НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “ Комп’ютерні системи та мережі”
спеціальності 123 “ Комп’ютерна інженерія”

на тему: Веб-застосунок з підтримкою шифруваннння текстових повідомлень для безпечного спілкування

Виконала: студенка __4__ курсу, групи ІО-02
(шифр групи)

_____Негрич Ярослава Олегівна_____
(прізвище, ім’я, по батькові) (підпис)

Керівник ____асистент Гончаренко Олександр Олексійович_____
(посада, науковий ступінь, вчене звання, прізвище, ім’я, по батькові) (підпис)

Консультант (нормоконтроль) ст. викладач Виноградов Ю. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __доцент каф. ІСТ, ФІОТ, к.т.н., доцент, Шимкович В.М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Комп’ютерні системи та мережі”

спеціальності 123 “ Комп’ютерна інеженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2024 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Негрич Ярослави Олегівни

1. Тема проєкту *Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування*
керівник проєкту асистент Гончаренко О. О.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 27 травня 2024 року №2112-с

2. Термін здачі студентом закінченого проєкту *3 червня 2024 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані з питань проектування веб-застосунків.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Огляд алгоритмів шифрування та функціонування месенджера.

Огляд алгоритмів та технологій для розробки програми.

Деталі розробки програми.

Дослідження та аналіз розробленої програми.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) схема структурна, діаграма сутність-зв'язок (схема функціональна), алгоритм роботи системи (схема принципова).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Виноградов Ю. М.		

7. Дата видачі завдання «10» грудня 2023 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*25.02.2024-27.03.2024*	
2.	*Вивчення та аналіз завдання*	*1.04.2024-13.04.2024*	
3.	*Розробка архітектури та загальної структури системи*	*14.04.2024-21.04.2024*	
4.	*Розробка структур окремих підсистем*	*22.04.2024-3.05.2024*	
5.	*Програмна реалізація системи*	*4.05.2024-12.05.2024*	
6.	*Оформлення пояснювальної записки*	*13.05.2024-19.05.2024*	
7.	*Захист програмного продукту*	*25.05.2024*	
8.	*Передзахист*	*3.06.2024*	
9.	*Захист*	*17.06.2024*	

Студент-дипломник _____ НЕГРИЧ Ярослава
(підпис)

Керівник проєкту _____ ГОНЧАРЕНКО Олександр
(підпис)

АНОТАЦІЯ

Даний дипломний проект присвячений розробці веб-застосунку, який зробить комунікацію безпечною за допомогою шифрування текстових повідомлень. Під час дослідження були проаналізовані переваги та недоліки існуючих месенджерів та питання, пов’язані з конфіденційністю та захистом даних користувачів. На основі цього аналізу були сформовані думки щодо нового месенджера та обрані відповідні практики з безпеки спілкування. У роботі описано сучасні методи шифрування, як симетричне, так і асиметричне шифрування, а також протоколи обміну ключами. Було реалізовано сервіс для користувачів, де вони можуть безпечно обмінюватися повідомленнями, особливою функцією якого є фейковий пароль, який додає ще один рівень захисту в умовах вимагання доступу до приватних повідомлень.

Ключові слова: кінцеве шифрування, месенджер, безпека, секретні чати, конфіденційність даних, криптографія.

ANNOTATION

This diploma project is devoted to the development of a web application that will make communication secure using the encryption of text messages. During the study, the advantages and disadvantages of existing messengers and issues related to privacy and protection of user data were analyzed. Based on this analysis, opinions about the new messenger were formed and appropriate communication security practices were selected. This work describes modern methods of encryption, both symmetric and asymmetric encryption, as well as key exchange protocols. A service has been implemented for users where they can exchange messages securely, the special feature of which is a fake password, which adds another layer of protection in conditions of demanding access to private messages.

Key words: end-to-end encryption, messenger, security, secret chats, data privacy, cryptography.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Заново розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Веб-застосунок з	3		
			підтримкою шифрування			
			текстових повідомлень			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Веб-застосунок з	61		
			підтримкою шифрування			
			текстових повідомлень			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Е1***	Структура веб-застосунка з	1		
			підтримкою шифрування			
			текстових повідомлень			
			Схема структурна			
	А4	***ІАЛЦ.4672008.005 Д2***	Веб-застосунок з	1		
			підтримкою шифрування			
			текстових повідомлень			
			Діаграма сутність-зв’язок			
			Схема функціональна			
	А4	***ІАЛЦ.467200.006 Д3***	Веб-застосунок з	1		
			підтримкою шифрування			
			текстових повідомлень			
			Алгоритм роботи системи			
			Схема принципова			
	А4	***ІАЛЦ.467200.007 Д4***	Веб-застосунок з	31		
			підтримкою шифрування			
			текстових повідомлень			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	

ІАЛЦ.467200.001 ОА

Розроб		Негрич Я.О.			
Перев		Гончаренко О.О.			

Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування

Опис альбому

Літ.	Аркуш	Аркушів
	1	1

КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування*»

Київ – 2024

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Негрич Я. О.			**Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування** Технічне завдання	Літ.	Аркуш 1 / Аркушів 3
Перевірив		Гончаренко О.О.					
						КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02	
Н. Контр.		Виноградов Ю.М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку сервісу для безпечного обміну текстовими повідомленнями, а також його подальшого вдосконалення. Цей веб-застосунок призначений для використання у середовищах, де важлива конфіденційність та безпека спілкування. Це можуть бути бізнес-середовища, де необхідно захищати корпоративну інформацію, також особисте спілкування, де користувачі хочуть захистити свої приватні дані, а в умовах війни застосунок може бути особливо корисним для військових.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Цей проект виконується відповідно до вимог для здобуття ступеня бакалавра за спеціальністю 123 “Комп’ютерна інженерія” в рамках освітньо-професійної програми “Комп’ютерні системи та мережі”, затвердженої кафедрою Комп’ютерних систем та мереж Національного технічного університету України “Київський політехнічний інститут імені Ігоря Сікорського”.

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є створення веб-застосунку, який забезпечує безпечне спілкування за допомогою шифрування текстових повідомлень та надання додаткових заходів захисту через функцію фейкового паролю.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є науково-технічна література, офіційні документації, публікації та статті в мережі Інтернет на дану тему.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Інтуїтивно зрозумілий інтерфейс для користувачів.
- Забезпечення кінцевого шифрування текстових повідомлень.
- Реалізація функції фейкового паролю.
- Можливість реєстрації та аутентифікації користувачів.
- Надійне зберігання даних користувачів.

5.2. Вимоги до програмного забезпечення

- Операційна система: Windows 10, macOS або Linux.
- Інтегроване середовище розробки: Visual Studio 1.83.1 або новіша версія.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3-2100T.
- Зовнішня пам’ять не менше ніж 64 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	25.02.2024-27.03.2024
Вивчення та аналіз завдання	1.04.2024-13.04.2024
Розробка архітектури та загальної структури системи	14.04.2024-21.04.2024
Розробка структур окремих частин системи	22.04.2024-3.05.2024
Програмна реалізація системи	4.05.2024-12.05.2024
Виправлення помилок	13.05.2024-15.05.2024
Оформлення пояснювальної записки	15.05.2024-19.05.2024

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування*»

Київ – 2024

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	(Application Programming Interface) Інтерфейс програмного застосування
AES	(Advanced Encryption Standard) Стандарт передового шифрування
DOM	(Document Object Model) Об'єктна модель документа
ECDH	(Elliptic Curve Diffie-Hellman) Алгоритм обміну ключами на основі еліптичних кривих
E2E	(End-to-End) Кінцеве шифрування
HTTP	(HyperText Transfer Protocol) Протокол передачі гіпертексту
MVC	(Model-View-Controller) Модель-Вид-Контролер
JSON	(JavaScript Object Notation) Нотація об'єктів JavaScript
JWK	(JSON Web Key) Веб-ключ у форматі JSON
SHA	(Secure Hash Algorithm) Безпечний алгоритм хешування
XML	(Extensible Markup Language) Розширювана мова розмітки
W3C	(World Wide Web Consortium) Консорціум всесвітньої павутини

ВСТУП

У теперішньому світі технологій стандартом комунікації між людьми стали месенджери. Разом зі зручністю та легкістю передачі повідомлень через Інтернет постає важливе питання їх конфіденційності та захисту. Саме тому на сьогоднішній день є актуальною розробка безпечних сервісів для обміну повідомленнями.

Так як майже усі месенджери є централізованими, вся надіслана користувачем інформація зберігається на сервері, і відтак втрачає свою приватність. Однак, реалізуваши кінцеве шифрування ми можемо забезпечити недоступність наших розмов стороннім особам. Кожне відправлене повідомлення перетворюється в нечитабельний вигляд, ключ до розшифрування якого мають лише учасники чату. Це означає, що навіть якщо дані будуть перехоплені під час їх передачі через мережу, вони залишатимуться недосяжними для тих, хто не має відповідного ключа для розшифрування.

Також однією з популярних проблем є вимагання паролю. Особливо це актуально в періоди війни, коли є окупанти, які можуть вимагати пароль для доступу до конфіденційної інформації. Тому, щоб забезпечити вищий рівень захисту, доцільно розглянути функцію хибного паролю, введення якого може спричинити автоматичне знищення секретних даних.

Потенційним користувачем розробленого месенджеру є будь-яка людина, яка турбується про приватність та безпеку спілкування в мережі Інтернет.

РОЗДІЛ 1
ОГЛЯД ІСНУЮЧИХ МЕСЕНДЖЕРІВ З ШИФРУВАННЯМ

1.1 Месенджер Telegram

Telegram швидко здобув популярність та став улюбленим месенджером для багатьох завдяки своїй зручності да адаптивності. Однією з головних переваг є те, що користувач може увійти в свій обліковий запис з будь-якого гаджета без втрати інформації, оскільки всі фото, відео, текстові повідомлення зберігаються у надійному хмарному сховищі Telegram, і вся інформація є доступною для користувача в будь-який момент і з будь-якого пристою.

Проте безпека та конфіденційність інформації є однією з головних проблем величезної кількості месенджерів. У цьому відношенні Telegram вирішує її забезпеченням захищеної передачі даних на транспортному рівні, від клієнта до сервера. Це дозволяє уникнути ситуацій, пов’язаних з перехопленням даних. Однак Telegram не використовує шифрування повідомлень для звичайних чатів. Для того, щоб підвищити рівень безпеки та увімкнути наскрізне шифрування, потрібно активувати функцію секретного чату. Більше того, в такому чаті буде не доступно пересилання повідомленнь та скріншоти. Усі повідомлення шифруються на стороні клієнта, і не можуть бути розшифровані ніким, окрім адресата. Це забезпечує максимальну конфіденційність, оскільки ваші особисті дані захищені навіть від провайдера месенджера [1].

Telegram використовує протокол MTProto для шифрування даних, створений спеціально для месенджера. Він використовує симетричне шифрування для кодування повідомлень на стороні клієнта, і асиметричне для обміну ключами між клієнтом та сервером. MTProto також включає механізм автентифікації та гарантує цілісність даних у спосіб, який не дозволить

жодному повідомленню бути відкритим для змін або підробки в процесі передачі. Але про надійність цього протоколу і досі важко сказати, так як код MTProto не є відкритим, і не може бути повністю досліджений на виявлення проблем та вразливостей незалежними науковцями, тому часто через це Telegram піддається критиці [2].

1.2 Месенджер Signal

В наш час Signal є одним з найбільш захищених засобів спілкування. До 2020 року він не був популярним серед звичайних людей, однак у зв'язку з останніми подіями, в тому числі і війною, він набрав обертів і зараз налічує близько 40 мільйонів користувачів.

Він використовує власний протокол шифрування, який вважається одним з найбезпечніших серед усіх, саме через це він широко використовується і серед інших месенджерів. Не менш важливим є те, що Signal має відкритий вихідний код, тому його може перевірити будь-хто, хто хоче довести, що месенджер дійсно є безпечним і прозорим, і кожен може переконатися в рівні безпеки своїх особистих даних. Варто також відмітити, що Signal є безкоштовним, першочерговим завданням команди розробників є досягнення надійності та безпеки месенджера, а не переслідування комерційних цілей. Вони ставлять інтереси користувачів на перший план і відмовляються від прибутків, які були б отримані від використання даних у комерційних цілях.

Слід зазначити, що Signal не збирає персональні дані про користувача, його пристрій та інші метадані, все що потрібно це номер телефону, необхідний для реєстрації. В березні 2024 року це змінилося, адже Signal випустив оновлення, де замінили телефонні номери на звичайні псевдоніми. Це забезпечує найвищий рівень анонімності та відсутність відстеження активності користувача. Крім того, месенджер має багатий функціонал для захисту конфіденційності: він

може автоматично знищувати повідомлення через певний час, блокувати можливість зробити скріншот і багато чого іншого [3].

Для забезпечення безпеки Signal було розроблено багато функціональностей, однією з яких є генерація різного роду ключів:

- Довгострокова пара ідентифікаційних ключів – для ідентифікації користувачів у системі та захисту зв’язку протягом тривалого часу.
- Середньострокова попередньо підписана пара ключів – для підписання повідомлень і перевірки їх автентичності. Такі ключі призначені для того, щоб перевірити, чи дійсним був відправник, який підписав повідомлення, і чи не було воно підписане вже на шляху до отримувача.
- Тимчасова пара попередніх ключів – генеруються при відправці для кожного повідомлення та, у свою чергу, використовуються лише один раз. Вони допомагають додати ще один рівень захисту.

Щодо алгоритмів, то Signal поєднує Double Ratchet для управління ключами, triple Elliptic-Curve Diffie-Hellman handshake протокол, еліптичну криву Curve25519, алгоритм AES-256, а також хеш-алгоритм HMAC-SHA256 [4].

1.3 Месенджер Threema

Threema – це продукт швейцарської компанії, відомої своїм підходом до безпеки. Не дарма ця країна заборонила використовувати своїм військовим всі месенджери, окрім Threema. Хоч на сьогоднішній день цей сервіс не надто популярний, але це не зменшує його потенціалу конфіденційності. Особливість полягає в тому, що месенджер не вимагає від вас жодної особистої інформації. Буде надано абсолютно випадковий ідентифікатор, тому всім користувачам гарантовано анонімність і особисту безпеку [5].

Ще одна перевага Threema полягає в тому, що всі дані зберігаються на пристрої користувача. Звичайно, завжди існує ризик втрати такого пристрою, а разом з ним і дані, але Threema має рішення для цього: Threema Safe. Він пропонує можливість резервного копіювання даних на сервер, обраний користувачами. Це дає надію на безпеку даних у непередбачуваній ситуації [6].

Варто відмітити, що Threema є проектом з відкритим вихідним кодом, а це означає, що кожен зацікавлений може вільно оцінювати надане програмне забезпечення з точки зору стабільності та безпеки. Програма також використовує наскрізне шифрування будь-яких даних і передачі за допомогою криптобібліотеки NaCl.

1.4 Порівняльний аналіз

Розглянувши описані вище месенджери, можна зробити висновок, що усі з них мають зручний та інтуїтивно зрозумілий інтерфейс, крім того, на високому рівні є їх швидкість та продуктивність, що є важливим для користувача. Signal та Telegram є безкоштовними, Threema – платний. Проте, оцінюючи їх особливості та рівень безпеки, можна знайти відмінності, які будуть важливішими для клієнта [7]. У табл. 1.1 проведемо стислий порівняльний аналіз критеріїв, які мають найбільше значення для користувачів.

Таблиця 1.1. – Порівняльний аналіз месенджерів

Критерій	Telegram	Signal	Threema
Кінцеве шифрування	Лише для секретних чатів	Всі чати	Всі чати
Збір метаданих	Так	Ні	Ні

Кінець таблиці 1.1

Критерій	Telegram	Signal	Threema
Відкритість коду програмного забезпечення	Приватно	Відкрито	Відкрито
Збереження інформації на сервері	Так	Ні	За бажанням користувача

ВИСНОВОК ДО РОЗДІЛУ 1

У цьому розділі було проведено детальний аналіз трьох популярних месенджерів, які використовують шифрування: Telegram, Signal та Threema. Незважаючи на відмінності, всі вони мають спільну мету – забезпечення високого рівня конфіденційності та безпеки комунікацій. Особливу увагу було приділено механізмам, завдяки яким ці платформи забезпечують безпеку даних користувачів. Кожне з розглянутих рішень має свої унікальні підходи для її забезпечення, відмінності в архітектурі та реалізації криптографічних алгоритмів. Telegram, Signal і Threema працюють на різних протоколах шифрування, кожен із яких має свої особливості щодо обробки та зберігання даних, а також відрізняються за рівнем зручності та можливостями, наприклад, анонімність, групові чати, хмарне сховище.

Ключовими концепціями, які ми хотіли б зберегти у нашому веб-застосунку, є наскрізне шифрування, впровадження сучасних сучасних криптографічних алгоритмів для захисту даних користувачів, надання інтуїтивно зрозумілого і простого інтерфейсу для кінцевих користувачів і додаткові, інноваційні функції для підвищення рівня безпеки.

РОЗДІЛ 2
ОГЛЯД ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ РОЗРОБКИ ТА ТЕХНОЛОГІЙ

Вивчаючи існуючі рішення, ми прийшли до висновку, що месенджер повинен містити пару основних функцій. По-перше, інтерфейс користувача має бути зручним у користуванні, що з першого погляду привабить користувача. По-друге, для забезпечення приватності та конфіденційності користувачів обов'язковим є наскрізне шифрування для всіх чатів. Ще однією особливістю, яка може підвищити безпеку месенджера, є реалізація захисту інформації. Цього можна досягти, наприклад, ввівши фальшивий пароль. Ця функція може мати велике значення, оскільки вона дозволить уникати ситуацій, коли вимагають пароль, і зберігатиме особисті повідомлення користувачів у безпеці. Загальний опис того, як розроблено месенджер, дозволить створити продукт, який буде надійним, зручним і безпечним для користувача.

