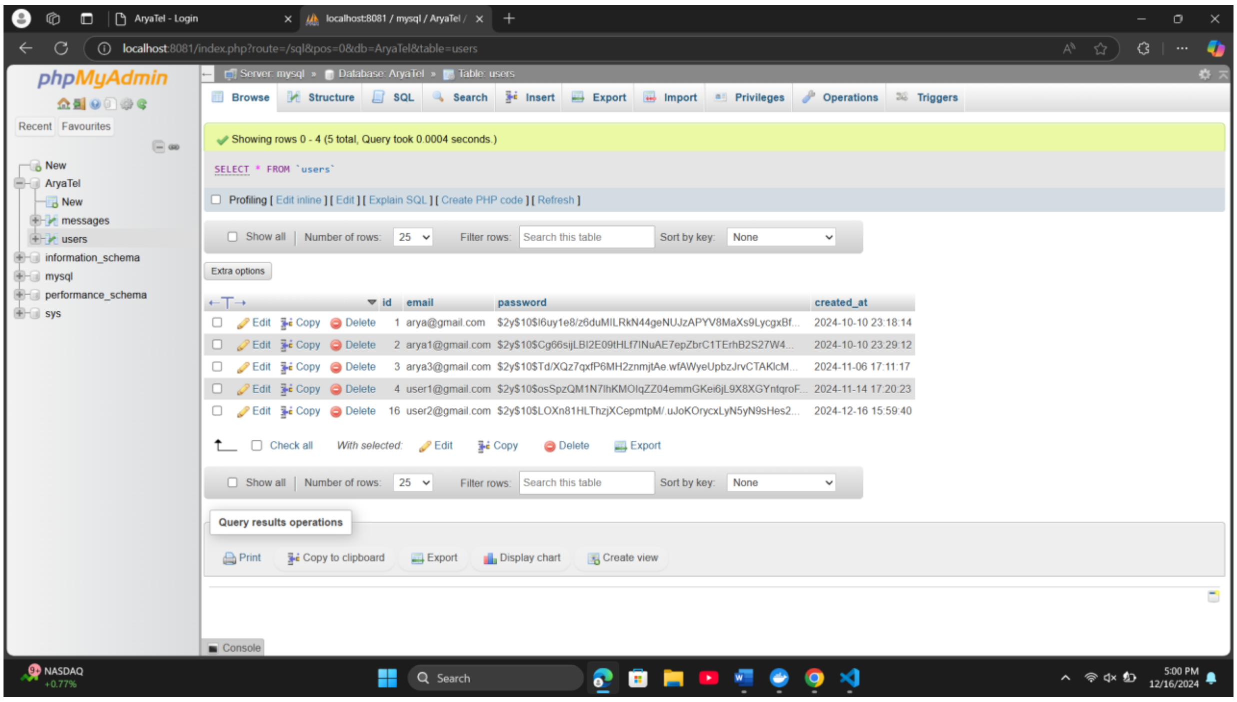
To make deployment easier and more secure, the entire application is containerized with Docker. Docker packages the client, server, and database into isolated environments, ensuring they work seamlessly together across different systems. This containerized setup also keeps each part of the application separate, reducing potential conflicts and making it easier to manage and deploy the app.

Functionality:

Registration:

The registration process allows users to create an account securely. When a user submits their email and password the system first checks if the passwords match. If yes the password is hashed using PHP's password_hash function. The hashed password and email are then safely stored in the database. This ensures that sensitive information remains secure as the actual password is never stored in plain text.




localhost:8081/index.php?route=/sql&pos=0&db=AryaTel&table=users
Server: mysql » Database: AryaTel » Table: users
Browse
Structure
SQL
Search
Insert
Export
Import
Privileges
Operations
Triggers

✓ Showing rows 0 - 4 (5 total, Query took 0.0004 seconds.)

```
SELECT * FROM `users`
```

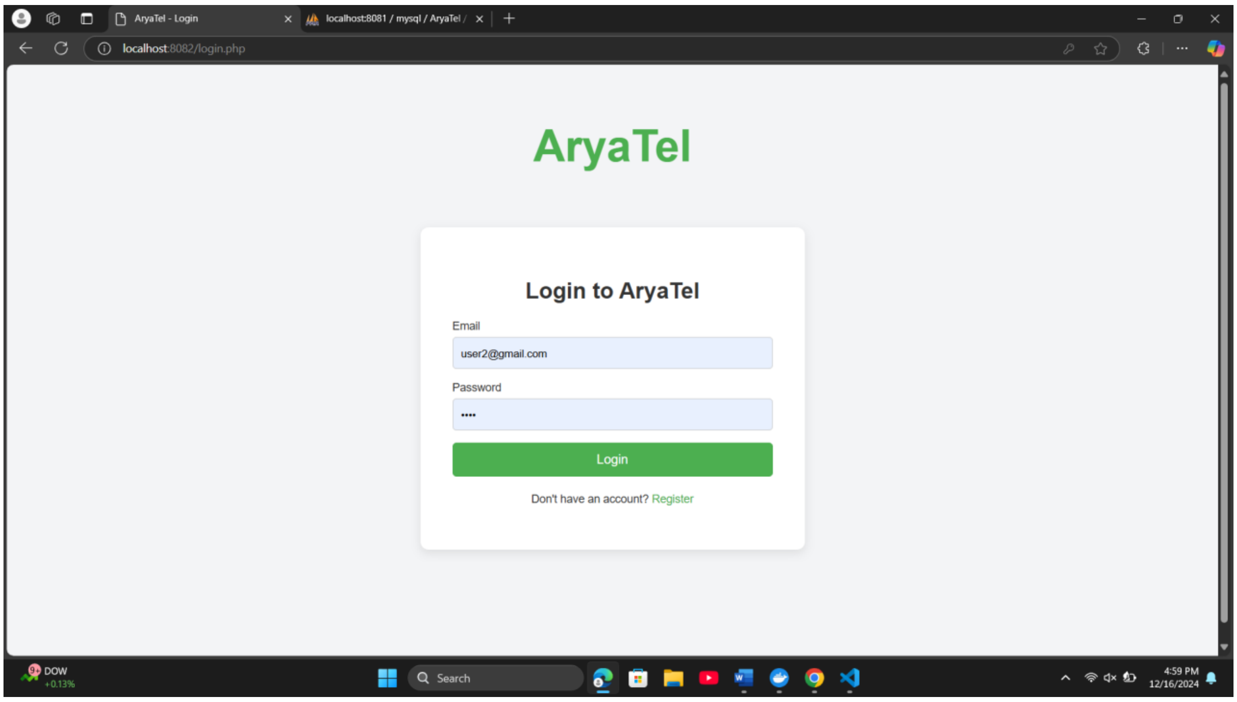
id	email	password	created_at
1	arya@gmail.com	$2y$10$I6uy1e8/z6duMILRkN44geNUJzAPYV8MaXs9LycgxBf...	2024-10-10 23:18:14
2	arya1@gmail.com	$2y$10$Cg66sijLBI2E09tHLf7INuAE7epZbrC1TErhB2S27W4...	2024-10-10 23:29:12
3	arya3@gmail.com	$2y$10$Td/XQz7qxfP6MH2znmjtAe.wfAWyeUpbzJrvCTAKlcM...	2024-11-06 17:11:17
4	user1@gmail.com	$2y$10$osSpzQM1N7lhKMOIqZZ04emmGKei6jL9X8XGYntqroF...	2024-11-14 17:20:23
16	user2@gmail.com	$2y$10$LOXn81HLThzjXCepmtpM/.uJoKOrycxLyN5yN9sHes2...	2024-12-16 15:59:40