2.1 Огляд технологій для реалізації поставленої задачі

Про розглянуті месенджери слід сказати, що вони були розроблені за допомогою багатьох відомих мов програмування, серед яких C++, Java, Python, C, Swift та ін. Для себе я обрала мову написання JavaScript з використанням додаткових технологій та бібліотек, а для серверної частини коду – Node.js. Зберігання даних реалізувати в MongoDB за допомогою NoSQL, який відомий своєю швидкістю та гнучкістю роботи з даними. Щоб забезпечити миттєвість надходження повідомлень, використаю бібліотеку для розробки додатків в режимі реального часу – Socket.IO. Все це допоможе створити ефективний та масштабований месенджер, який задовольнить потреби користувачів у зручному та безпечному способі спілкування.

2.1.1 React.js

React – це бібліотека для створення користувацьких інтерфейсів, яка насамперед популярна через свою динамічність. Одна з найбільш привабливих розробок у React – можливість створювати невеликі незалежні компоненти – є однією з найважливіших. Це значно спрощує процес розробки та налагодження. В цих функціональних компонентах ми можемо використовувати так звані «хуки», така функція з'явилася відносно нещодавано, що дозволило відійти від використання класів та зручно управляти локальним становищем кожного компоненту окремо.

Серед інших переваг React – використання віртуального DOM. Це копія реального DOM, який присутній у пам'яті. React порівнює реальний DOM з віртуальним і знаходить різницю між ними, а потім відображає її на веб-сайті [8]. Таким чином, React мінімізує кількість операцій над справжньою моделлю, тому веб-додаток працює дуже швидко, особливо це буде корисно для нашої розробки месенджера, адже при зміні якогось компонента, наприклад при надходженні нового повідомлення, будуть змінюватися лише залежні від нього компоненти.

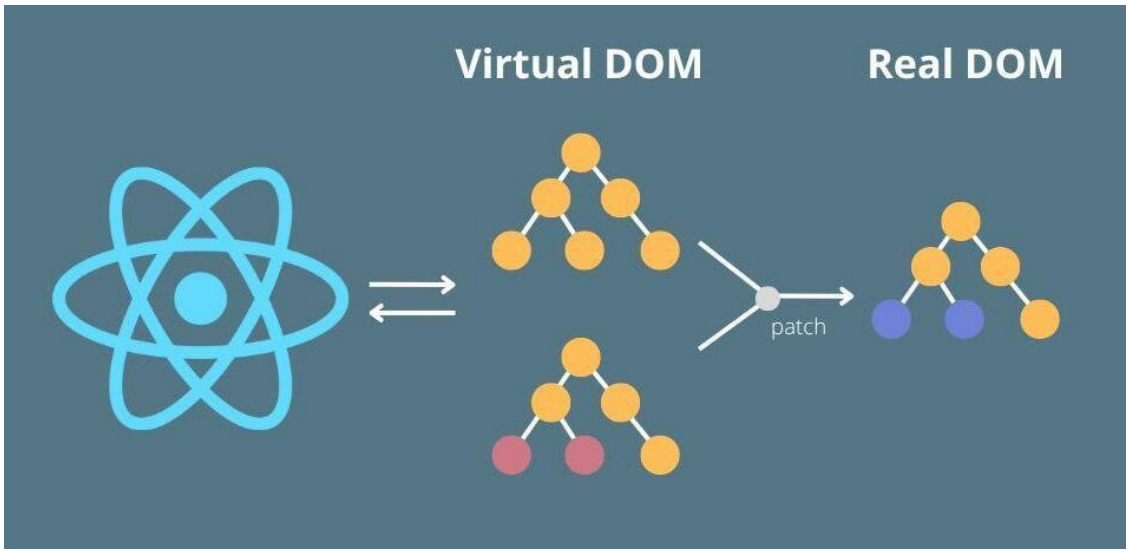


Рисунок 2.1. – Робота Virtual DOM в React

Файли компонентів мають розширення JSX – це спеціальний синтаксис JavaScript, який дозволяє писати HTML-подібний код у поєднанні з JavaScript. Цей факт робить код React більш читабельним і приємним для розробників, оскільки вони працюють над користувацькими інтерфейсами за допомогою синтаксису HTML, але в той же час можуть використовувати JavaScript у повній мірі.

Отож, React має кілька переваг, таких як висока продуктивність, швидка реакція на зміни стану, легке налагодження, розробка та розгортання, а також велика спільнота, де можна знайти відповіді на будь-які питання.

2.1.2 Node.js

Node.js – це середовище виконання коду, написане на мові JavaScript і створене на движку Chrome V8. Перевагою є те, що він дозволяє запускати JavaScript на стороні сервера, що робить його чи не найкращим інструментом для повномаштабної розробки веб-додатків. Ключова перевага Node.js полягає в тому, що можна писати як і клієнтський, так і серверний код на одній мові програмування. Це безумовно полегшує завдання для розробника, адже зазвичай програміст повинен був знати як мінімум дві мови програмування для написання фронтенду та бекенду.

Однією з найперспективніших сфер Node.js є розробка додатків у реальному часі, таких як чати, спільне редагування документів, що власне і влучно підходить для нашого завдання. Це відбувається через те, що Node.js має асинхронну природу, і таким чином може обробляти багато одночасних з’єднань, не блокуючи потоки [9].

Саме тому він і є популярним вибором для серверної частини програми, так як багато клієнтів роблять запити на сервери і чекають від них на відповідь.

Основна відмінність Node.js від інших серверних платформ полягає в тому, що під час роботи з клієнтом він не створює новий потік для нового з'єднання, а використовує один головний, який обробляє події в асинхронному стилі, що є дуже ефективним та маштабованим.

2.1.3 MongoDB

MongoDB – це база даних на основі NoSQL для зберігання документів. Розробнику легко уявити саме сховище для даних, адже вони збегрігаються там у вигляді об'єктів JavaScript. Великою перевагою MongoDB є те, що не потрібно суворо дотримуватися схеми, можна зберігати дані гнучким способом і легко змінювати структуру, не переробляючи всю базу даних [10].

У MongoDB колекція – це і є сховище даних, яке можна розглядати як таблицю в звичайній реляційній базі даних. Документи в колекції є окремими об'єктами JavaScript. Наприклад, при розробці месенджера наша база даних буде містити три колекції: для зберігання користувачів, чатів та окремих повідомлень. Створювати структуру документів доволі легко – за допомогою схем. Хоч це і не є обов'язковим, але дозволяє організувати всі дані, полегшує валідацію та в загальному роботу.

Також MongoDB дуже легко поєднується у використанні з Node.js, завдяки бібліотеці Mongoose. Інсталюючи її, ви спрощуєте взаємодію з базою даних, оскільки з'являються широкі можливості, за допомогою її інтерфейсу можна визначати моделі, використовувати вже написані методи для роботи з данми та багато чого іншого. Наприклад, метод *find()* є одним із найпопулярніших у Mongoose, який дозволяє пошук даних в базі за певними критеріями. Не менш корисними є *save()* і *update()* – для збереження та оновлення документів відповідно [11].

У MongoDB кожному документу автоматично призначається унікальне поле *_id* з метою ідентифікації документа. За замовчуванням поле представлено об’єктом типу ObjectId. Що робить розподілене сховище корисним, так це те, що MongoDB може добре масштабуватися в горизонтальному напрямку. Він легко справляється з точки зору продуктивності з величезними обсягами даних і одночасними підключеннями.

2.1.4 Socket.io

Socket.IO – це бібліотека JavaScript, яка працює на основі веб-сокетів і дозволяє двосторонній зв’язок у реальному часі між клієнтом і сервером. По суті, з її використанням розробники можуть створювати потужні та ефективні веб-додатки, які надсилатимуть і отримуватимуть дані без необхідності перезавантажувати веб-сторінку.

Коли клієнт запитує щось із сервера за допомогою Socket.IO, він відкриває постійне з’єднання з сервером. Таким чином, сервер може надсилати оновлення всім підключеним клієнтам, не роблячи жодних запитів від клієнта. Також можна і обмежити надсиланнння, вибравши певний критерій [12].

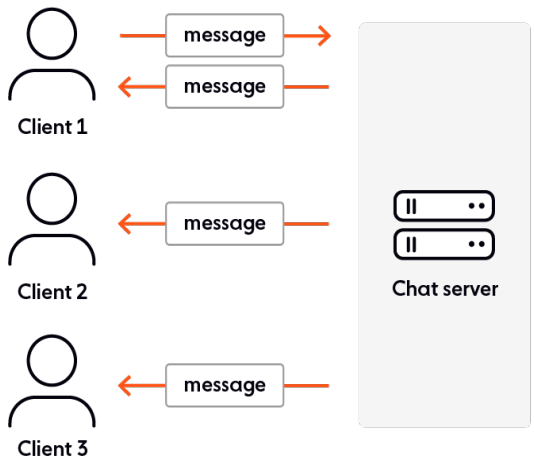


Рисунок 2.2. – Робота Socket.IO

Найважливішими з усіх методів, які пропонує Socket.IO, є *emit()* для надсилання повідомлення від клієнта до сервера або навпаки, і *on()* – для отримання повідомлень від протилежної сторони.

Socket.IO часто використовують для реалізації месенджерів через асинхронність та потужну архітектуру, завдяки чому можна підтримувати багато клієнтів та легко маштабувати додаток. І, завдяки функції кластеризації, можна запускати програми на різних серверах одночасно. Таким чином, навантаження розподілятиметься між різними серверами, що забезпечує ще більшу стійкість до високих обсягів запитів.

2.2 Кінцеве шифрування

Ідея E2E шифрування побудована максимально на принципі абсолютної конфіденційності та захисту даних від будь-якого стороннього впливу. Вся ця технологія використовується для того, щоб розшифровану інформацію могли переглядати лише відправник і одержувач, навіть сам сервер, як посередник, не може отримати доступ до вмісту повідомлень [13]. В перекладі це шифрування звучить як «end to end», де вже сама назва говорить нам про цю логіку. Існує два види шифрування – симетричне та асиметричне, які забезпечують обмін даними між відправником та отримувачем [14].

2.2.1 Симетричне шифрування

Симетричне шифрування є дуже ефективним і швидким методом у світі криптографії та захисту даних. Проста ідея полягає в тому, що той самий ключ використовується як для шифрування, так і розшифрування даних. Таким чином, це робить процес обробки простим і ефективним, оскільки немає необхідності мати складні алгоритми для керування ключами [15].

Однак варто зазначити, що ця техніка обмежена рівнем безпеки, адже цей ключ повинен бути переданий співрозмовнику для майбутнього розшифрування, що робить його вразливим до перехоплення. Якщо зловмиснику вдасться отримати такий ключ, він зможе легко здобути всю інформацію [16]. Отож, для симетричного ключа потрібно вживати заходи, які посилять його безпеку, наприклад, шифрування за допомогою іншого симетричного ключа або обміну ключами за допомогою асиметричного шифрування.

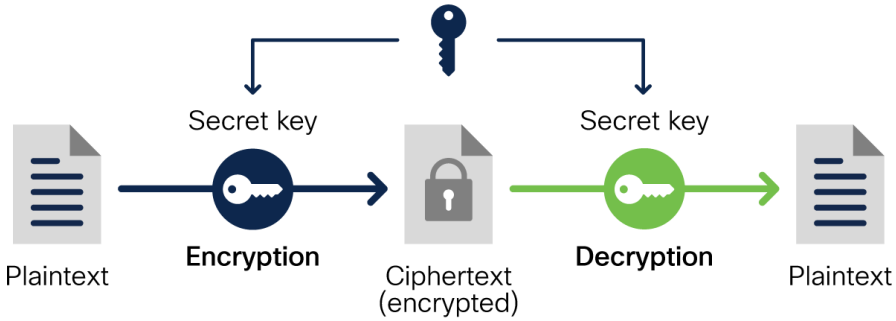


Рисунок 2.3. – Схема синхронного шифрування

2.2.2 Асиметричне шифрування

Найважливішим моментом асиметричного шифрування є наявність різних типів ключів для кожного користувача – публічного і приватного [17]. Публічний ключ працює над процесом шифрування даних перед надсиланням, таким чином забезпечуючи безпечну передачу. Такі ключі є загальнодоступними та зберігаються в базі даних. А ось приватний ключ є високозахищеним і не повинен бути відкритий нікому, крім самого власника, і використовується для розшифровки.

Принцип дії такий: коли відправник надсилає якусь інформацію, він шифрує її за допомогою публічного ключа одержувача. Таким чином, створює безпечний канал, через який передається інформація. Надійшовше повідомлення може розшифрувати одержувач, лише якщо він має свій

приватний ключ, адже це пов’язана частина пари ключів [18]. Така процедура шифрування забезпечує підвищену конфіденційність і безпеку даних, що вкрай необхідно для будь-якого сучасного месенджера.

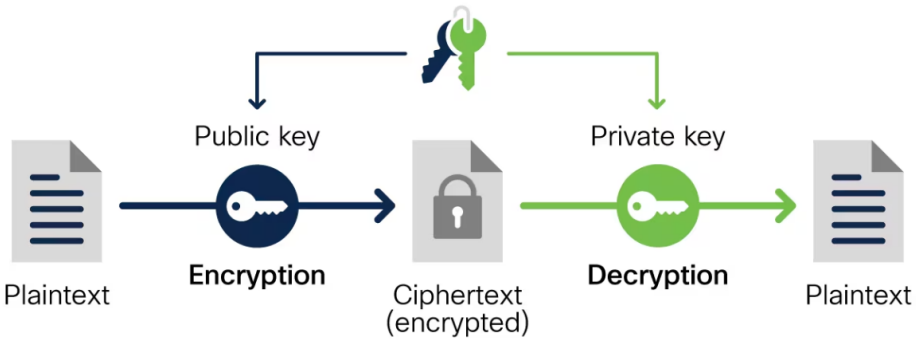


Рисунок 2.4. – Схема асинхронного шифрування

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі ми детально розглянули інструментальні засоби та технології, які використовуються у розробці нашого веб-застосунку.

React було обрано для розробки інтерфейсу користувача завдяки її потужним можливостям створення динамічних і швидких веб-додатків, а її компонентна архітерктура є легкою для сприйняття і написання коду. Для серверної частини використовуватиметься Node.js, який також пишеться на JavaScript і спрощує процес розробки. MongoDB, в якості сховища, обрана через свою гнучкість та можливість масштабування. Ця NoSQL база даних ідеально підходить для зберігання структурованих та неструктурованих даних, які використовуються в нашому додатку. Socket.io є невід'ємною частиною в розробці будь-якого месенджера, адже саме він забезпечує миттєву взаємодію між користувачами і разом з цим зручність у спілкуванні.

Окремо ми дослідили та розглянули методи кінцевого шифрування: симетричне та асиметричне, які забезпечують конфіденційність при передачі та збереженні повідомлень, що передаються через наш застосунок.

Використання цих інструментів та технологій у поєднанні з відповідними підходами до розробки дозволить нам створити веб-застосунок, який буде відповідати вимогам безпеки, продуктивності та зручності в користуванні.

РОЗДІЛ 3
ОПИС АРХІТЕКТУРИ ПРОЕКТУ

У цьому розділі перейдемо до самої розробки захищеного месенджера, використовуючи всі техніки та знання, викладені в попередніх розділах.

3.1 Компоненти системи та їх взаємодія

Цей підрозділ зосереджений на описі архітектурних технологій та принципів, які лягли в основу розробки месенджера. Розглянемо принцип розподіленої системи, на якому базується архітектура месенджера, технологію REST API, яка є ключовою при взємодії клієнтської та серверної частини, і патерн, на основі якого організований код.

3.1.1 Принципи архітектури клієнт-сервер

Клієнт-серверна архітектура на сьогодні є найпопулярнішою в розробці програмного забезпечення. Використання цього принципу розділяє функціональні можливості між клієнтською та серверною частинами програми, надаючи контроль над ресурсами та інтерфейсом з користувачем [19].

На стороні клієнта користувач може отримати доступ до взаємодії системи через інтерфейс, представлений у нашому випадку у вигляді веб-сторінки. Додаток здатний відправляти різноманітні запити у відповідь на дії користувача, наприкад, натиск на кнопку, та отримувати відповіді безпосередньо на своїх пристроях, що гарантує користувачам швидку та зручну роботу з програмою.

Серверна частина програми піклується про обробку цих запитів, взаємодію з базою даних і надання правильних відповідей клієнту. Вона

спілкується з клієнтською стороною за допомогою різних протоколів, таких як HTTP або WebSocket. У свою чергу, база даних зберігає і впорядковує дані, і також забезпечує надійне та ефективне зберігання інформації, тому програма працюватиме швидко та надійно.

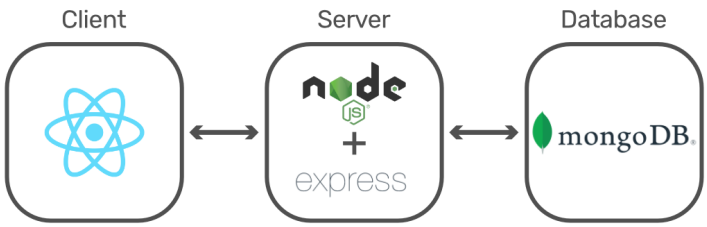


Рисунок 3.1. – Клієнт-серверна модель архітектури

Це дуже масштабована та ефективна архітектура, оскільки часто великі компанії приміняють кластеризацію, щоб розподіляти навантаження між декількома серверами, відтак запити розподіляються між ними і клієнт впевнено та швидко отримає свою відповідь. Також це зменшує ризик збою системи, оскільки при відмові одного сервера, інші підтримують виконання роботи [20].