This code securely hashes the user’s password before storing it in the database. The password_hash function applies a hashing algorithm to the password making it unreadable to anyone who gains access to the database.

```php
$hashed_password = password_hash($password, PASSWORD_DEFAULT);
$sql = "INSERT INTO users (email, password) VALUES ('$email', '$hashed_password')";
```

Login:

After registration user can try to log in. After submitting their email and password the PHP script checks if the email exists in the database. If the email is found it retrieves the user's hashed password. Then using password_verify, it compares the entered password with the hashed password in the database. If the passwords match the login is successful and the user's ID and email are stored in a session to keep them logged in. and the user is redirected to chat_selection.php. If the email isn't found or the password is incorrect an error message is displayed. This approach securely verifies the user's identity without exposing their actual password.



AryaTel

Login to AryaTel

Email

user2@gmail.com

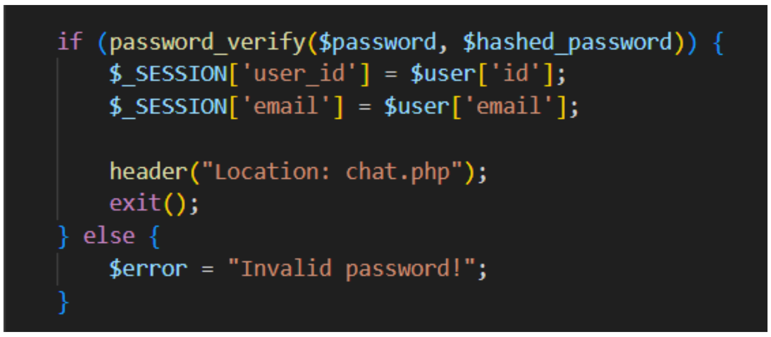
Password

••••

Login

Don't have an account? Register

This part verifies the user's entered password against the hashed password stored in the database ensuring only authenticated users can log in.

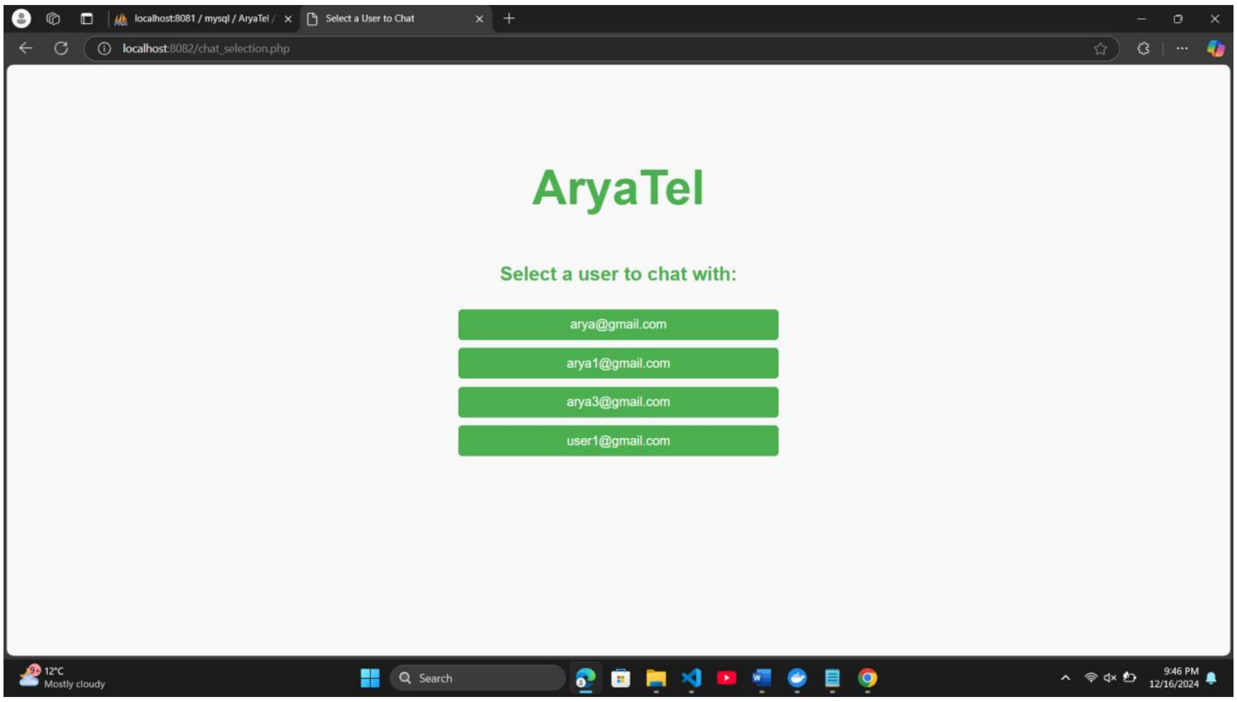


```php
if (password_verify($password, $hashed_password)) {
    $_SESSION['user_id'] = $user['id'];
    $_SESSION['email'] = $user['email'];

    header("Location: chat.php");
    exit();
} else {
    $error = "Invalid password!";
}
```