3.1.2 REST API

REST API в оригіналі розшифровується як Representational State Transfer Application Programming Interface. Це стиль архітектури програмного забезпечення, який використовується для веб-додатків. REST API базується на принципах протоколу HTTP і використовує різні методи у взаємодії з сервером.

Основні методи REST API включають:

- GET – використовується для отримання даних із сервера. Наприклад, отримати список усіх користувачів або повідомлень.
- POST – використовується для створення нових даних на сервері. Наприклад, при надсиланні нового повідомлення або створенні нового користувача.

- PUT – використовується для оновлення існуючих даних на сервері. Наприклад, редагування інформації про користувача.
- DELETE – використовується для видалення чогось на сервері, наприклад користувача або повідомлення.

У коді, наведеному нижче, визначаються маршрути, які вказують на відповідні контролери для обробки запитів. Як бачимо, тут використовуються методи GET отримання всіх повідомлень та POST для створення нового.

```
const express = require('express');
const router = express.Router();
const messageController = require('../controllers/message');

router.post('/send/:id', messageController.sendMessage);
router.get('/:id', messageController.getMessages);

module.exports = router;
```

Рисунок 3.2. – Використання REST API для взаємодії з клієнтом

За допомогою REST API інформація передається в різних форматах даних: наприклад, JSON, HTML, XML або навіть звичайний текст [21]. JSON означає JavaScript Object Notation – це легкий і зручний формат обміну даними, його використовують найчастіше, тому він був обраний для проекту. Відповіді від сервера будуть у формі JSON і включатимуть інформацію про те, чи був запит успішним чи ні. Відповідь вважається успішною, якщо вона містить дані, які запитує клієнт і має статус 2xx. У випадку, якщо запит клієнта не вдається, повертається повідомлення про помилку з відповідним кодом статусу HTTP – 4xx при помилці з клієнтської частини або 5xx при помилці з серверної частини.

3.1.3 Використання MVC

Враховуючи складність проекту, було вирішено примінити архітектурний шаблон Model-View-Controller, щоб розробити структуровану, легко змінювану та підтримувану систему. MVC ділить програму на три

компоненти: модель, представлення та контролер. Модель відповідає за дані та взаємодіє з базою даних. Представлення відповідає за інтерфейс та відображення інформації користувачу. Контролер відповідає за обробку запитів від користувача, взаємодіє з моделлю, щоб подати необхідні дані для відображення [22].

Одним із головних принципів такого способу роботи є розподіл обов'язків: кожен компонент виконує свій функціонал і, таким чином, ви можете ефективно розподіляти завдання між командою розробників, легко розширювати чи змінювати функціонал.

У нашому проекті створені окремі папки для кожного компоненту: в папці *controllers* знаходиться код, що пов'язаний з обробкою запитів, у папці *models* – схеми Mongoose для опису структури даних, які зберігаються у нашій базі даних MongoDB, а за *views* відповідають React компоненти, які надають користрувачу html код для відображення інтерфейсу.

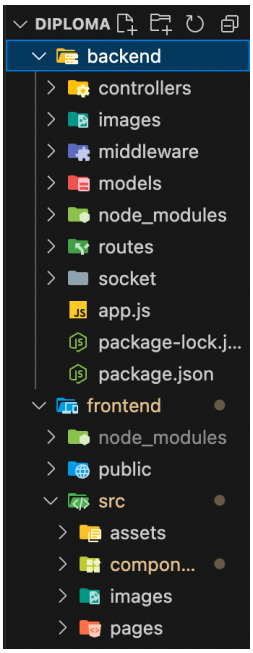


Рисунок 3.3. – Структура проєкту

Розглянемо шматок коду, який представляє роботу контролера, та на його прикладі розберемо взаємодію з іншими компонентами:

```
exports.signup = async (req, res, next) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      const error = new Error('Wrong input!');
      error.statusCode = 422;
      error.data = errors.array();
      throw error;
    }

    const username = req.body.username;
    const password = req.body.password;
    const email = req.body.email;
    const confirmedPassword = req.body.confirmedPassword;

    if (password !== confirmedPassword) {
      const error = new Error('Passwords should match!');
      error.statusCode = 400;
      throw error;
    }
    const hashedPassword = await bcrypt.hash(password, 12);

    const user = new User({
      username,
      password: hashedPassword,
      email,
    });

    const result = await user.save();
    res.status(201).json({ result });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
```

Рисунок 3.4. – Контролер для обробки реєстрації

Спочатку контролер отримує від клієнта дані про нового користувача з req.body. І після хешування паролю контролер починає взаємодію з моделлю User, таким чином зберігаючи новий документ в базі даних. Отриманий результат надсилається назад до клієнта, через що той бачить зміни в своєму інтерфейсі.

3.2 Огляд бази даних

Реалізація месенджера передбачає використання бази даних, тож було створено три основні схеми: «users», яка містить інформацію про користувачів, «messages» для збереження зашифрованих повідомлень та «chats» для бесід.

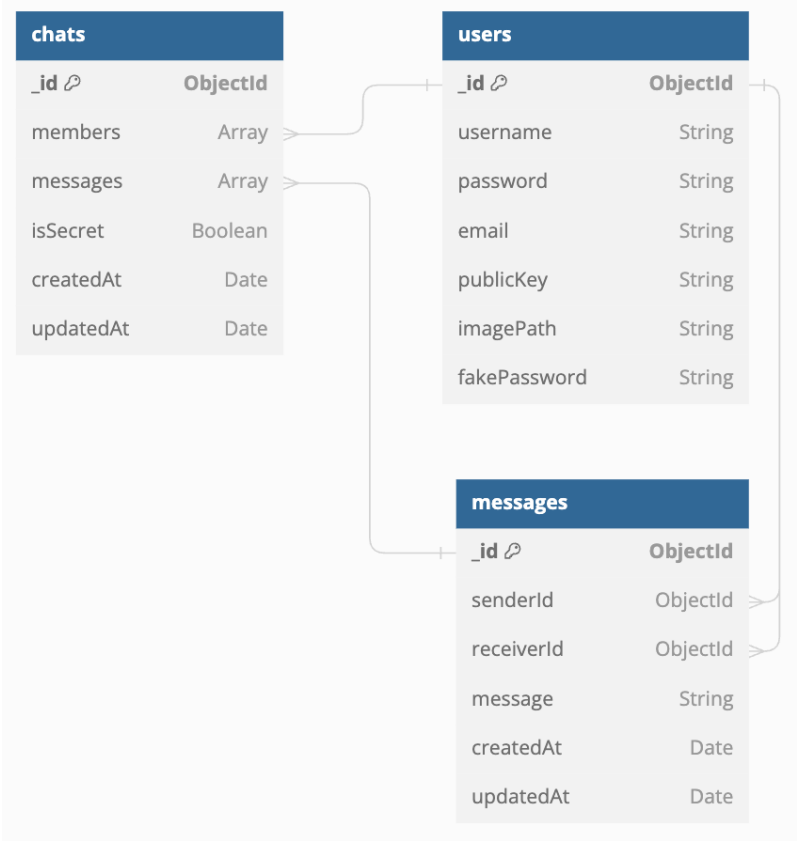


Рисунок 3.5. – ER діаграма

На рисунку 3.5 представлена діаграма, яка складається з ряду сутностей і зв’язків між ними, що відображає основну структуру наших даних. Для кращого розуміння, яке місце мають ці схеми, наведемо короткий опис полів:

Таблиця 3.1. – Опис позначень полів в схемах бази даних

Схема «users»	
_id	Ідентифікатор користувача, створенний автоматично, є унікальним
username	Ім’я або псевдонім

Кінець таблиці 3.1

password	Пароль, зберігається зашифрованим
email	Електронна адреса
publicKey	Публічний ключ із згенерованої зв’язки, призначений для шифрування повідомлень
imagePath	Шлях до зображення профілю користувача
fakePassword	Опціональне поле, пароль для захисту від несанкціонованого доступу
Схема «messages»	
_id	Ідентифікатор повідомлення, створениий автоматично, є унікальним
senderId	Ідентифікатор користувача-відправника
receiverId	Ідентифікатор користувача-отримувача
message	Зашифрований вміст повідомлення
createdAt, updatedAt	Час створення або редагування повідомлення, створений за допомогою timestamps
Схема «chats»	
_id	Ідентифікатор бесіди, створениий автоматично, є унікальним
members	Список ідентифікаторів користувачів, які беруть участь у чаті
messages	Список ідентифікаторів повідомлень, які створені між учасниками чату
isSecret	Додаткове поле для позначення чату конфіденційним
createdAt, updatedAt	Час створення або редагування чату, створений за допомогою timestamps

3.3 Реалізація кінцевого шифрування

Для реалізації наскрізного шифрування мною було обрано Web Crypto API, який є вбудованим і визначає стандартний набір інтерфейсів веб-браузера для доступу до криптографічних операцій, таких як генерація ключів, шифрування даних, дешифрування даних. Будучи частиною HTML5, він дозволяє веб-розробнику виконувати багато криптографічних процесів безпосередньо у своєму веб-додатку. Цей інтерфейс був реалізований з використанням сучасних крипто-алгоритмів і стандартів безпеки, таких як Advanced Encryption Standard (AES), RSA, Elliptic Curve Cryptography (ECC) і SHA, і розроблений на основі відкритих стандартів та є частиною специфікацій W3C [23]. Завдяки Web Crypto API можна зручно зберігати інформацію в зашифрованому форматі на стороні клієнта, що і потрібно в нашому випадку кінцевого шифрування і безумовно зробить додаток більш безпечним. Також великою перевагою є те, що Web Crypto API робить додаток швидшим і легшим, оскільки не потрібно використовувати сторонні бібліотеки, а також його можна без проблем використовувати у більшості сучасних веб-браузерах, лише частково в IE.

3.3.1 Генерування ключів

Для кожного користувача ми повинні створити пару асиметричних ключів: відкритий та закритий ключ. Для цього реалізуємо функцію *generateKeyPair()*, у якій викликаємо метод *window.crypto.subtle.generateKey()*. Параметрами для *generateKey()* є алгоритм, чи може цей ключ бути експортований та для чого він буде використаний [24].

Серед багатьох відомих алгоритмів, таких як HMAC, AES-GCM, RSA-OAEP, ECDH, я обрала саме ECDH (еліптична крива Діффі-Геллмана), оскільки

він має кращу продуктивність і забезпечує високий рівень безпеки. У контексті еліптичних кривих, для генерації ключів використовується точка на кривій, яку ще називають генератор. Коли два користувачі використовують алгоритм ECDH, вони використовують свої приватні ключі і цю основну точку для обчислення відповідних публічних ключів [25]. Після цього вони обмінюються своїми публічними ключами і використовують їх для обчислення спільного секретного ключа, який і буде надалі використаний для шифрування і розшифрування повідомлень.

При виборі цієї еліптичної кривої P-384 для нашої системи вже встановлено високий ступінь безпеки та ефективності. Крім того, ми визначили, що в поточному алгоритмі буде дозволений експорт ключів, щоб ключі могли використовуватись і передаватись між користувачами. І як останній параметр ми вказали в списку deriveKey, це означає, що ключі будуть використовуватися для отримання спільного похідного ключа, який буде застосовуватися обома користувачами [26].

```
export async function generateKeyPair() {
  const keys = await window.crypto.subtle.generateKey(
    {
      name: 'ECDH',
      namedCurve: 'P-384',
    },
    true,
    ['deriveKey']
  );
```

Рисунок 3.6. – Функція для генерування ключів

Після того, як наші ключі згенеровані, нам потрібно експортувати їх за допомогою методу *exportKey()*. Цей метод відповідає за перетворення об'єкта CryptoKey в зручний і зрозумілий для нас формат. Представлені такі формати на вибір: raw, pkcs8, spki і jwk. У нашому випадку оберемо формат JSON Web Key (JWK). JWK – це стандартний метод представлення ключів у структурі JSON, який дозволяє зберігати ключі у формі, зручній для спільного

використання, формат JWK також спрощує процес обміну ключами і легко сприймається людиною.

```
const publicKeyJwk = await window.crypto.subtle.exportKey(
  'jwk',
  keys.publicKey
);

const privateKeyJwk = await window.crypto.subtle.exportKey(
  'jwk',
  keys.privateKey
);
```

Рисунок 3.7. – Експорт ключів у форматі JWK

Цю логіку генерування ключів ми застосуємо при вході користувача в його акаунт. Створений публічний ключ зберігаємо в базі даних разом зі всією наявною інформацією про користувача. Приватний ключ не повинен бути доступним ні для кого, окрім самого власника ключа, тому його потрібно зберігати лише в локальній пам’яті пристрою користувача, в нашому випадку це буде локальне сховище браузера.

```
let privateKeyJwk;
let publicKeyJwk;
try {
  privateKeyJwk = getPrivateKey();
  publicKeyJwk = getPublicKey();
  if (!privateKeyJwk || !publicKeyJwk) {
    ({ publicKeyJwk, privateKeyJwk } = await generateKeyPair());
    console.log(publicKeyJwk, privateKeyJwk);
  }
} catch (error) {
  console.log(error);
}

const response = await fetch('http://localhost:3000/auth/login', {
  method: request.method,
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    submitData,
    publicKeyJwk: JSON.stringify(publicKeyJwk),
  }),
});

if (!response.ok) {
  return response;
}

const resultData = await response.json();
const { userId, token, isFake } = resultData;

storeData(userId, token);
localStorage.setItem('privateKey', JSON.stringify(privateKeyJwk));
localStorage.setItem('publicKey', JSON.stringify(publicKeyJwk));
```

Рисунок 3.8. – Генерування та збереженння ключів користувача

3.3.2 Створення спільного ключа

Перейдемо до процесу отримання спільного секретного ключа між двома користувачами. Спочатку імпортуємо їх ключі, щоб отримати об'єкти CryptoKey, які потрібні нам для подальших операцій.

Після цього викликаємо метод *window.crypto.subtle.deriveKey()*, який і слугує для створення нового спільного ключа з використанням інформації з існуючих зв’язок ключів користувачів [27]. Цей метод вимагає такі параметри:

1. *algorithm* для виведення ключа – вибираємо той самий ECDH, який потребує ще публічний ключ отримувача повідомленння, він є доступним і може бути отриманий з бази даних.
2. *baseKey* – ключ, з якого буде здійснюватися похідний процес. У нашому випадку при алгоритмі ECDH це приватний ключ власника повідомлення [28].
3. *derivedKeyAlgorithm* – алгоритм, який буде задіяний при використанні цього ключа. Був обраний алгоритм AES-GCM з довжиною 128, так як серед інших у нього є одна велика перевага – перевірка, чи не було повідомленння зміненим на шляху до одержувача.
4. *extractable* – true, оскільки ми будемо витягувати отриманий ключ для захисту повідомлень.
5. *keyUsages* – масив зі списком дозволених дій з отриманим ключем. У нашому випадку це ['encrypt', 'decrypt'], оскільки отриманий ключ буде використовуватися безпосередньо для шифрування та розшифрування повідомлень [29].

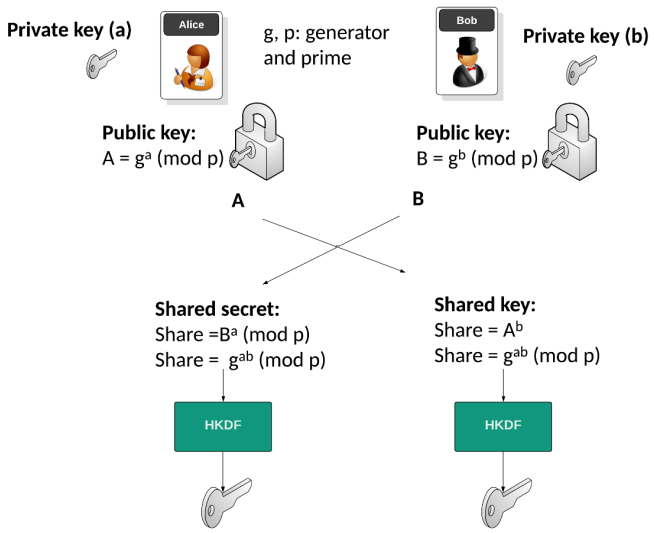


Рисунок 3.9. – Принцип створення спільного ключа за алгоритмом ECDH [25]

Результатом є спільний секретний ключ, абсолютно однаковий для обох учасників, який вони використовуватимуть для шифрування та дешифрування повідомлень між собою, забезпечуючи конфіденційність і безпеку спілкування [25, 30].

```
export async function generateSharedKey(publicKeyJwk, privateKeyJwk) {
  const publicKey = await window.crypto.subtle.importKey(
    'jwk',
    publicKeyJwk,
    {
      name: 'ECDH',
      namedCurve: 'P-384',
    },
    true,
    []
  );

  const privateKey = await window.crypto.subtle.importKey(
    'jwk',
    privateKeyJwk,
    {
      name: 'ECDH',
      namedCurve: 'P-384',
    },
    true,
    ['deriveKey']
  );

  return await window.crypto.subtle.deriveKey(
    { name: 'ECDH', public: publicKey },
    privateKey,
    { name: 'AES-GCM', length: 128 },
    true,
    ['encrypt', 'decrypt']
  );
}
```

Рисунок 3.10. – Функція для виведення спільного ключа

Виклик функції *generateSharedKey()* відбувається при виборі співрозмовника, спочатку ми надсилаємо запит до сервера задля отримання публічного ключа цього користувача, і тоді з наявною інформацію про публічний ключ отримувача повідомлення і власний приватний ключ, передаємо їх як парметри для функції і утворюємо спільний [31].

```
async function getSharedKey() {
  try {
    if (receiverPublicKey && privateKey) {
      const sharedKey = await generateSharedKey(
        receiverPublicKey,
        privateKey
      );
      setSharedKey(sharedKey);
    }
  } catch (error) {
    console.log(error);
  }
}
getSharedKey();
```

Рисунок 3.11. – Отримання спільного ключа

3.3.3 Шифрування повідомленння

Тепер, маючи спільний ключ, можемо перейти до самого шифрування повідомлення, для цього створимо функцію *encryptMessage()*. Спочатку перетворимо текстове повідомленння у байтовий масив, використовуючи *TextEncoder()* та його метод *encode()*. І далі виконуємо саме шифрування методом *encrypt()*, де параметрами є алгоритм, ключ для шифрування та текст. Так як ми будемо використовувати один і той самий ключ як для шифрування, так і розшифрування повідомлення, то потрібно обрати симетричний алгоритм [32].

Одним з найпопулярніших варіантів є AES (Advanced Encryption Standard) в режимі Galois/Counter. Основний його принцип полягає в тому, що вхідне повідомлення розбивається на блоки фіксованого розміру (у нашому випадку 128 біт), і кожен з цих блоків обробляється окремо. Як вже згадувалося,

вибір GCM є обґрунтованим, оскільки дані можуть бути перевірені на цілісність та автентичність за допомогою наявного при GCM тегу автентифікації після процесу дешифрування [33]. Використовуючи цей режим, також потрібно вказати вектор ініціалізації як випадковий байтовий масив, який додається до даних перед шифруванням, щоб запобігти відомим атакам, таким як known-plaintext attacks і chosen-plaintext attacks.

І наприкінці перетворюємо повідомлення у рядок за допомогою *String.fromCharCode.apply(null, encryptedBytes)* і у послідовність ASCII-символів за допомогою *btoa()*, для коректної передачі закодованого повідомлення через мережу.

```
export async function encryptMessage(message, sharedKey) {
  const encodedMessage = new TextEncoder().encode(message);
  const encryptedMessage = await window.crypto.subtle.encrypt(
    { name: 'AES-GCM', iv: new Uint8Array(13) },
    sharedKey,
    encodedMessage
  );

  const encryptedBytes = new Uint8Array(encryptedMessage);
  const encryptedText = String.fromCharCode.apply(null, encryptedBytes);
  const encryptedBase64 = btoa(encryptedText);

  return encryptedBase64;
}
```

Рисунок 3.12. – Функція для шифрування текстового повідомлення

Ось як ми використовуємо функцію *encryptMessage()* при відправці кожного повідомлення, зберігаючи в базу даних вже зашифроване повідомлення:

```
async function sendMessageHandler() {
  const encryptedMessage = await encryptMessage(message, sharedKey);
  const response = await fetch(
    `http://localhost:3000/message/send/${ctx.activeUser._id}`,
    {
      method: 'POST',
      headers: {
        Authorization: `Bearer ${token}`,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ message: encryptedMessage }),
    }
  );
  const answer = await response.json();
  onSend({ ...answer.newMessage, message: message });
  setMessage('');
}
```

Рисунок 3.13. – Шифрування повідомлення з використанням спільного ключа

3.3.4 Розшифрування повідомленння

Останньою функцією, яка запезпечує конфіденційність інформації, є *decryptMessage()*, яка слугує для розшифровки повідомлення і робить дії, протилежні шифруванню. Спочатку зашифроване повідомлення, яке витягується з бази даних у вигляді рядка base64, декодується з використанням функції *atob()* і конвертується в масив байтів. Далі викликаємо метод *window.crypto.subtle.decrypt()*, використовуючи як параметри той самий алгоритм AES-GCM і вектор ініціалзації [34]. І на кінець, отримані розшифровані байти конвертуємо назад у рядок, отримуючи оригінальний текст повідомлення. Якщо розшифрування буде невдалим, то замість повідомлення користувач отримає текст помилки.

```
export async function decryptMessage(message, sharedKey) {
  try {
    const decryptedBase64 = atob(message);
    const decryptedBytes = new Uint8Array(
      [...decryptedBase64].map((char) => char.charCodeAt(0))
    );
    const decryptedMessage = await window.crypto.subtle.decrypt(
      {
        name: 'AES-GCM',
        iv: new Uint8Array(13),
      },
      sharedKey,
      decryptedBytes
    );

    const decodedMessage = new TextDecoder().decode(decryptedMessage);

    return decodedMessage;
  } catch (error) {
    console.log(error);
    return `Decryption failed. The key may be incompatible.`;
  }
}
```

Рисунок 3.14. – Функція для дешифрування текстового повідомлення

Цю функцію викликаємо для кожного повідомлення між учасниками при виборі бесіди.

3.4 Безпека та основні функції

3.4.1 Реєстрація та автентифікація

Як і для більшості застосунків, щоб користуватися ним, потрібно створити акаунт. Для того, щоб зареєструватися у месенджері, потрібно ввести електронну адресу, псевдонім, пароль і також його підтвердження. Всі дані проходять валідацію на клієнтській стороні, однак це не значить, що не повинна здійснюватися перевірка на коректність введених даних на стороні сервера, адже шахраї можуть змінити певні параметри валідації через інструменти розробника на веб-сторінці і таким чином створити акаунт з неправильними даними. Тому на сервері ми перевіряємо введені користувачем дані, а також чи не повторюється електронна адреса з наявними у базі даних, використовуючи бібліотеку *express-validator*. Якщо валідація пройдена і помилок немає, можемо додати новий акаунт до бази даних.

```
router.post(
  '/signup',
  [
    body('email')
      .isEmail()
      .withMessage('Invalid email!')
      .custom((value) => {
        return User.findOne({ email: value }).then((user) => {
          if (user) {
            return Promise.reject('User with this email already exists!');
          }
        });
      })
      .normalizeEmail(),
    body('password').trim().isLength({ min: 8 }),
    body('username').trim().not().isEmpty(),
  ],
  authController.signup
);
```

Рисунок 3.15. – Валідація введених користувачем даних на сервері

Щоб забезпечити недоступність до акаунту користувача третіми особами, потрібно зберігати пароль у безпечному форматі. Це достатньо легко зробити за допомогою бібліотеки *bcrypt,* яка надає інструменти для хешування паролів. Для обробки паролю викликаємо функцію *bcrypt.hash(password, 12)*, де

password – оригінальний пароль користувача, 12 – кількість ітерацій, які будуть використані для генерації хешу, це число є оптимальним значенням між продуктивністю і стійкістю паролю до brute force attacks [35]. Тепер користувач вважається зареєстрованим і перенаправляється на сторінку входу.

Переходимо до автентифікації, де користувачу потрібно ввести електронну адресу та пароль до його акаунту. Якщо надана адреса наявна в колекції users, перевіряємо відповідність введеного паролю з паролем у підходящому документі, за допомогою *bcrypt.compare(password, user.password).* Якщо такий користувач дійсно існує і автентифікація пройшла успішно, потрібно створити токен, який потім можна передавати при кожному запиті для перевірки авторизації та доступу до захищених ресурсів.

Популярним варіантом рішення цієї задачі в Node.js є бібліотека *jsonwebtoken*, де можна створити токен можна за допомогою методу *sign()*, передаючи вибрану інформацію про користувача (в нашому випадку електронну адресу та унікальний ідентифікатор) , секретний ключ для підпису, який забезпечує захист і цілісність інформації, та параметри терміну дії (в даному випадку 7 днів) [36].

```
const token = jwt.sign(
  {
    email: updatedUser.email,
    userId: updatedUser._id.toString(),
  },
  SECRET_KEY,
  { expiresIn: '7d' }
);
```

Рисунок 3.16. – Створення токену

Після входу користувача, токен та ідентифікатор повертаються як відповідь до клієнта, ці дані ми зберігаємо в локальному сховищі. По проходженню 7-ми днів, токен автоматично стає недійсним, і коли клієнтська сторона спробує використати цей токен для доступу до захищених ресурсів, сервер поверне відповідь з помилкою неавторизованого доступу. Тобто клієнту потрібно буде авторизуватися знову. Для правильної обробки завершення дії

токену, збережемо цей час на стороні клієнта в локальному сховищі, що дорівнює 7*24*60*60*1000 мілісекунд. Тепер, при спробі отримати доступ до сторінок, для яких потрібна автентифікація, спочатку буде перевірено, чи користувач має токен і чи не сплив його термін дії, у випадку чого буде перенаправлений на сторінку входу.

```
export function getDuration() {
  const expiration = localStorage.getItem('tokenExpiration');
  const expirationDate = new Date(expiration);
  const now = new Date();
  const duration = expirationDate.getTime() - now.getTime();
  return duration;
}

Codeium: Refactor | Explain | Generate JSDoc | ×
export function getToken() {
  const token = localStorage.getItem('token');
  if (!token) {
    return null;
  }
  const duration = getDuration();
  if (duration < 0) {
    return 'Token is expired';
  }
  return token;
}

Codeium: Refactor | Explain | Generate JSDoc | ×
export function checkAuth() {
  const token = getToken();
  if (!token || token === 'Token is expired') {
    return redirect('/auth/login');
  }
  return null;
}
```

Рисунок 3.17. – Перевірка наявності токена та терміну його дії

Для захисту певних маршрутів, які вимагають, щоб користувач був автентифікованим, потрібно створити проміжний middleware, який буде перевіряти наявність токену та підтверджувати його цілісність. В наведеному на рисунку 3.18 коді спочатку перевіряємо, чи містить запит заголовок ‘Authorization’, і якщо ні, то генеруємо помилку статусу 401 – Unauthorized. Якщо ж він все таки присутній, дістаємо з нього сам рядок з токеном, розділяючи по пробілу і відкидаючи перший елемент Bearer. Тепер потрібно перевірити токен, розшифровуємо його за допомогою функції *verify()* з використанням того ж самого секретного ключа, з яким він і створювався. Якщо

токен успішно розшифровано, результат зберігається в *decodedToken*, де також наявна інформація про ідентифікованого користувача. Наприкінці додаємо до запиту id користувача, та викликаємо функцію *next()*, яка передає керування наступному маршруту, зберігаючи при цьому інформацію про користувача в самому запиті.

```
exports.isAuth = (req, res, next) => {
  const authHeader = req.get('Authorization');
  if (!authHeader) {
    const error = new Error('No Authorization Header!');
    error.statusCode = 401;
    throw error;
  }
  const token = authHeader.split(' ')[1];
  let decodedToken;
  try {
    decodedToken = jwt.verify(token, SECRET_KEY);
  } catch (error) {
    error.statusCode = 500;
    throw error;
  }

  if (!decodedToken) {
    const error = new Error('Not authenticated!');
    error.statusCode = 401;
    throw error;
  }
  req.userId = decodedToken.userId;
  next();
};
```

Рисунок 3.18. – Middleware для перевірки автентифікації

Відповідно відправка будь-якого запиту авторизованого користувача з фронтенду вимагає встановлення заголовка ‘Authorization’ таким чином: *headers: { Authorization: ‘Bearer ${token}’}.*

Цей auth middleware тепер буде використаний перед маршутами до messageRoutes та userRoutes, обмежуючи доступ неавтентифікованим користувачам до головних сторінок з користувачами та повідомленнями.

3.4.2 Використання фейкового паролю для захисту від вимагання та несанкціонованого доступу

Безперечно, у месенджерах зберігається велика кількість секретної інформації, тому забезпечення безпеки особистих повідомлень є одним з найважливіших аспектів розробки. Для цього в нас вже реалізоване кінцеве шифрування повідомлень, однак трапляються ситуації, коли під тиском користувачі змушені розкрити свій пароль. Одним з методів захисту від вимагання та несанкціонованого доступу є використання фейкового паролю.

Фейковий пароль – це спеціально створений заздалегідь пароль, який користувач може використати у випадку примусу і вимаганння надати доступ до свого облікового запису. При його введені відбувається видалення або приховування певних даних, замість надання доступу до справжнього вмісту акаунту. Таким чином вимагач думає, що отримав доступ до всієї інформації користувача, тоді як насправді він бачить лише неповну інформацію, те, що ми не вважаємо конфіденційним або хочемо, щоб він побачив. Реалізація цієї функціональності не вимагає значних змін в архітектурі веб-застосунку, додамо її як додаткову функцію безпеки.

Користувач має змогу встановити хибний пароль на сторінці редагування свого профілю, і він буде збережений у відповідному форматі в базі даних з іншою інформацією про нього.

```
if (fakePassword && fakePassword.trim().length < 8) {
  const error = new Error('You should set valid fake password!');
  error.statusCode = 401;
  throw error;
} else if (fakePassword && fakePassword.trim().length >= 8) {
  const fakePasswordIsEqual = await bcrypt.compare(
    fakePassword,
    user.password
  );
  if (fakePasswordIsEqual) {
    const error = new Error('Fake password should be unique!');
    error.statusCode = 401;
    throw error;
  }
  const hashedFakePassword = await bcrypt.hash(fakePassword, 12);
  user.fakePassword = hashedFakePassword;
}
const result = await user.save();
```

Рисунок 3.19. – Обробка встановлення хибного паролю

Звісно нам потрібен якийсь механізм для позначення чатів конфіденційнми, які не повинні бути викриті вимагачами. Для цього, на головній сторінці інтерфейсу, де є блок інформації про активний чат, ми додамо іконку відкритого замку, при кліку на яку відкриватиметься модальне вікно для підтвердження зробити цей чат секретним та застереження, що при вході в акаунт з фейковим паролем всі повідомлення цього діалогу будуть видалені. Після підтвердження позитивною відповіддю іконка змінюється на закритий замок, цим самим сповіщаючи і співбесідника про те, що діалог захищений. У відповідь на цей перехід буде відправлятись POST запит на бекенд, де поле *isSecret* в колекції *chats* для конкретних *members* буде зміненим відповідно до дії користувача на true або false (за замовчуванням встановлене в false). Маршрут для запиту має динамічний параметр *${ctx.activeUser._id}*, який позначає користувача з вибраного чату. За допомогою нього на сервері можна буде отримати id користувача з *req.params*.

```
async function postIsSecret(isSecretBoolean) {
  const response = await fetch(
    `http://localhost:3000/users/setChatIsSecret/${ctx.activeUser._id}`,
    {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({ isSecret: isSecretBoolean }),
    }
  );
  const data = await response.json();
  setIsSecret(data.isSecret);
}
```

Рисунок 3.20. – Запит для зміни чату зі звичайного на секретний

```
exports.setChatIsSecret = async (req, res, next) => {
  try {
    const senderId = req.userId;
    const receiverId = req.params.receiverId;
    const isSecret = req.body.isSecret;
    const chat = await Chat.findOne({
      members: { $all: [senderId, receiverId] },
    });
    if (!chat) {
      const error = new Error('No chat found!');
      error.statusCode = 401;
      throw error;
    }
    chat.isSecret = isSecret;
    const result = await chat.save();

    res.status(200).json({ isSecret: result.isSecret });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};
```

Рисунок 3.21. – Обробка запиту зміни режиму чата

Тепер, коли користувач потрапляє у ситуацію вимагання, він може надати зловмиснику фейковий пароль замість справжнього. Акаунт користувача має виглядати таким чином, щоб вимагач нічого не запідозрив, тому окрім того, що всі повідомлення чатів, які позначені секретними, видаляються, головна сторінка при такому вході не містить в інтерфейсі іконок для зміни режиму чата, а також при редагуванні користувача схована можливість додати фейковий пароль та його відображення взагалі.

При спробі входу в акаунт знаходимо користувача з відповідною електронною адресою, і якщо він існує, спочатку йде порівняння введеного паролю з фейковим. Якщо є співпадіння, ми знаходимо всі чати в базі даних, в яких бере участь даний користувач, і видаляємо повідомлення в них. Далі генеруємо токен і зберігаємо публічний ключ, як і при звичайному вході, щоб все виглядало справжнім. Однак до відповіді клієнтській стороні додається ключ isFake, значенням якого є true, і дозволить нам відповідно змінити інтерфейс.

Якщо ж був введний не фейковий пароль, порівнюємо його зі справжнім паролем користувача, і продовжуємо звичайний процес автентифікації.

```
exports.login = async (req, res, next) => {
  try {
    const email = req.body.submitData.email;
    const password = req.body.submitData.password;
    const publicKey = req.body.publicKeyJwk;
    const user = await User.findOne({ email: email });
    let fakePasswordIsEqual = undefined;

    if (!user) {
      const error = new Error("User with this email doesn't exists.");
      error.statusCode = 404;
      throw error;
    }
    if (user.fakePassword) {
      fakePasswordIsEqual = await bcrypt.compare(password, user.fakePassword);
    }

    if (fakePasswordIsEqual) {
      const secretChats = await Chat.find({
        members: user._id,
        isSecret: true,
      });
      const deleteRes = await Promise.all(
        secretChats.map((chat) => chat.deleteOne())
      );
      const token = await handlePublicKeyAndToken(user, publicKey);

      res.status(200).json({ token: token, userId: user._id, isFake: true });
    }

    const passwordIsEqual = await bcrypt.compare(password, user.password);
    if (!passwordIsEqual) {
      const error = new Error('Wrong password!');
      error.statusCode = 401;
      throw error;
    }
    const token = await handlePublicKeyAndToken(user, publicKey);

    res.status(200).json({ token: token, userId: user._id, isFake: false });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};
```

Рисунок 3.22. – Логіка контролера при вході в акаунт

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі була розглянута загальна архітектура проекту, яка базується на принципі клієнт-сервер. Щоб запезпечити між ними взаємодію, було використано REST API через звичайні HTTP запити. Ми також застосували патерн MVC, який сприяє такому важливому аспекту як чистий код.