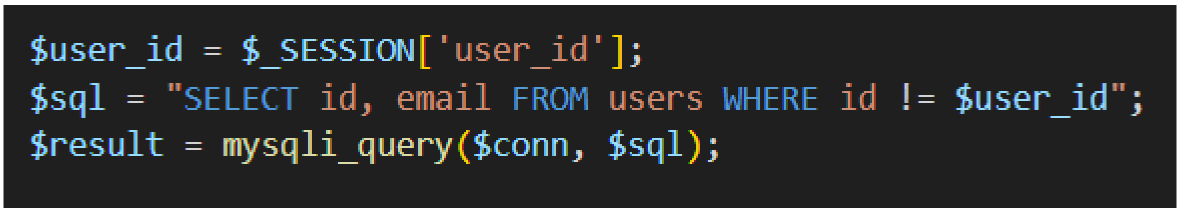
Chat selection:

The chat selection page allows users to choose a person to chat with. When a user logs in, they are shown a list of all other registered users. The currently logged in user is excluded from the list to ensuring they cannot start a chat with themselves.

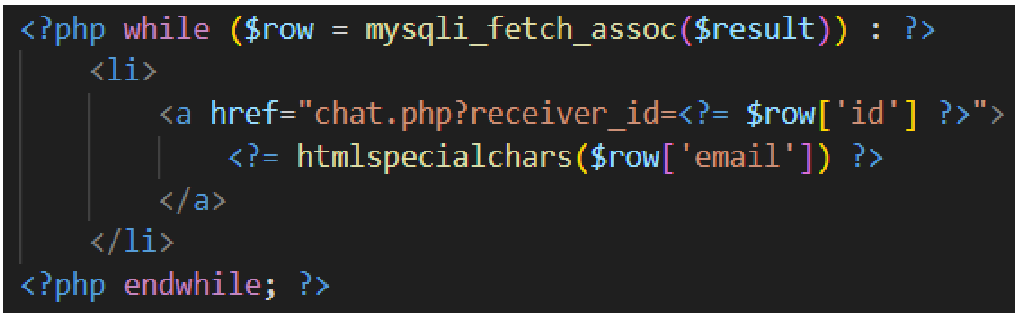
The list of users is fetched from the database and displayed as clickable buttons. Each button redirects the user to the chat page (chat.php) with the selected user's ID passed as a query parameter (receiver_id).



These snippets represent the functionality of fetching all users except the logged in user from the database and display them as clickable links to start a chat.



```php
$user_id = $_SESSION['user_id'];
$sql = "SELECT id, email FROM users WHERE id != $user_id";
$result = mysqli_query($conn, $sql);
```



```php
<?php while ($row = mysqli_fetch_assoc($result)) : ?>
    <li>
        <a href="chat.php?receiver_id=<?= $row['id'] ?>">
            <?= htmlspecialchars($row['email']) ?>
        </a>
    </li>
<?php endwhile; ?>
```

End to end encryption:

In my project, I implemented End-to-End Encryption to ensure that messages remain private between the sender and receiver. The encryption and decryption processes take place entirely on the client side, while the server only stores encrypted data and IVs. This guarantees that the server cannot access plaintext messages at any point.

I designed the system to generate a shared AES key locally for each pair of users by combining their IDs mathematically and using them with PBKDF2 for key derivation. This ensures that both users derive the same key without needing to share it explicitly. Before a message is sent, it is encrypted on the client side using the shared key and a randomly generated IV. The encrypted message and IV are then sent to the server for storage. When the recipient fetches the messages, they decrypt them locally using the same shared AES key this ensures that the encryption is client side.

This approach ensures that only the sender and receiver can read the messages, while the server acts solely as a storage layer. By implementing client side key generation, encryption, and decryption, I achieved a secure system.

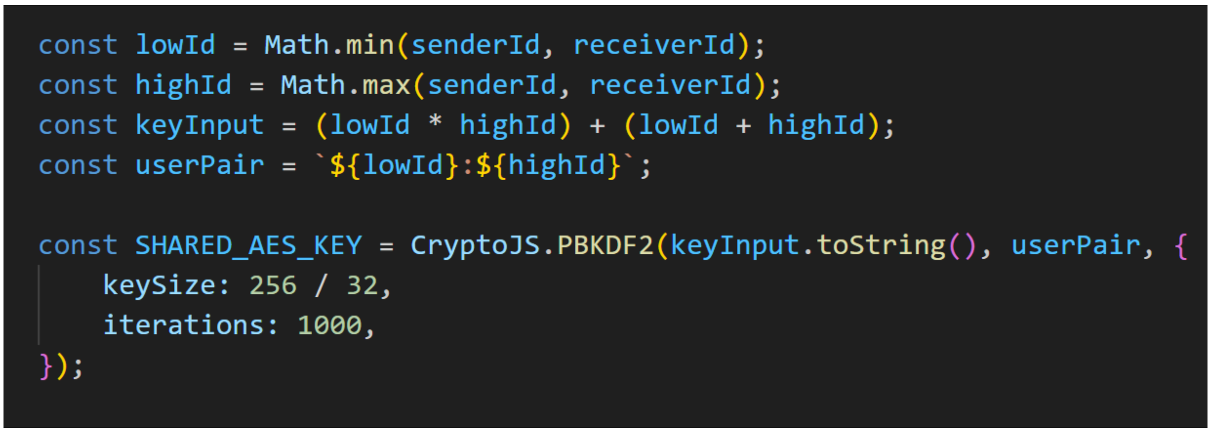
The functionality involves three main components:

1. Client-Side Logic: chat.js
2. Message Storage: send_message.php
3. Message Retrieval: fetch_messages.php

1. Client-Side Logic – chat.js

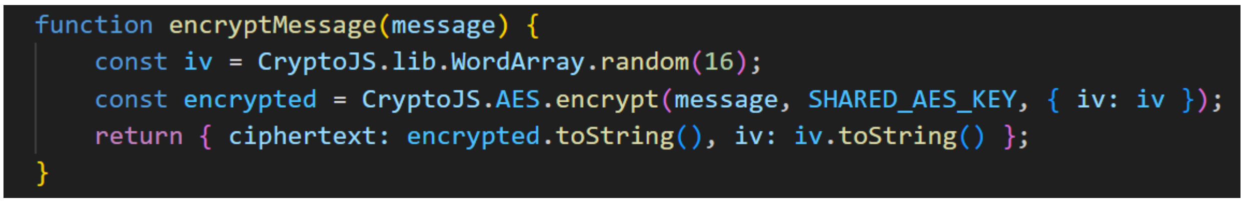
The client-side logic handles key generation, encryption, decryption, and real time updates for the chat system. It ensures that messages remain secure and private by implementing all encryption and decryption tasks on the user's device

Shared Key Generation: I implemented a shared key for each user pair by combining their IDs mathematically. The key input is derived as the sum of the product and addition of the sender and receiver IDs, ensuring a consistent value for both users. This input, along with the sorted IDs as the salt, is passed to CryptoJS.PBKDF2() to generate a secure AES key. The key is never transmitted to the server, ensuring end-to-end encryption.



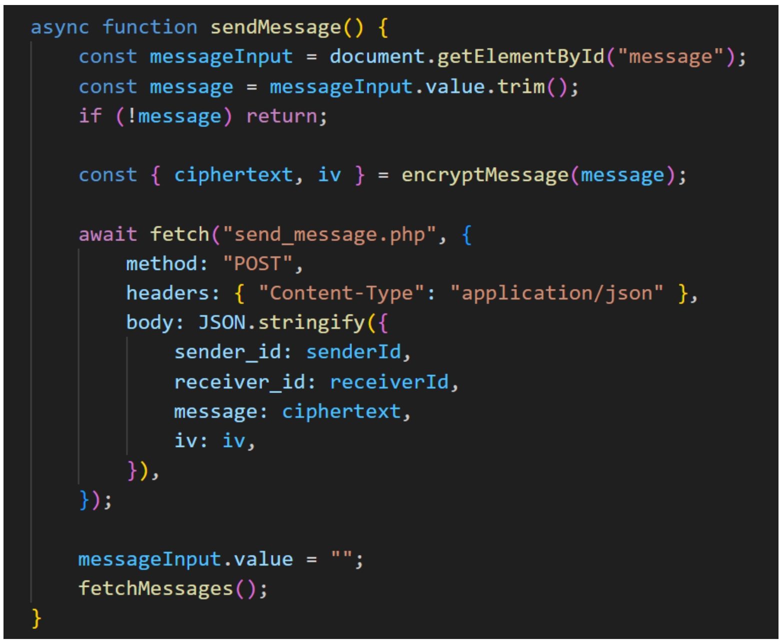
```javascript
const lowId = Math.min(senderId, receiverId);
const highId = Math.max(senderId, receiverId);
const keyInput = (lowId * highId) + (lowId + highId);
const userPair = `${lowId}:${highId}`;

const SHARED_AES_KEY = CryptoJS.PBKDF2(keyInput.toString(), userPair, {
    keySize: 256 / 32,
    iterations: 1000,
});
```

Message Encryption: Before sending, each message is encrypted using the shared AES key. A IV is generated for each message to ensure that even identical messages produce unique ciphertexts



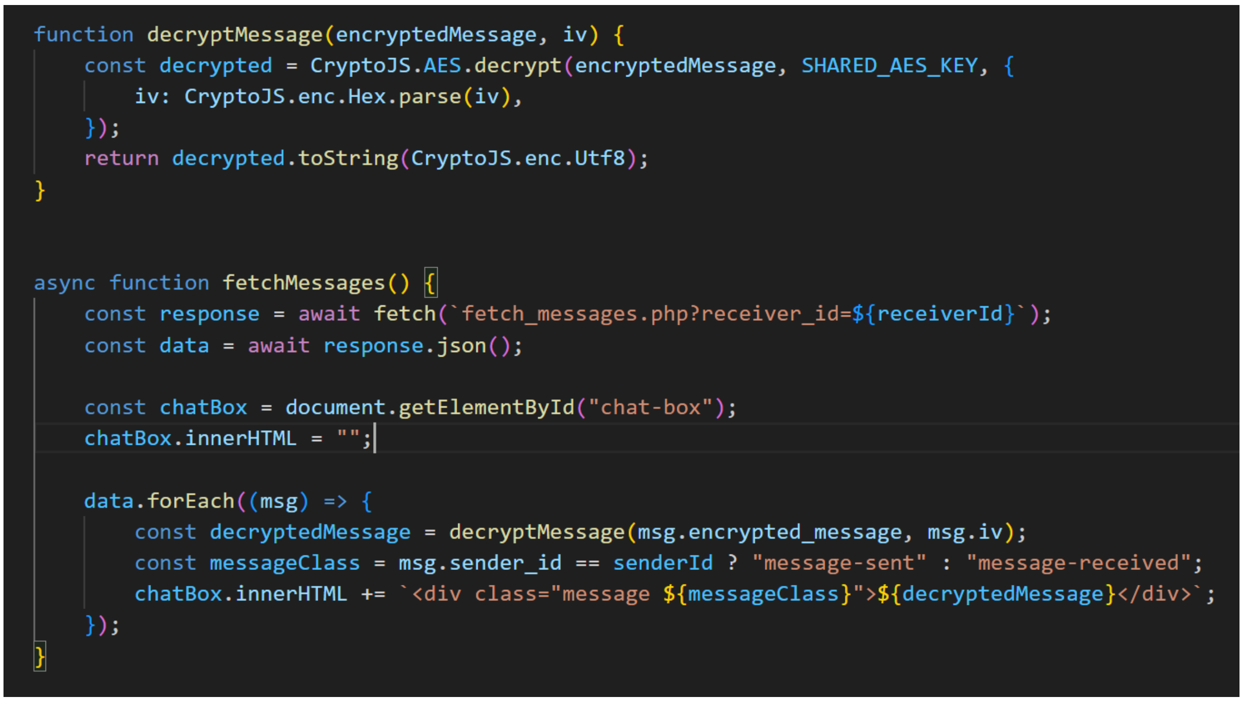
```javascript
function encryptMessage(message) {
    const iv = CryptoJS.lib.WordArray.random(16);
    const encrypted = CryptoJS.AES.encrypt(message, SHARED_AES_KEY, { iv: iv });
    return { ciphertext: encrypted.toString(), iv: iv.toString() };
}
```