На основі колекцій бази даних було побудовано діаграму сутностей і зв’язків, дивлячись на яку, можна зрозуміти логіку структурування інформації. Для реалізації кінцевого шифрування було обрано Web Crypto API, так як цей спосіб дозволяє використати спеціальні методи для генерації ключів, створення спільного, а також шифрування і розшифрування тексту.

Також було продемонстровано основні функції, які забезпечують доступ користувача до акаунта на тривалий час та захищають певні ресурси від відвідування неавторизованими користувачами. Важливою деталлю в цій роботі також стала розробка функціональності фальшивого паролю, яка включає створення секретних чатів і вхід в акаунт з підробленим паролем.

В цілому, ця структура проекту забезпечує конфіденційність інформації та зручність в користуванні, що і було нашим завданням, і також готова до можливого подальшого розвитку.

РОЗДІЛ 4
АНАЛІЗ РОЗРОБЛЕНОГО ПРОЕКТУ

Розглянувши реалізацію месенджера, продемонструємо основні функціональні можливості проекту, взаємодію користувача із системою, а також порівняємо отриманий результат з існуючими рішеннями.

4.1 Демонстрація роботи проекту

4.1.1 Вхід до системи

Перший крок для кожного користувача – це реєстрація нового облікового запису, в процесі якої передбачена валідація введених даних. Якщо хоч одне з полів заповнено неправильно, відповідні границі стануть червоними, і обліковий запис не буде створено. Наприклад, це може відбутися, якщо електронна адреса не має належного вигляду, не введене валідне ім’я, занадто короткий пароль або невідповідний повтор паролю.

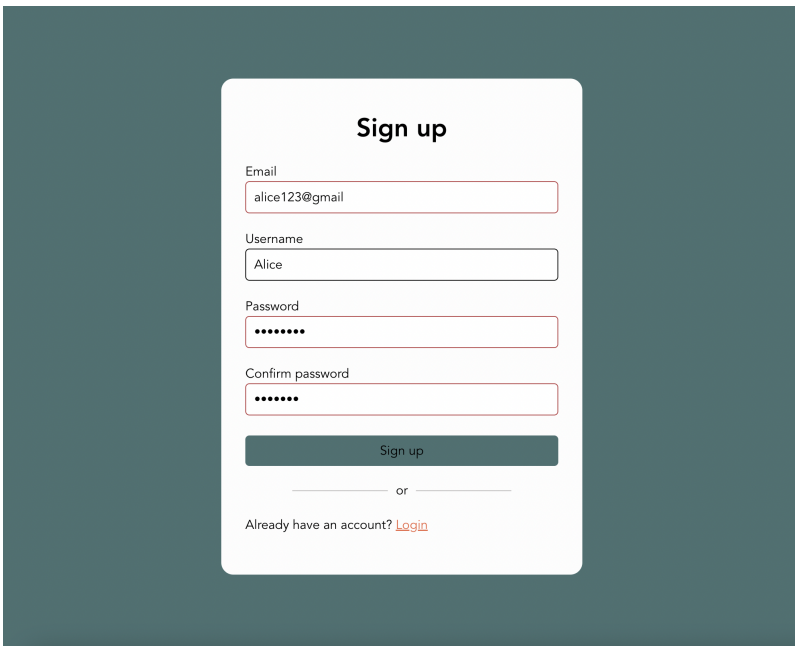


Рисунок 4.1. – Валідація введних даних при реєстрації

Після виправлення всіх помилок та введення правильних даних, обліковий запис успішно створюється. Далі користувач повинен увійти в систему, використовуючи електронну адресу та створений пароль. Під час входу також передбачена валідація введених даних. Якщо користувач вводить неправильний пароль або якщо обліковий запис з вказаною адресою електронної пошти не існує, система відобразить відповідне повідомлення про помилку. Це допомагає користувачеві зрозуміти, що саме було введено неправильно, і виправити помилку.

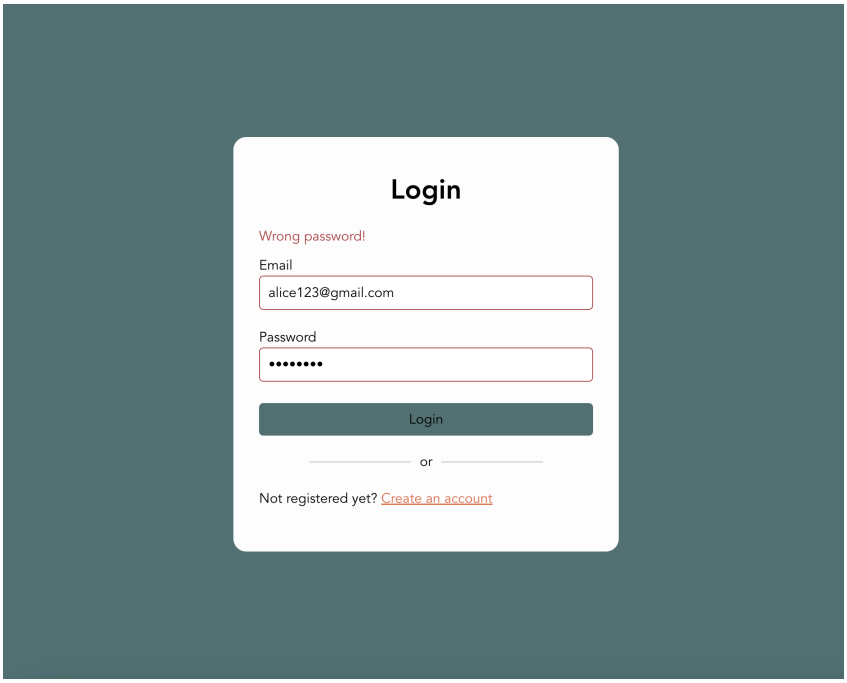


Рисунок 4.2. – Валідація введних даних при вході

Якщо ж дані коректні, користувач успішно входить в систему і потрапляє на основний екран месенджера. Як видно на рисунку 4.3, збережені в базі даних паролі зашифровані, що забезпечує їх захист.

```
_id: ObjectId('664b72a8d0e08387deb1cb9c')
username : "Alice"
password : "$2a$12$6iWa7J5nFsBC8du3nKfax.YOjNmf9lobHnCkY9NsTI.n7g0ZsQUTq"
email : "alice123@gmail.com"
imagePath : "images/2024-05-20T16:44:07.713Z-alice.png"
__v : 0
publicKey : "{"crv":"P-384","ext":true,"key_ops":[],"kty":"EC","x":"7oG3iSUj42qr4s9…"
```

Рисунок 4.3. – Документ створеного користувача

4.1.2 Надсилання та отримання повідомлень

На цьому екрані користувач бачить бокову панель (sidebar), де відображені всі користувачі, з якими можна спілкуватися (активні виділені спеціальною зеленою позначкою), а також опції для пошуку користувачів. Щоб почати спілкування, потрібно обрати чат.

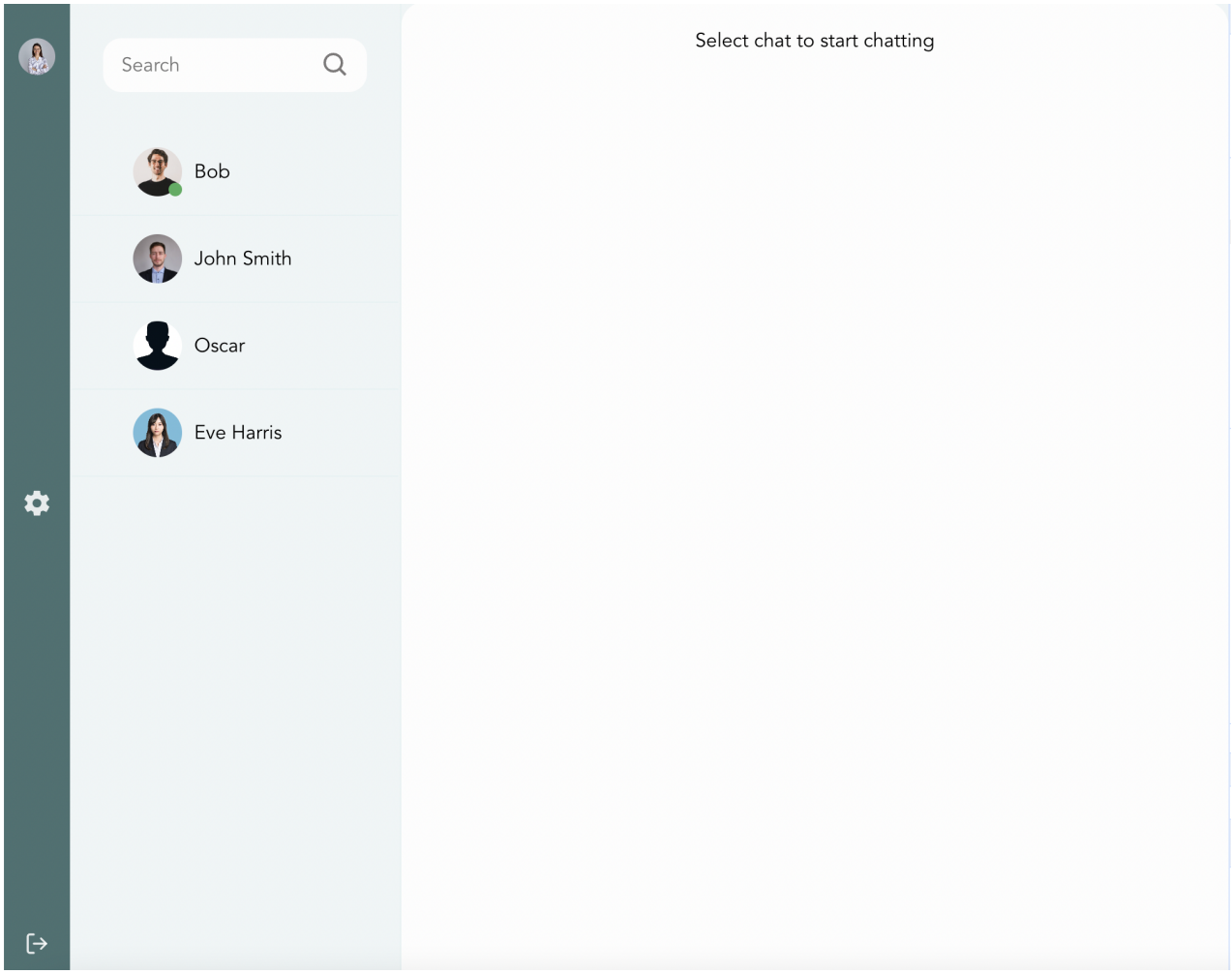


Рисунок 4.4. – Головна сторінка месенджера

Коли користувач відправляє повідомлення, воно миттєво з’являється у співбесідника, оскільки використовується Socket.IO, забезпечуючи зручне і ефективне спілкування.

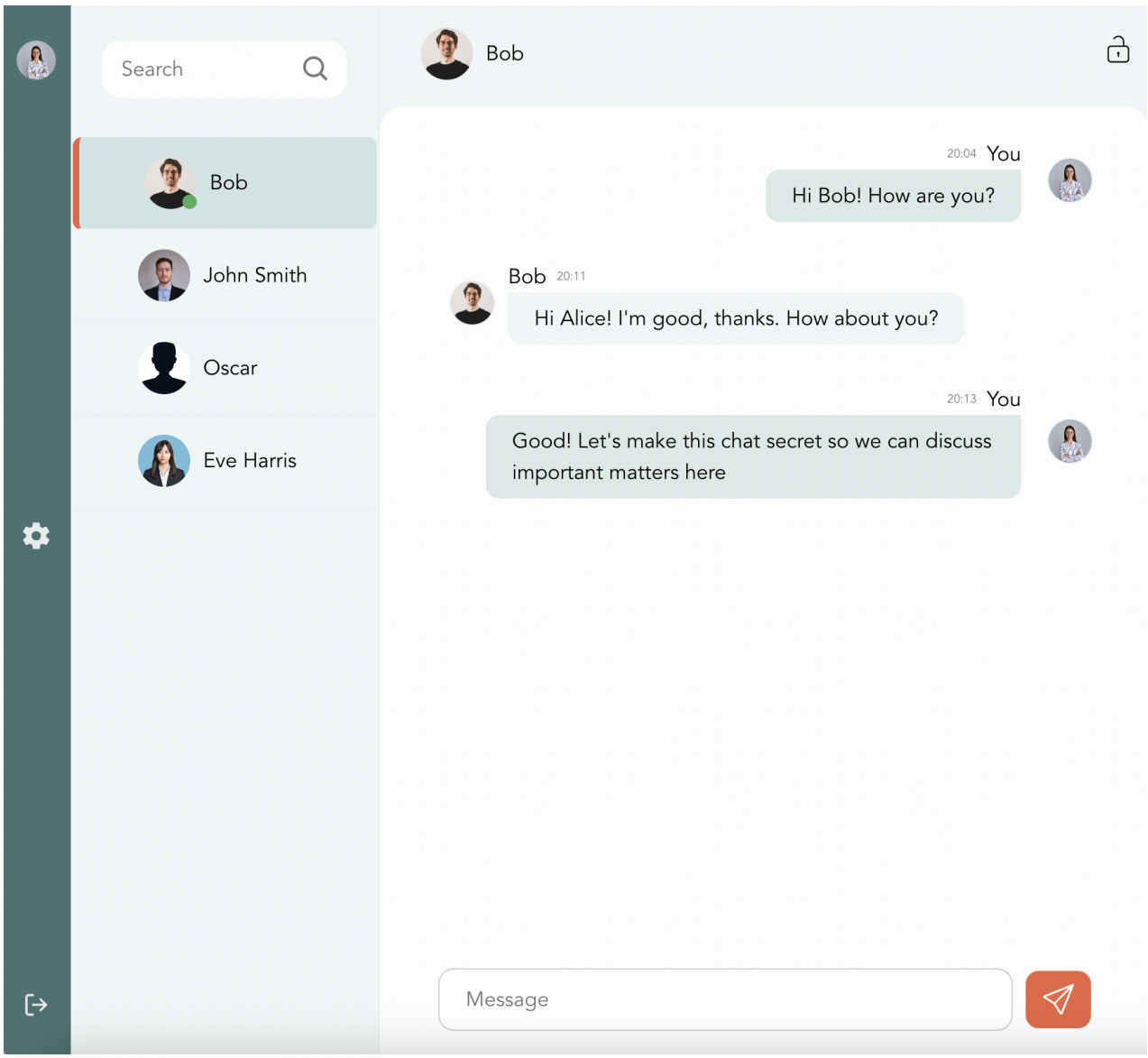


Рисунок 4.5. – Обмін повідомленнями між користувачами

Так як ми використовували кінцеве шифрування, при якому дані кодуються на одному кінці і декодуються на іншому, всі повідомлення зберігаються в базі даних в зашифрованому вигляді, що забезпечує захист і конфіденційності навіть у разі несанкціонованого доступу або перехопленні повідомлень [37].

```
_id: ObjectId('664b84a5fb4ee089395b3ec3')
senderId : ObjectId('664b72a8d0e08387deb1cb9c')
receiverId : ObjectId('664b7d7cd0e08387deb1cbbb')
message : "2Rz778bbu8ntHJuvXcQ4rXfwsXWjfYXDQXQFXrxWLpAS9IPNWjxOZ+wVx87e7KoZUZ5607…"
createdAt : 2024-05-20T17:13:09.340+00:00
updatedAt : 2024-05-20T17:13:09.340+00:00
__v : 0
```

Рисунок 4.6. – Документ надісланого повідомлення

Кожен користувач також має можливість зробити чат з певною особою секретним. Для цього потрібно натиснути на іконку відкритого замка поруч з іменем співбесідника, після чого відкривається вікно з підтвердженням та застереженням (перетворення чату на секретний означає, що всі повідомлення в ньому будуть видалені у разі введення фейкового паролю). Після натискання «Confirm» чат стає секретним і щоб це було відображено інтерфейсі, відкритий замок стає закритим.