Sending Messages: When a user sends a message, the sendMessage() function first encrypts the message using the shared AES key and a generated IV. The encrypted message, IV, sender ID, and receiver ID are sent to the server via a POST request to send_message.php, which stores the data securely in the database.This ensures that the server only stores encrypted data, not plaintext



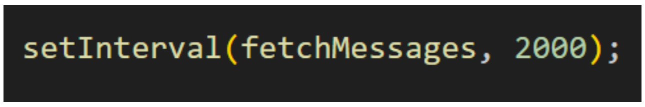
```javascript
async function sendMessage() {
    const messageInput = document.getElementById("message");
    const message = messageInput.value.trim();
    if (!message) return;

    const { ciphertext, iv } = encryptMessage(message);

    await fetch("send_message.php", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
            sender_id: senderId,
            receiver_id: receiverId,
            message: ciphertext,
            iv: iv,
        }),
    });

    messageInput.value = "";
    fetchMessages();
}
```

Fetching and Decrypting Messages: To retrieve messages, the fetchMessages() function sends a GET request to fetch_messages.php, which returns encrypted messages and their IVs. These messages are then decrypted using the shared AES key and IV with the decryptMessage() function. The decrypted messages are displayed in the chat box, ensuring that only the client can view the plaintext content.



```javascript
function decryptMessage(encryptedMessage, iv) {
    const decrypted = CryptoJS.AES.decrypt(encryptedMessage, SHARED_AES_KEY, {
        iv: CryptoJS.enc.Hex.parse(iv),
    });
    return decrypted.toString(CryptoJS.enc.Utf8);
}


async function fetchMessages() {
    const response = await fetch(`fetch_messages.php?receiver_id=${receiverId}`);
    const data = await response.json();

    const chatBox = document.getElementById("chat-box");
    chatBox.innerHTML = "";

    data.forEach((msg) => {
        const decryptedMessage = decryptMessage(msg.encrypted_message, msg.iv);
        const messageClass = msg.sender_id == senderId ? "message-sent" : "message-received";
        chatBox.innerHTML += `<div class="message ${messageClass}">${decryptedMessage}</div>`;
    });
}
```

Real-Time Updates: To keep the chat responsive, the application periodically fetches new messages every 2 seconds using setInterval



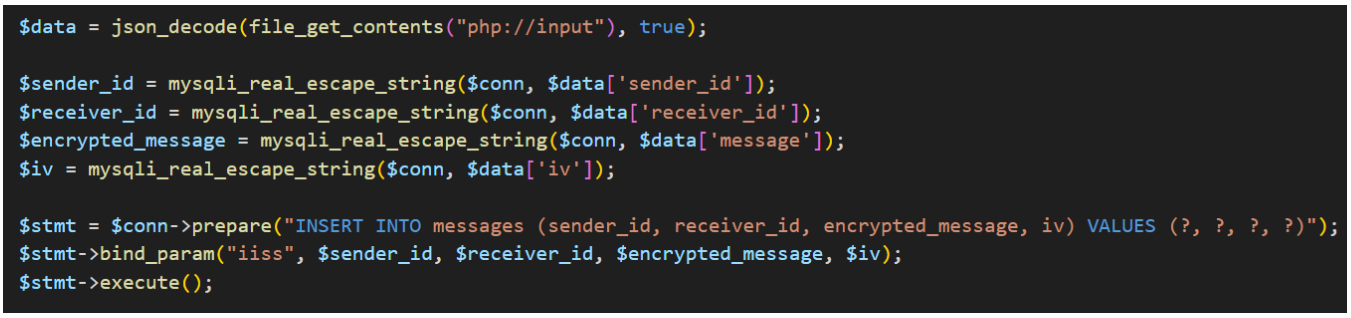
```javascript
setInterval(fetchMessages, 2000);
```

Message Storage and Retrieval

The messaging system securely stores and retrieves encrypted messages using two key server-side scripts: send_message.php and fetch_messages.php. These interact with the client-side logic in chat.js to ensure seamless and secure communication.

2. Message Storage: send_message.php

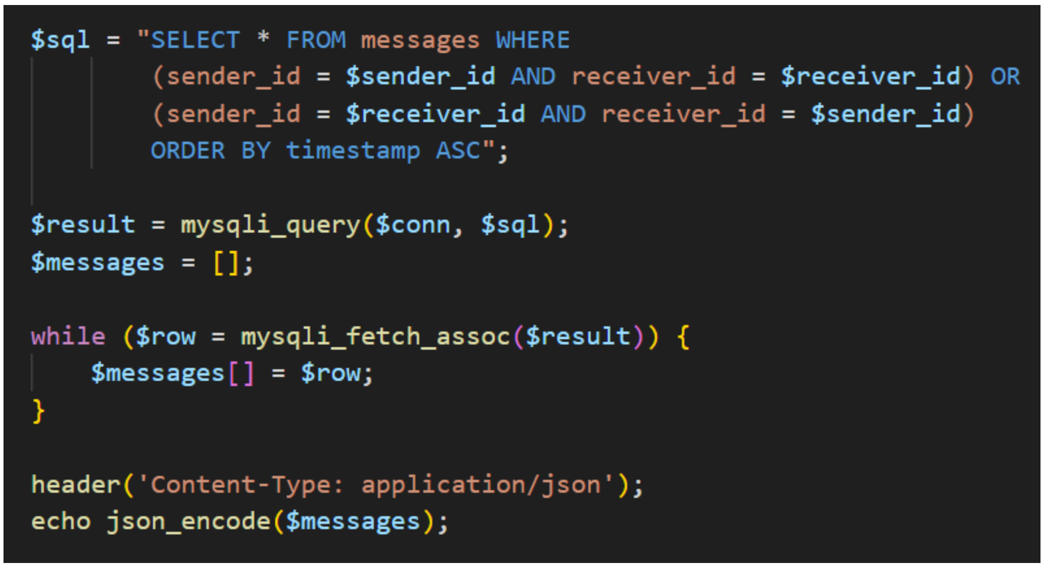
This script is responsible for saving messages securely in the database. It receives the encrypted message, IV, sender ID, and receiver ID from the client. The data is sanitized and stored in the database. By storing only the encrypted message and IV, the server ensures it has no access to the plaintext content, maintaining the end-to-end encryption setup.



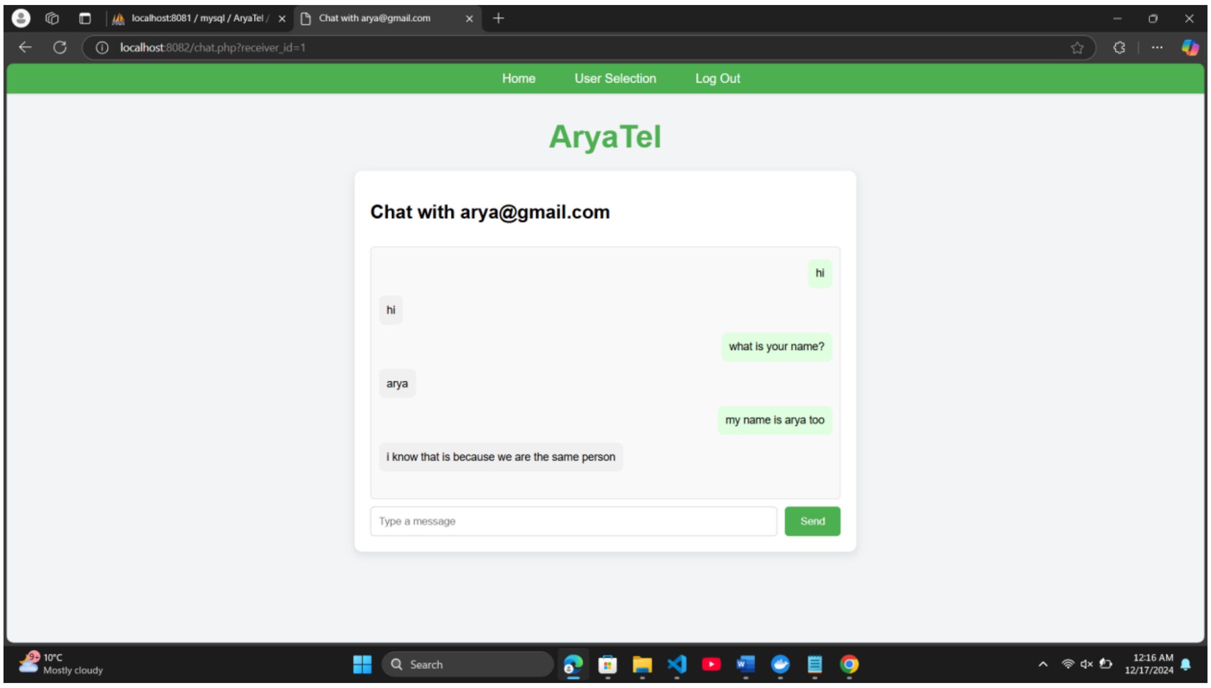
```php
$data = json_decode(file_get_contents("php://input"), true);

$sender_id = mysqli_real_escape_string($conn, $data['sender_id']);
$receiver_id = mysqli_real_escape_string($conn, $data['receiver_id']);
$encrypted_message = mysqli_real_escape_string($conn, $data['message']);
$iv = mysqli_real_escape_string($conn, $data['iv']);

$stmt = $conn->prepare("INSERT INTO messages (sender_id, receiver_id, encrypted_message, iv) VALUES (?, ?, ?, ?)");
$stmt->bind_param("iiss", $sender_id, $receiver_id, $encrypted_message, $iv);
$stmt->execute();
```

3. Message Retrieval: fetch_messages.php

This script retrieves encrypted messages from the database for a conversation. It uses the sender and receiver IDs to fetch messages exchanged between the two users, ordered by timestamp. The encrypted messages and their associated IVs are sent back to the client in JSON format.



```php
$sql = "SELECT * FROM messages WHERE
        (sender_id = $sender_id AND receiver_id = $receiver_id) OR
        (sender_id = $receiver_id AND receiver_id = $sender_id)
        ORDER BY timestamp ASC";

$result = mysqli_query($conn, $sql);
$messages = [];

while ($row = mysqli_fetch_assoc($result)) {
    $messages[] = $row;
}

header('Content-Type: application/json');
echo json_encode($messages);
```


localhost:8081 / mysql / AryaTel /
Chat with arya@gmail.com
localhost:8082/chat.php?receiver_id=1
Home
User Selection
Log Out