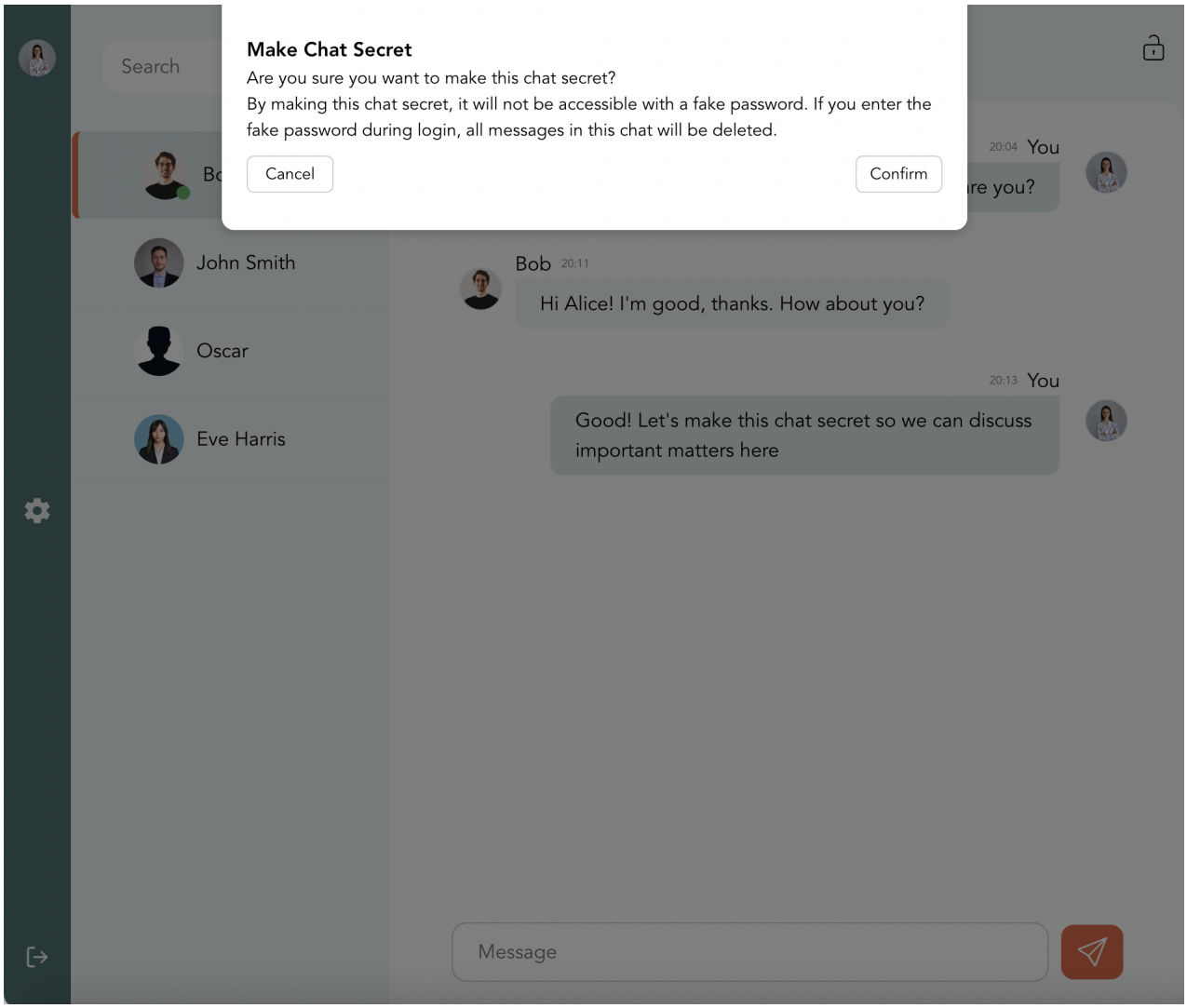


Рисунок 4.7. – Перетворення чату на секретний

4.1.3 Редагування особистих даних

Користувачі також мають можливість відкрити вікно налаштувань для редагування своїх облікових даних. Для цього потрібно натиснути на кнопку «Settings» на лівій панелі. У відкритому вікні власник може змінити ім’я, пароль (звісно з підтвердженнм старого), встановити або змінити фото, а також задати спеціальний фейковий пароль, який слугує захистом від вимагання. Встановлюючи його, можна прочитати підказку, де є пояснення щодо функціональності та корисності цієї розробки.

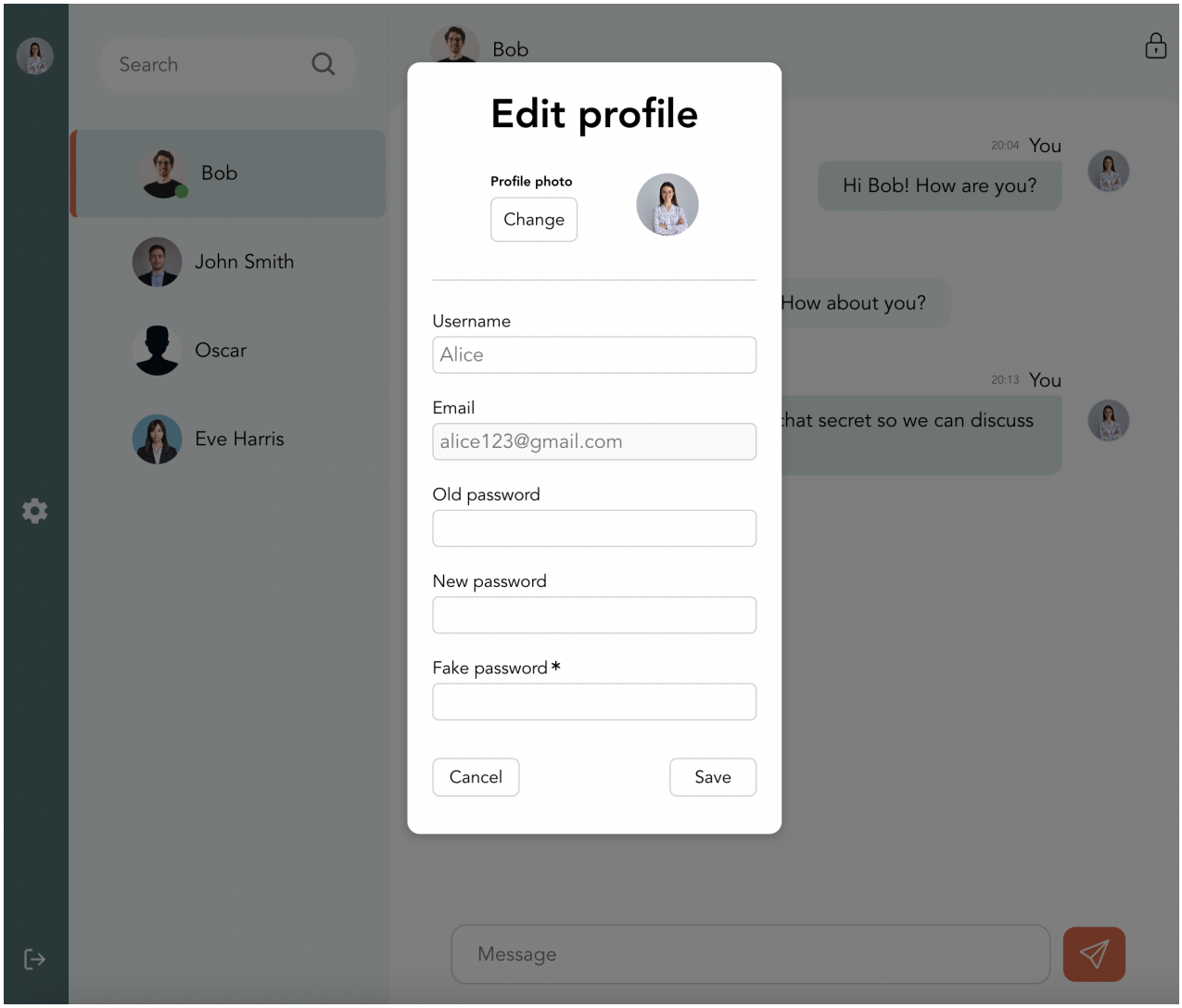


Рисунок 4.8. – Вікно редагування користувача

4.1.4 Використання фейкового паролю

Після встановлення фейкового паролю розглянемо його дію. Припустимо, що зловмисник, погрожуючи, вимагає пароль до акаунта, у такому випадку замість справжнього паролю надаємо йому фальшивий, електронна адреса залишається справжньою. Відразу після його введення користувач перенапрявляється на головне вікно месенджера, однак всі повідомлення в чатах, які були позначені секретними, є видаленими, як і в сам чат в базі даних. Іконки та функції, пов'язані з секретними чатами, приховуються, щоб здавалося, ніби вони ніколи не існували. Це створює ілюзію нормальної роботи месенджера без виявлення факту використання секретних чатів.

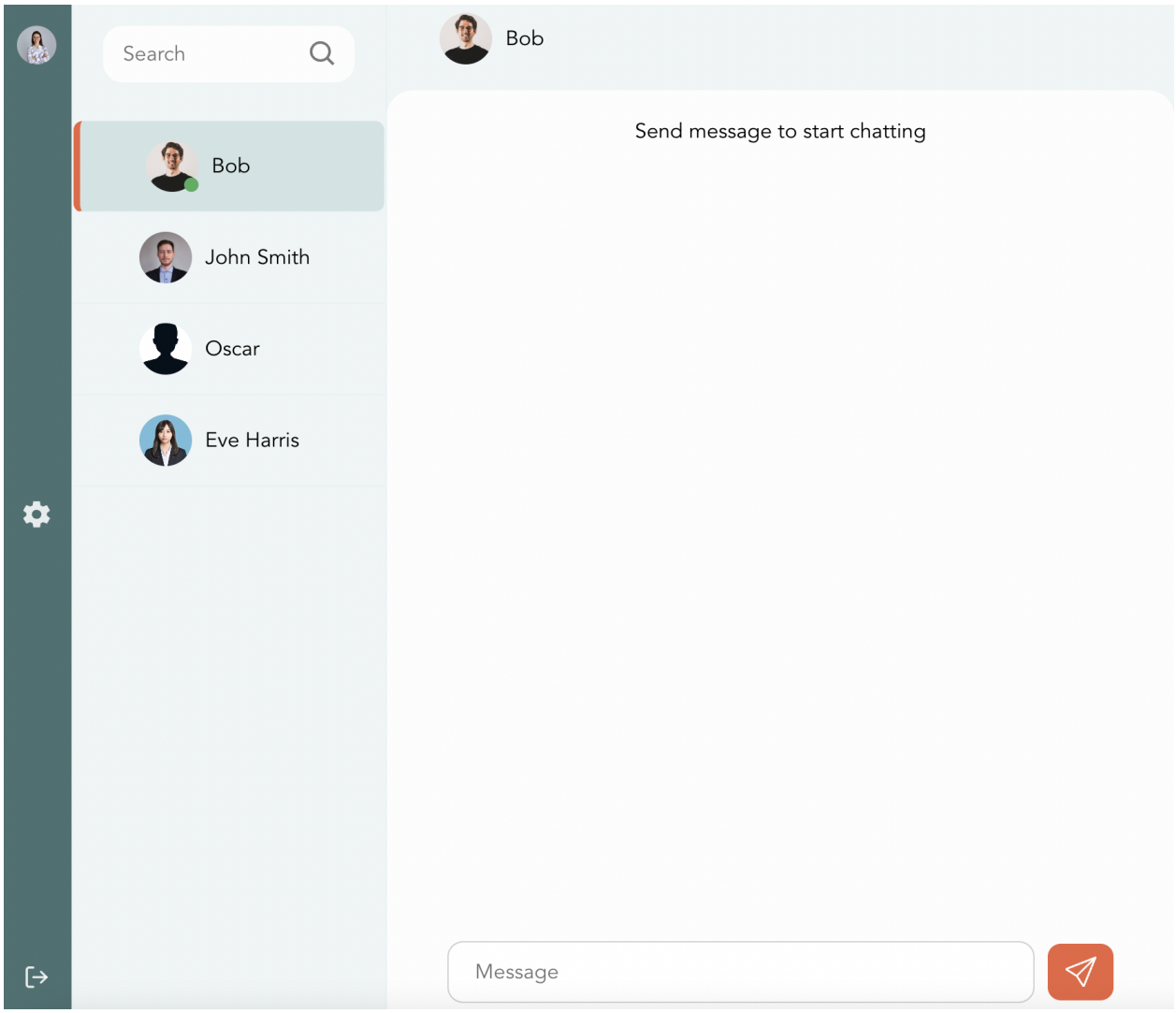


Рисунок 4.9. – Вікно користувача після введення фейкового паролю

4.2 Порівняння з існуючими рішеннями

У цьому розділі ми порівняємо розроблений веб-застосунок з розглянутими у першому месенджерами, такими як Telegram, Signal та Threema, які вже зарекомендували себе завдяки високій надійності, зручності використання та широкому функціоналу. Важливо зазначити, що ці месенджери розроблялися великими командами досвідчених розробників, і мають деякі унікальні переваги як в криптографічному сенсі, так і з інших сторін. І все ж у нашого проекту є ряд переваг, які роблять його конкурентоспроможним.

Перш за все, високий рівень безпеки, який забезпечують Telegram, Signal і Threema, полягає в тому, що вони застосовують передові криптографічні методи. Telegram використовує власний протокол MTProto, який реалізує можливість шифрування і забезпечує безпеку переданих даних. З іншого боку, Signal використовує один із найкращих у світі протоколів – Signal Protocol. Threema забезпечує повну анонімність своїх користувачів і зберігає всі дані локально, таким чином, централізоване сховище не несе такого ризику. Наш месенджер також використовує кінцеве шифрування для забезпечення безпеки повідомлень. Проте, на відміну від більш масштабних месенджерів, наш застосунок має просту і менш стійку реалізацію наскрізного шифрування, обмін і захист лише текстових повідомлень.

Відомі месенджери також є привабливими для користувачів через широкий спектр функцій: Telegram підтримує боти, канали, групи та інтеграції з іншими сервісами, Signal найкраще дотримується конфіденційності, з мінімізацією зберігання метаданих та відкритим вихідним кодом. Threema пропонує високу анонімність та безпеку завдяки локальному зберіганню даних. А серед важливих і унікальних функцій нашого месенджера є фейковий пароль. Це ще одна опція захисту, яка дає можливість користувачу стерти всю історію

повідомлень у секретних чатах при введенні фальшивого паролю. Така розробка відсутня у Telegram, Signal та Threema, що робить наш застосунок особливо корисним у випадках примусу та вимагання.

Отже, наш месенджер, хоч і слабший у глобальному криптографічному контексті, має менше функцій порівняно з розглянутими рішеннями та менше ресурсів для його розробки та вдосконалення, але він надає розширену функцію фальшивого пароля, якої бракує більшості популярних месенджерів і робить його цінним інструментом для користувачів, які шукають додаткові заходи безпеки для захисту своїх конфіденційних даних.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі цієї дипломної роботи ми проаналізували вже готовий проект месенджера, що забезпечує шифрування повідомлень. Ми розглянули процес входу до системи, який відображає механізм автентифікації та авторизації користувачів, процес надсилання та отримання повідомлень, редагування особистої інформації, але особлива увага була приділена додатковій функції безпеки – встановленню і використанню фейкового паролю.

Наш аналіз також охоплює порівняння розробленого месенджера з раніше розглянутими, де ми дійшли висновку, що він готовий задовольнити потреби користувачів у безпеці та зручності спілкування, і має місце для подальшого удосконалення.

ВИСНОВКИ

У результаті виконання дипломної роботи було створено веб-застосунок для безпечного шифрування текстових повідомлень, що відповідає сучасним вимогам інформаційної безпеки. Основна мета проекту – розробка надійного інструменту для захищеного спілкування – була успішно досягнута завдяки впровадженню кінцевого шифрування та інноваційної функції фейкового паролю.

Порівняння з існуючими месенджерами, такими як Telegram, Signal та Threema, показало, що, незважаючи на обмежені ресурси, наш застосунок може запропонувати унікальні функції, які підвищують безпеку користувачів. Відомі месенджери розроблені великими командами та володіють значними ресурсами, проте наш застосунок пропонує нові рівні захисту, яких часто бракує популярним рішенням.

Проект об'єднав найсучасніші технології та методи забезпечення безпеки даних. Використання технологічного стеку, до якого увійшли React, Node.js, MongoDB та Socket.io, дозволило створити продуктивний та масштабований застосунок. Використання Web Crypto API для реалізації шифрування забезпечило високий рівень захисту переданої інформації, що є критично важливим для будь-якого сучасного месенджера.

Архітектурні рішення проекту були спрямовані на забезпечення гнучкості та масштабованості системи. Впровадження клієнт-серверної архітектури з використанням REST API та моделі MVC дозволило забезпечити модульність, зручність підтримки та розширюваність системи. Така архітектура створює міцну основу для майбутніх оновлень та інтеграції нових функцій.

Унікальною особливістю розробленого застосунку є функція фейкового паролю, яка дозволяє користувачам захистити свої секретні повідомлення у випадку вимагання доступу, надаючи користувачам більше контролю над

своїми даними. Особливо важливо, що ця функція може бути надзвичайно корисною для військових, захищаючи їхні повідомлення від окупантів.

Загалом, виконана робота демонструє успішне застосування сучасних технологій для вирішення актуальної проблеми захисту інформації у сфері комунікацій. Розроблений веб-застосунок має значний потенціал для подальшого розвитку та використання у різних галузях, де необхідно забезпечити високий рівень конфіденційності та захисту даних. Цей проект підтверджує важливість інноваційного підходу до забезпечення інформаційної безпеки та демонструє, як сучасні технології можуть бути використані для створення ефективних та безпечних комунікаційних інструментів. Технічне завдання виконано повністю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is Telegram Messenger? [Електронний ресурс] – Режим доступу до ресурсу: https://mailchimp.com/resources/what-is-telegram/.
2. Is Telegram safe? Discover the app's strengths and limitations [Електронний ресурс] – Режим доступу до ресурсу: https://nordvpn.com/uk/blog/is-telegram-safe/.
3. Kraus R. What is Signal? The basics of the most secure messaging app. [Електронний ресурс] / R. Kraus, C. Silva. – 2023. – Режим доступу до ресурсу: https://mashable.com/article/what-is-signal-app.
4. Mora J. Demystifying the Signal Protocol for End-to-End Encryption (E2EE) [Електронний ресурс] / Justino Mora. – 2017. – Режим доступу до ресурсу: https://medium.com/@justinomora/demystifying-the-signal-protocol-for-end-to-end-encryption-e2ee-ad6a567e6cb4.
5. Threema Review 2024: Secure Messenger (Pros and Cons) [Електронний ресурс] – Режим доступу до ресурсу: https://restoreprivacy.com/secure-encrypted-messaging-apps/threema/.
6. Threema — Вікіпедія [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Threema.
7. ТОП безпечних та захищених месенджерів [Електронний ресурс] – Режим доступу до ресурсу: https://kr-labs.com.ua/blog/top-secure-and-privacy-messaging-apps/.
8. Deshpande C. What Is React? [Електронний ресурс] / Chinmayee Deshpande. – 2020. – Режим доступу до ресурсу: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs.
9. What Is Node.js and Why You Should Use It [Електронний ресурс] – Режим доступу до ресурсу: https://kinsta.com/knowledgebase/what-is-node-js/.