AryaTel

Chat with arya@gmail.com

hi

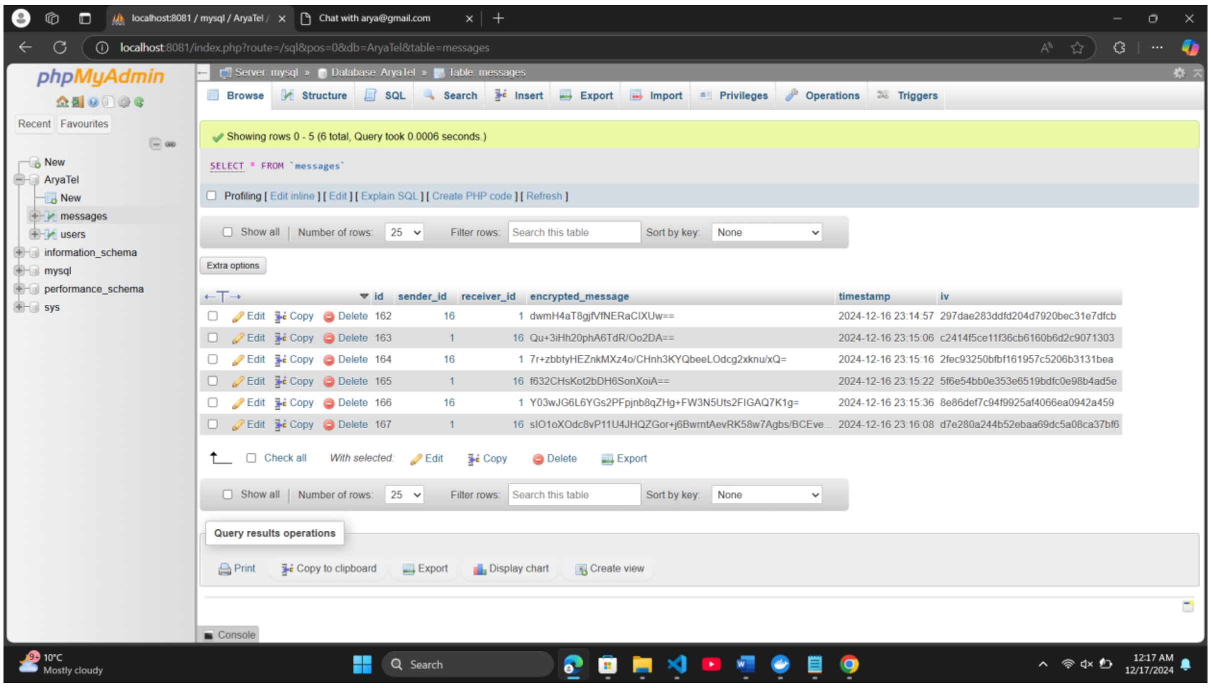
hi

what is your name?

arya

my name is arya too

i know that is because we are the same person


localhost:8081 / mysql / AryaTel /
Chat with arya@gmail.com
localhost:8081/index.php?route=/sql&pos=0&db=AryaTel&table=messages
phpMyAdmin
Recent
Favourites
New
AryaTel
New
messages
users
information_schema
mysql
performance_schema
sys
Server: mysql » Database: AryaTel » Table: messages
Browse
Structure
SQL
Search
Insert
Export
Import
Privileges
Operations
Triggers

Showing rows 0 - 5 (6 total, Query took 0.0006 seconds.)

```
SELECT * FROM `messages`
```

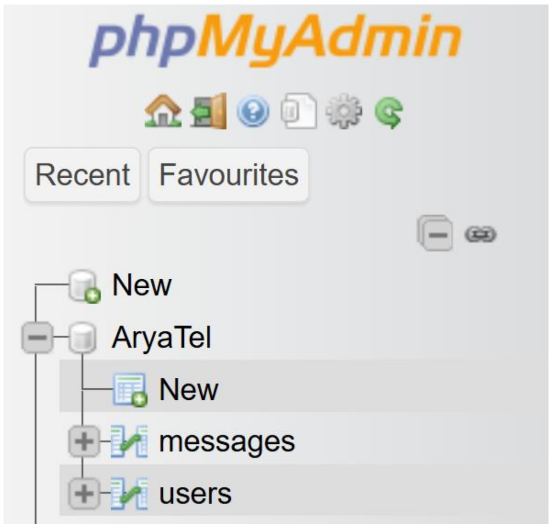
Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
Show all
Number of rows: 25
Filter rows: Search this table
Sort by key: None
Extra options

id	sender_id	receiver_id	encrypted_message	timestamp	iv
162	16	1	dwmH4aT8gjfVfNERaClXUw==	2024-12-16 23:14:57	297dae283ddfd204d7920bec31e7dfcb
163	1	16	Qu+3iHh20phA6TdR/Oo2DA==	2024-12-16 23:15:06	c2414f5ce11f36cb6160b6d2c9071303
164	16	1	7r+zbbtyHEZnkMXz4o/CHnh3KYQbeeLOdcg2xknu/xQ=	2024-12-16 23:15:16	2fec93250bfbf161957c5206b3131bea
165	1	16	f632CHsKot2bDH6SonXoiA==	2024-12-16 23:15:22	5f6e54bb0e353e6519bdfc0e98b4ad5e
166	16	1	Y03wJG6L6YGs2PFpjnb8qZHg+FW3N5Uts2FIGAQ7K1g=	2024-12-16 23:15:36	8e86def7c94f9925af4066ea0942a459
167	1	16	sIO1oXOdc8vP11U4JHQZGor+j6BwrntAevRK58w7Agbs/BCEve...	2024-12-16 23:16:08	d7e280a244b52ebaa69dc5a08ca37bf6

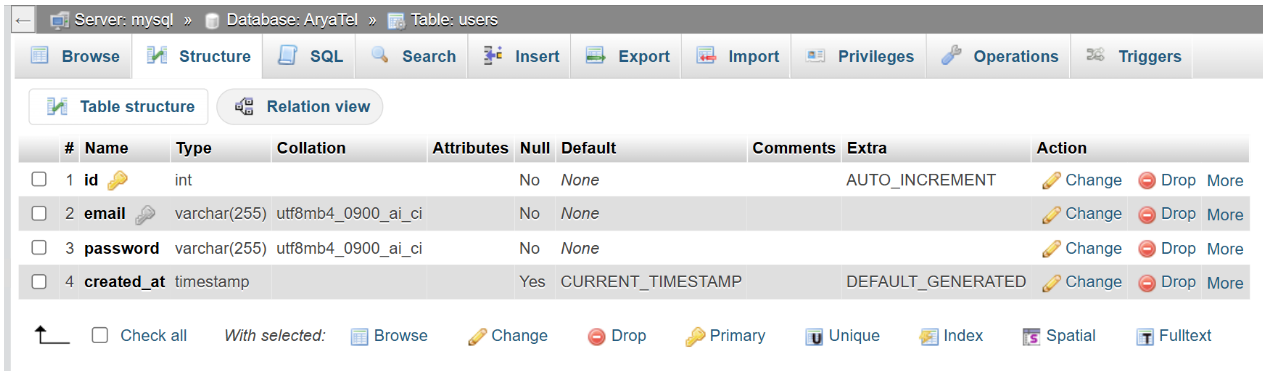
Database Design

I used Docker and phpMyAdmin to set up the database for my chat application for security and efficiency. Here's how each table contributes to the security and functionality of the app

In my project, I designed the database with two main tables: users and messages, which work together to manage users and their encrypted messages.

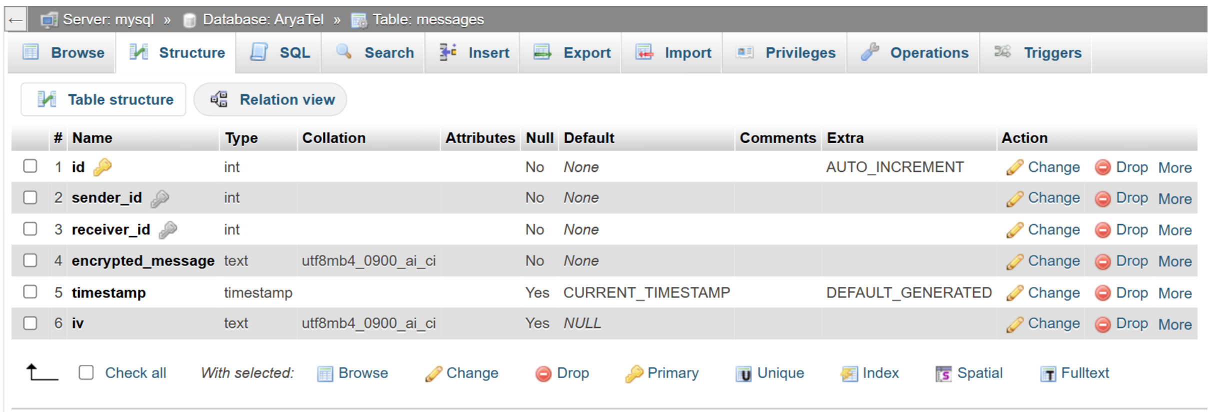


The users table is designed to securely store users data. It contains each user's ID, email, hashed password, and the registration timestamp. By hashing the passwords I ensure that plain text passwords are not stored so even if the database is compromised passwords remain protected.



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	**id**	int			No	*None*		AUTO_INCREMENT	Change Drop More
2	**email**	varchar(255)	utf8mb4_0900_ai_ci		No	*None*			Change Drop More
3	**password**	varchar(255)	utf8mb4_0900_ai_ci		No	*None*			Change Drop More
4	**created_at**	timestamp			Yes	CURRENT_TIMESTAMP		DEFAULT_GENERATED	Change Drop More

The messages table is where encrypted messages between users are stored. Each message has sender and receiver's IDs to identify who sent and received the message, the encrypted message text, and a timestamp for when the message was sent. This table ensures that messages are unreadable without decryption keeping conversations private and secure.


Server: mysql » Database: AryaTel » Table: messages
Browse Structure SQL Search Insert Export Import Privileges Operations Triggers
Table structure Relation view
Check all With selected: Browse Change Drop Primary Unique Index Spatial Fulltext

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int			No	*None*		AUTO_INCREMENT	Change Drop More
2	sender_id	int			No	*None*			Change Drop More
3	receiver_id	int			No	*None*			Change Drop More
4	encrypted_message	text	utf8mb4_0900_ai_ci		No	*None*			Change Drop More
5	timestamp	timestamp			Yes	CURRENT_TIMESTAMP		DEFAULT_GENERATED	Change Drop More
6	iv	text	utf8mb4_0900_ai_ci		Yes	*NULL*			Change Drop More

Together, these tables help me manage user accounts and securely store encrypted messages exchanged between users.

Overview of Security Measures

In my project, I implemented several security measures to ensure the privacy and integrity of user data and messages. The main focus was on achieving End-to-End Encryption and securing sensitive information both on the client and server sides.

Client-Side Encryption:

Messages are encrypted on the client side using AES encryption before being sent to the server. The encryption key is never transmitted, as it is generated locally using a combination of the sender's ID, receiver's ID, and the user's password. This ensures that only the sender and receiver can decrypt the messages.

Initialization Vector (IV):

A random IV is generated for each message, ensuring that even identical messages produce different ciphertexts. The IV is sent alongside the encrypted message and used for proper decryption on the client side.

Password Hashing:

User passwords are securely hashed and stored in the users table. This protects login credentials from being exposed in case of unauthorized access to the database.

Data Validation and Sanitization:

All inputs sent to the server are sanitized using mysqli_real_escape_string() and prepared statements to prevent SQL injection attacks.

Session Management:

User sessions are used to ensure that only authenticated users can access the chat system and interact with the database.

Conclusion

In this project, I built a secure real-time chat application with client-side end to end encryption. Messages are encrypted on the sender's device using AES encryption with a unique key derived from user IDs. The encryption key is never sent to the server, ensuring that only the sender and receiver can decrypt the messages.

Each message also uses a unique IV, making the encryption more secure. Only the encrypted message and IV are stored in the database, and the server has no access to the actual message content.

This approach ensures that the entire encryption and decryption process happens on the client side, keeping messages private and unreadable to anyone else, including the server. By combining secure encryption, session management, and efficient database design, I created a reliable and private communication platform.

References:

ADVANCED ENCRYPTION STANDARD (AES)

CRYPTOJS documentation

MDN WEB documentation (javascript fetch api)

PBKDF2 standard

PHP documentation

MySQL documentation

Java script documentation

phpMyAdmin documentation

Docker documentation