10. What Is MongoDB? All About the Popular Open Source Database [Електронний ресурс] – Режим доступу до ресурсу: https://kinsta.com/knowledgebase/what-is-mongodb/.
11. What Is MongoDB? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/mongodb.
12. Socket.IO: How it works, when to use it, and how to get started [Електронний ресурс] – Режим доступу до ресурсу: https://ably.com/topic/socketio.
13. What Is End-to-End Encryption? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/end-to-end-encryption.
14. H. Bawake, S. Cholke, A. Dhol. E2EE Web Messaging Application Using Cryptography Techniques // International Journal for Research in Applied Science and Engineering Technology, 2023. – (11; т. 4). – С. 2368–2373.
15. Symmetric Key Encryption - why, where and how it's used in banking [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking.
16. Renuka D. Comparative analysis of enhanced hybrid encryption algorithm with single encryption algorithm / D. Renuka, D. Vaishali // Proceeding International Conference on Science and Engineering, 2023. – (11; т. 1).
17. End-to-end encryption and guide on how it works [Електронний ресурс] – Режим доступу до ресурсу: https://www.preveil.com/blog/end-to-end-encryption/.
18. Difference Between Symmetric and Asymmetric Key Encryption - GeeksforGeeks [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/difference-between-symmetric-and-asymmetric-key-encryption/.
19. Saternos C. Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful / Casimir Saternos., 2014. – 260 с.

20. Curry E. Flexible Self-Management Using the Model-View-Controller Pattern [Електронний ресурс] / E. Curry, P. Grace // IEEE Computer Society. – 2008. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/abstract/document/4497770.
21. What is a REST API? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/rest-apis.
22. MVC Framework Introduction - GeeksforGeeks [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/mvc-framework-introduction/.
23. Web Crypto [Електронний ресурс] – Режим доступу до ресурсу: https://developers.cloudflare.com/workers/runtime-apis/web-crypto/.
24. Wichmann P., Blochberger M., Federrath H. Web Cryptography API: Prevalence and Possible Developer Mistakes. – New York, NY, USA: ARES 2022: The 17th International Conference on Availability, Reliability and Security, 2022.
25. ECDH with Golang and Kryptology [Електронний ресурс] – Режим доступу до ресурсу: https://asecuritysite.com/ecc/go_ecdh.
26. Ahmed M. Diffie-Hellman and Its Application in Security Protocols [Електронний ресурс] / M. Ahmed, B. Sanjabi, D. Aldiaz // International Journal of Engineering Science and Innovative Technology (IJESIT). – 2008. – Режим доступу до ресурсу: https://www.researchgate.net/profile/Maryam-Ahmed-19/publication/279725863_Diffie_helman_and_its_Use_in_Secure_Internet_Protocols/links/5617703c08ae839f3c7d8baa/Diffie-helman-and-its-Use-in-Secure-Internet-Protocols.pdf.
27. SubtleCrypto - Web APIs | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto#browser_compatibility.

28. Luoma-aho M. JavaScript Web Cryptography API [Електронний ресурс] / Mika Luoma-aho. – 2015. – Режим доступу до ресурсу: https://www.theseus.fi/bitstream/handle/10024/92960/Web_Cryptography_API_Luoma-aho.pdf.
29. SubtleCrypto: deriveKey() method - Web APIs | MDN [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveKey.
30. Elliptic Curve Diffie-Hellman key agreement [Електронний ресурс] – Режим доступу до ресурсу: https://docs.yubico.com/yesdk/users-manual/application-piv/key-agreement.html.
31. Web Crypto API example [Електронний ресурс] – Режим доступу до ресурсу: https://mdn.github.io/dom-examples/web-crypto/derive-key/index.html.
32. Deshmukh S. Hybrid cryptography technique using modified Diffie-Hellman and RSA [Електронний ресурс] / S. Deshmukh, R. Patil // International Journal of Computer Science and Information Technologies. – 2014. – Режим доступу до ресурсу: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7e0c0e0badb70968c9d0c59e76de9a0db39908c5.
33. Web Cryptography API [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3.org/TR/WebCryptoAPI/#aes-gcm.
34. Matheus C. End-to-End Encrypted Chat with JS & Web Crypto API [Електронний ресурс] / Matheus. – 2020. – Режим доступу до ресурсу: https://getstream.io/blog/web-crypto-api-chat/.
35. What is Bcrypt and how it works? [Електронний ресурс] – Режим доступу до ресурсу: https://nordvpn.com/uk/blog/what-is-bcrypt/.
36. JWT Authentication with Node.js - GeeksforGeeks [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/jwt-authentication-with-node-js/.

37. Nabeel M. The Many Faces of End-to-End Encryption and Their Security Analysis [Електронний ресурс] / Mohamed Nabeel // IEEE 1st International Conference on Edge Computing. – 2017. – Режим доступу до ресурсу: https://www.researchgate.net/profile/Mohamed-Nabeel-2/publication/319633280_The_Many_Faces_of_End-to-End_Encryption_and_Their_Security_Analysis/links/5f231da8299bf1340494b2ee/The-Many-Faces-of-End-to-End-Encryption-and-Their-Security-Analysis.pdf.

ДОДАТОК 1

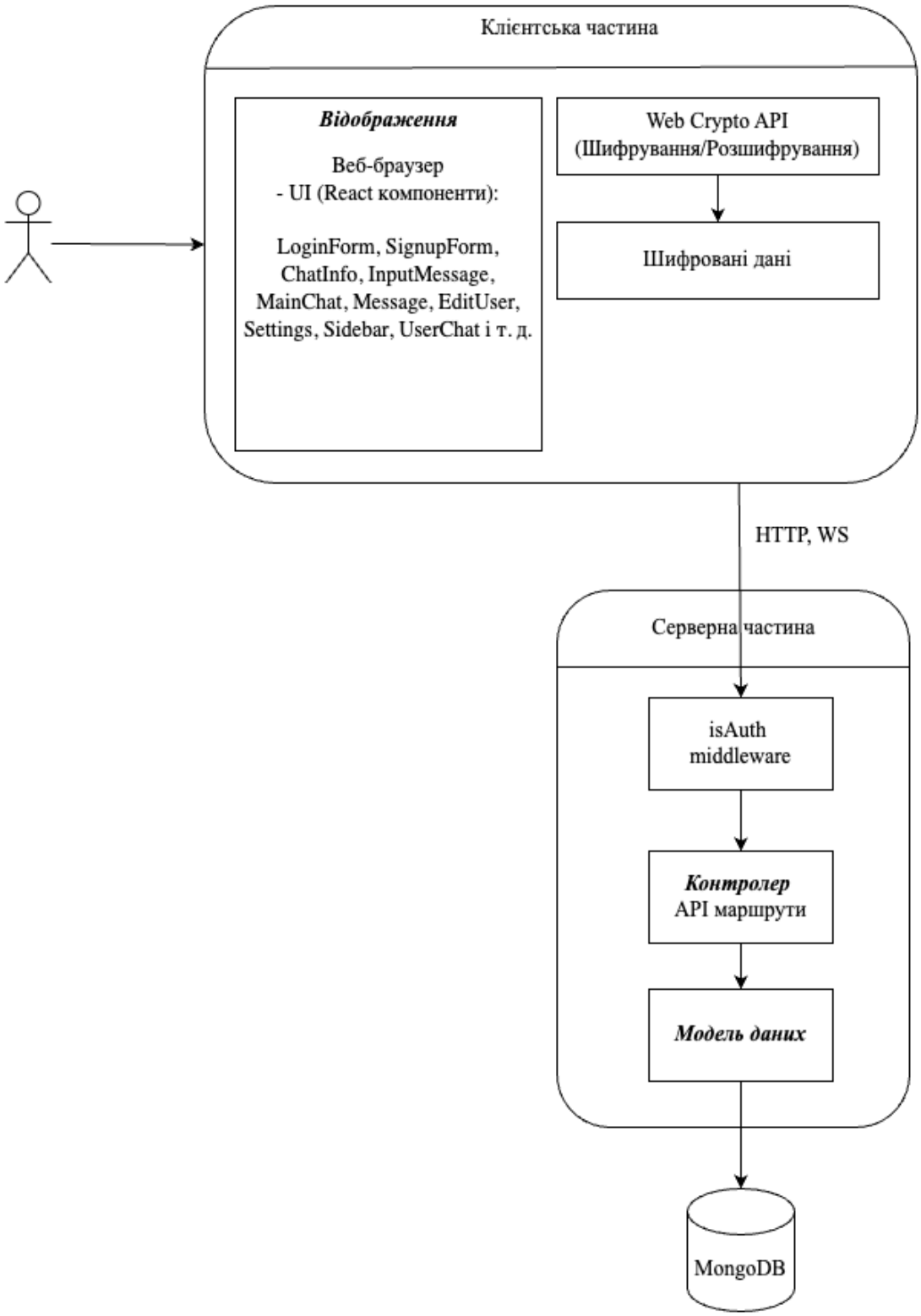
Структура веб-застосунка з підтримкою шифрування текстових повідомлень для безпечного спілкування

Схема структурна

ІАЛЦ.467200.004 Е1

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.004 Е1		
		№ докум.	Підпис	Дата			
Розробив		Негрич Я. О.			**Структура веб-застосунка з підтримкою шифрування текстових повідомлень для безпечного спілкування** Схема структурна	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Гончаренко О.О.					
Н. Контр.		Виноградов Ю.М				КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02	
Затвердив							

ДОДАТОК 2

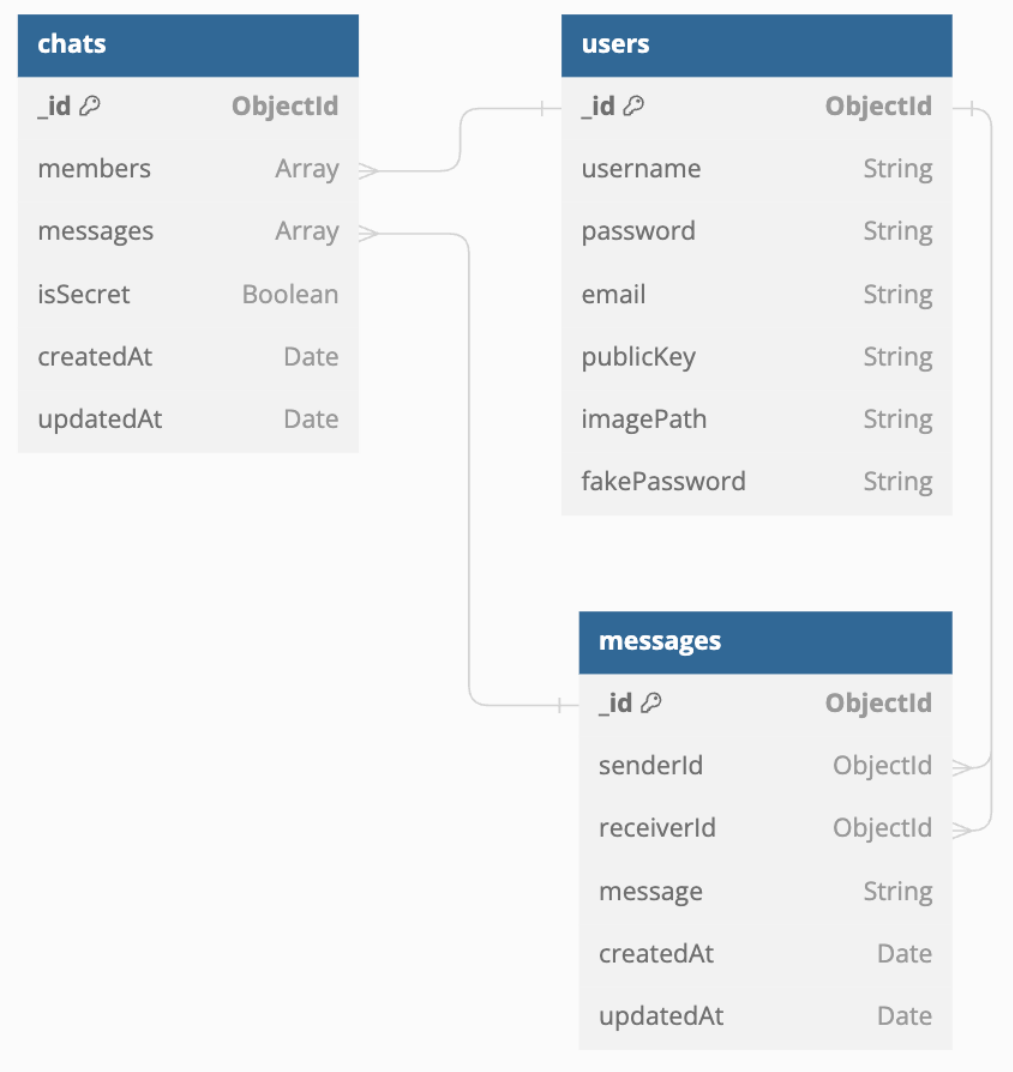
Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування

Діаграма сутність-зв’язок (Схема функціональна)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.005 Д2	
		№ докум.	Підпис	Дата		
Розробив		Негрич Я. О.			**Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування** Діаграма сутність-зв'язок Схема функціональна	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гончаренко О.О.				
Н. Контр.		Виноградов Ю.М				КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02
Затвердив						

ДОДАТОК 3

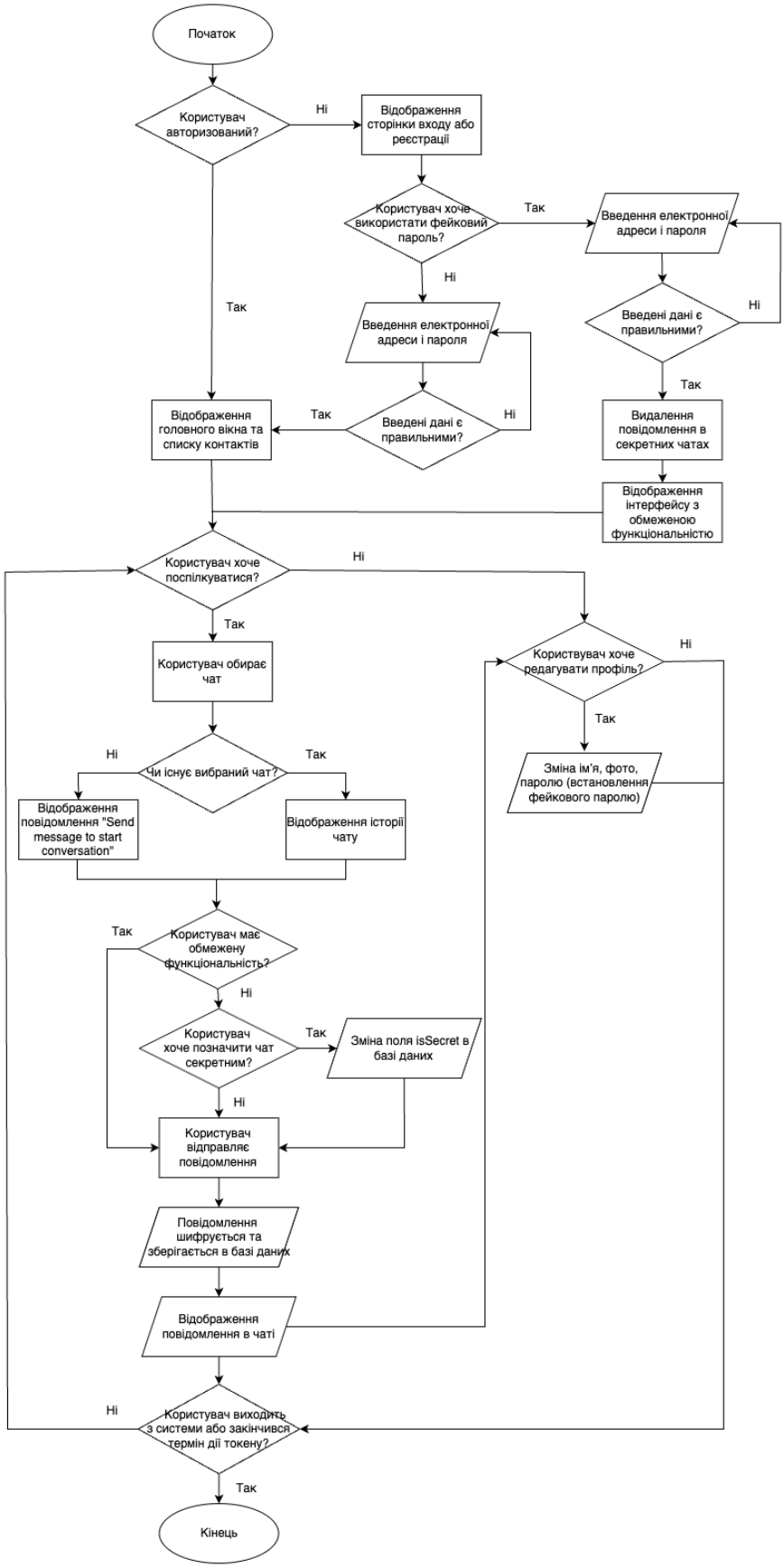
Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування

Алгоритм роботи системи (Схема принципова)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2024 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Негрич Я. О.			**Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування** Алгоритм роботи системи Схема принципова	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Гончаренко О.О.					
Н. Контр.		Виноградов Ю.М				КПІ ім. Ігоря Сікорського, ФІОТ, гр. ІО-02	
Затвердив							

ДОДАТОК 4

Веб-застосунок з підтримкою шифрування текстових повідомлень для безпечного спілкування

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 31

Київ 2024 р

backend/app.js

```
const DB_URL =
  'mongodb+srv://slnegrich:hzzXOK8JOsz21brm@cluster0.vs37vsv.mongodb.net/messenger-
db?retryWrites=true&w=majority&appName=Cluster0';
const express = require('express');
const cors = require('cors');
const authRoutes = require('./routes/auth');
const messageRoutes = require('./routes/message');
const userRoutes = require('./routes/user');
const { isAuth } = require('./middleware/isAuth');
const mongoose = require('mongoose');
const app = express();
const bodyParser = require('body-parser');
const { socketManager } = require('./socket/socket');
const path = require('path');
const multer = require('multer');

const fileStorage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'images');
  },
  filename: (req, file, cb) => {
    cb(null, new Date().toISOString() + '-' + file.originalname);
  },
});

const fileFilter = (req, file, cb) => {
  if (
    file.mimetype === 'image/png' ||
    file.mimetype === 'image/jpg' ||
    file.mimetype === 'image/jpeg'
  ) {
    cb(null, true);
  } else {
    cb(null, false);
  }
};

app.use(bodyParser.json());
app.use(
  multer({ storage: fileStorage, fileFilter: fileFilter }).single('image')
);
app.use('/images', express.static(path.join(__dirname, 'images')));
app.use(cors({ origin: 'http://localhost:5173' }));
app.use((req, res, next) => {
  // res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET,POST,PATCH,DELETE');
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');
  next();
});

app.use('/auth', authRoutes);
app.use('/message', isAuth, messageRoutes);
app.use('/users', isAuth, userRoutes);

app.use((error, req, res, next) => {
  console.log(error);
```

```
const status = error.statusCode || 500;
const message = error.message || 'An error occurred';
const data = error.data || [];
res.status(status).json({ message: message, data: data });
});
const onlineUserIds = {};

mongoose
  .connect(DB_URL)
  .then((result) => {
    const server = app.listen(3000);
    const io = require('./socket/socketHelper').init(server);
    socketManager();
  })
  .catch((err) => {
    console.log(err);
  });
```

backend/routes/auth.js

```
const express = require('express');
const authController = require('../controllers/auth');
const router = express.Router();
const { body } = require('express-validator/check');
const User = require('../models/user');

router.post(
  '/signup',
  [
    body('email')
      .isEmail()
      .withMessage('Invalid email!')
      .custom((value) => {
        return User.findOne({ email: value }).then((user) => {
          if (user) {
            return Promise.reject('User with this email already exists!');
          }
        });
      })
      .normalizeEmail(),
    body('password').trim().isLength({ min: 8 }),
    body('username').trim().not().isEmpty(),
  ],
  authController.signup
);
router.post('/login', authController.login);

module.exports = router;
```

backend/routes/message.js

```
const express = require('express');
const router = express.Router();
const messageController = require('../controllers/message');

router.post('/send/:id', messageController.sendMessage);
router.get('/:id', messageController.getMessages);

module.exports = router;
```

backend/routes/user.js

```
const express = require('express');
const router = express.Router();
```

```
const userController = require('../controllers/user');

router.get('/', userController.getSidebarChats);
router.get('/getPublicKey/:receiverId', userController.getReceiverPublicKey);
router.get('/getEditUser', userController.getEditUser);
router.post('/editUser', userController.editUser);
router.get('/getLoggedUser', userController.getUserInfo);
router.post('/setChatIsSecret/:receiverId', userController.setChatIsSecret);

module.exports = router;
```

backend/models/chat.js

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const chatSchema = new Schema(
  {
    members: [{ type: Schema.Types.ObjectId, ref: 'User' }],
    messages: [{ type: Schema.Types.ObjectId, ref: 'Message' }],
    isSecret: { type: Boolean, default: false },
  },
  { timestamps: true }
);

module.exports = mongoose.model('Chat', chatSchema);
```

backend/models/message.js

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const messageSchema = new Schema(
  {
    senderId: {
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: true,
    },
    receiverId: {
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: true,
    },
    message: {
      type: String,
      required: true,
    },
  },
  { timestamps: true }
);

module.exports = mongoose.model('Message', messageSchema);
```

backend/models/user.js

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;
```

```
const userSchema = new Schema({
  username: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
  },
  publicKey: {
    type: String,
  },
  imagePath: {
    type: String,
    default: '',
  },
  fakePassword: {
    type: String,
  },
});

module.exports = mongoose.model('User', userSchema);
```

backend/middleware/isAuth.js

```
const jwt = require('jsonwebtoken');
const { SECRET_KEY } = require('../controllers/auth');
exports.isAuth = (req, res, next) => {
  const authHeader = req.get('Authorization');
  if (!authHeader) {
    const error = new Error('No Authorization Header!');
    error.statusCode = 401;
    throw error;
  }
  const token = authHeader.split(' ')[1];
  let decodedToken;
  try {
    decodedToken = jwt.verify(token, SECRET_KEY);
  } catch (error) {
    error.statusCode = 500;
    throw error;
  }

  if (!decodedToken) {
    const error = new Error('Not authenticated!');
    error.statusCode = 401;
    throw error;
  }
  req.userId = decodedToken.userId;
  next();
};
```

backend/controllers/auth.js

```
const User = require('../models/user');
const bcrypt = require('bcryptjs');
const { validationResult } = require('express-validator/check');
const jwt = require('jsonwebtoken');
const Chat = require('../models/chat');
```

```
const SECRET_KEY =
  '5227c42b3c78f653868cfd1b525b2a8d6f28e32bb8e1813c54ccc703b58bb9f3';

exports.signup = async (req, res, next) => {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      const error = new Error('Wrong input!');
      error.statusCode = 422;
      error.data = errors.array();
      throw error;
    }

    const username = req.body.username;
    const password = req.body.password;
    const email = req.body.email;
    const confirmedPassword = req.body.confirmedPassword;

    if (password !== confirmedPassword) {
      const error = new Error('Passwords should match!');
      error.statusCode = 400;
      throw error;
    }
    const hashedPassword = await bcrypt.hash(password, 12);

    const user = new User({
      username,
      password: hashedPassword,
      email,
    });

    const result = await user.save();
    res.status(201).json({ result });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};
exports.login = async (req, res, next) => {
  try {
    const email = req.body.submitData.email;
    const password = req.body.submitData.password;
    const publicKey = req.body.publicKeyJwk;
    const user = await User.findOne({ email: email });
    let fakePasswordIsEqual = undefined;

    if (!user) {
      const error = new Error("User with this email doesn't exists.");
      error.statusCode = 404;
      throw error;
    }
    if (user.fakePassword) {
      fakePasswordIsEqual = await bcrypt.compare(password, user.fakePassword);
    }

    if (fakePasswordIsEqual) {
      const secretChats = await Chat.find({
        members: user._id,
        isSecret: true,
      });
      const deleteRes = await Promise.all(
        secretChats.map((chat) => chat.deleteOne())
      );
```

```
      const token = await handlePublicKeyAndToken(user, publicKey);

      res.status(200).json({ token: token, userId: user._id, isFake: true });
    }

    const passwordIsEqual = await bcrypt.compare(password, user.password);
    if (!passwordIsEqual) {
      const error = new Error('Wrong password!');
      error.statusCode = 401;
      throw error;
    }
    const token = await handlePublicKeyAndToken(user, publicKey);

    res.status(200).json({ token: token, userId: user._id, isFake: false });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

const handlePublicKeyAndToken = async (user, publicKey) => {
  if (!publicKey) {
    const error = new Error('Public key is missing in the request!');
    error.statusCode = 400;
    throw error;
  }
  user.publicKey = publicKey;
  const updatedUser = await user.save();

  const token = jwt.sign(
    {
      email: updatedUser.email,
      userId: updatedUser._id.toString(),
    },
    SECRET_KEY,
    { expiresIn: '7d' }
  );

  return token;
};

exports.SECRET_KEY = SECRET_KEY;
```

backend/controllers/message.js

```
const Chat = require('../models/chat');
const Message = require('../models/message');
const { getSocketId } = require('../socket/socket');
const io = require('../socket/socketHelper');

exports.sendMessage = async (req, res, next) => {
  try {
    const receiverId = req.params.id;
    const senderId = req.userId;
    const message = req.body.message;

    let chat = await Chat.findOne({
      members: { $all: [senderId, receiverId] },
    });

    if (!chat) {
      chat = await new Chat({
```

```
        members: [senderId, receiverId],
        messages: [],
      });
    }

    const newMessage = await new Message({
      senderId,
      receiverId,
      message,
    });

    const result = await newMessage.save();

    chat.messages.push(result._id);
    await chat.save();

    const receiverSocket = getSocketId(receiverId);
    if (receiverSocket) {
      io.getIo().to(receiverSocket).emit('new-message', newMessage);
    }

    res.status(201).json({ newMessage });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.getMessages = async (req, res, next) => {
  const senderId = req.userId;
  const receiverId = req.params.id;
  try {
    const chat = await Chat.findOne({
      members: { $all: [senderId, receiverId] },
    }).populate('messages');
    if (!chat || !chat.messages) {
      const error = new Error('No messages found!');
      error.statusCode = 404;
      throw error;
    }
    const messages = chat.messages;
    res.status(200).json({ chat: messages, isSecret: chat.isSecret });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};
```

backend/controllers/user.js

```
const User = require('../models/user');
const bcrypt = require('bcryptjs');
const path = require('path');
const fs = require('fs');
const Chat = require('../models/chat');
exports.getSidebarChats = async (req, res, next) => {
  try {
    const userId = req.userId;
```

```
    const users = await User.find({ _id: { $ne: userId } }).select(
      '-password -fakePassword'
    );
    res.status(200).json({ users });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.getReceiverPublicKey = async (req, res, next) => {
  try {
    const receiverId = req.params.receiverId;
    const receiver = await User.findById(receiverId);
    res.status(200).json({ publicKey: receiver.publicKey });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.getEditUser = async (req, res, next) => {

try {
    const userId = req.userId;
    const user = await User.findById(userId).select('-password -fakePassword');
    res.status(200).json({ user });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.editUser = async (req, res, next) => {
  try {
    const userId = req.userId;
    const imagePath = req.file?.path || '';
    const username = req.body.username;
    const oldPassword = req.body['old-password'];
    const newPassword = req.body['new-password'];
    const fakePassword = req.body['fake-password'] || '';

    const user = await User.findById(userId);
    if (username && username.length <= 2) {
      const error = new Error('Username should be minimun 3 characters!');
      error.statusCode = 401;
      throw error;
    } else if (username && username.length >= 3) {
      user.username = username;
    }
    if (imagePath) {
      if (user.imagePath) {
        clearImage(user.imagePath);
      }
      user.imagePath = imagePath;
    }
    if (oldPassword) {
      const passwordIsEqual = await bcrypt.compare(oldPassword, user.password);
```

```
if (!passwordIsEqual) {
        const error = new Error('Wrong old password!');
        error.statusCode = 401;
        throw error;
      } else if (passwordIsEqual) {
        if (newPassword && newPassword.trim().length >= 8) {
          user.password = newPassword;
        } else {
          const error = new Error('You should set new valid password!');
          error.statusCode = 401;
          throw error;
        }
      }
    }
    if (fakePassword && fakePassword.trim().length < 8) {
      const error = new Error('You should set valid fake password!');
      error.statusCode = 401;
      throw error;
    } else if (fakePassword && fakePassword.trim().length >= 8) {
      const fakePasswordIsEqual = await bcrypt.compare(
        fakePassword,
        user.password
      );
      if (fakePasswordIsEqual) {
        const error = new Error('Fake password should be unique!');
        error.statusCode = 401;
        throw error;
      }
      const hashedFakePassword = await bcrypt.hash(fakePassword, 12);
      user.fakePassword = hashedFakePassword;
    }
    const result = await user.save();
    // console.log(result);
    res.status(200).json({ result });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.getUserInfo = async (req, res, next) => {
  try {
    const userId = req.userId;
    const user = await User.findById(userId).select('-password -fakePassword');
    res.status(200).json({ user });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};

exports.setChatIsSecret = async (req, res, next) => {
  try {
    const senderId = req.userId;
    const receiverId = req.params.receiverId;
    const isSecret = req.body.isSecret;
    const chat = await Chat.findOne({
      members: { $all: [senderId, receiverId] },
    });
```

```
if (!chat) {
      const error = new Error('No chat found!');
      error.statusCode = 401;
      throw error;
    }
    chat.isSecret = isSecret;
    const result = await chat.save();

    res.status(200).json({ isSecret: result.isSecret });
  } catch (error) {
    if (!error.statusCode) {
      error.statusCode = 500;
    }
    next(error);
  }
};
const clearImage = (filePath) => {
  filePath = path.join(__dirname, '..', filePath);
  fs.unlink(filePath, (err) => console.log(err));
};
```

backend/socket/socketHelper.js

```
let io;

module.exports = {
  init: (httpServer) => {
    io = require('socket.io')(httpServer, {
      cors: {
        origin: 'http://localhost:5173',
      },
    });
    return io;
  },
  getIo: () => {
    if (!io) {
      throw new Error('Socket not initialized!');
    }
    return io;
  },
};
```

backend/socket/socket.js

```
const io = require('./socketHelper');
const onlineUserIds = {};
exports.getSocketId = (id) => {
  return onlineUserIds[id];
};

exports.socketManager = function () {
  io.getIo().on('connection', (socket) => {
    const userId = socket.handshake.query.userId;
    onlineUserIds[userId] = socket.id;
    io.getIo().emit('online-users', onlineUserIds);

    socket.on('disconnect', () => {
      delete onlineUserIds[userId];
      io.getIo().emit('online-users', onlineUserIds);
    });
  });
};
```

frontend/src/App.jsx

```
import Main from './pages/Main';
import { RouterProvider, createBrowserRouter } from 'react-router-dom';
import Error from './pages/Error';
import LoginPage, { action as loginAction } from './pages/LoginPage';
import SignupPage, { action as signupAction } from './pages/SignupPage';
import { loader as mainLoader } from './pages/Main';

function App() {
  const router = createBrowserRouter([
    {
      path: '/auth',
      errorElement: <Error />,
      children: [
        { path: 'login', element: <LoginPage />, action: loginAction },
        { path: 'signup', element: <SignupPage />, action: signupAction },
      ],
    },
    { path: '/user', element: <Main />, id: 'main', loader: mainLoader },
  ]);
  return <RouterProvider router={router} />;
}

export default App;
```

frontend/src/components/Auth/LoginForm.jsx

```
import classes from './FormStyle.module.css';
import { Link, Form, useActionData, useNavigation } from 'react-router-dom';
export default function LoginForm() {
  const data = useActionData();
  const navigation = useNavigation;
  const isSubmitting = navigation.state === 'submitting';
  return (
    <Form method="post" action="/auth/login" className={classes.form}>
      <h1>Login</h1>
      <p className={classes.backendError}>
        {data && data.message ? data.message : null}
      </p>
      <div className={classes.inputField}>
        <label htmlFor="email">Email</label>
        <input
          type="text"
          name="email"
          id="email"
          className={data && !data.emailIsValid ? classes.invalid : ''}
        />
      </div>
      <div className={classes.inputField}>
        <label htmlFor="password">Password</label>
        <input
          type="password"
          name="password"
          id="password"


       className={data && !data.passwordIsValid ? classes.invalid : ''}
        />
      </div>
      <button disabled={isSubmitting} className={classes.btn}>
        {isSubmitting ? 'Submitting...' : 'Login'}
      </button>
      <div className={classes.lineContainer}>
        <div className={classes.line}></div>
```

```
        <p>or</p>
      </div>
      <p>
        Not registered yet? <Link to="/auth/signup">Create an account</Link>
      </p>
    </Form>
  );
}
```

frontend/src/components/Auth/SignupForm.jsx

```
import classes from '../Auth/FormStyle.module.css';
import { Link, Form, useActionData, useNavigation } from 'react-router-dom';

export default function SignupForm() {
  const data = useActionData();
  const navigation = useNavigation;
  const isSubmitting = navigation.state === 'submitting';
  return (
    <Form method="post" action="/auth/signup" className={classes.form}>
      <h1>Sign up</h1>
      <p className={classes.backendError}>
        {data && data.data ? data.data.map((error) => error.msg) : null}
      </p>

      <div className={classes.inputField}>
        <label htmlFor="email">Email</label>
        <input
          type="text"
          name="email"
          id="email"
          className={data && !data.emailIsValid ? classes.invalid : ''}
        />
      </div>
      <div className={classes.inputField}>
        <label htmlFor="username">Username</label>
        <input
          type="text"
          name="username"
          id="username"
          className={data && !data.usernameIsValid ? classes.invalid : ''}
        />
      </div>
      <div className={classes.inputField}>
        <label htmlFor="password">Password</label>
        <input
          type="password"
          name="password"
          id="password"
          className={data && !data.passwordIsValid ? classes.invalid : ''}
        />
      </div>
      <div className={classes.inputField}>
        <label htmlFor="confirmed-password">Confirm password</label>
        <input
          type="password"

          name="confirmed-password"
          id="confirm-password"
          className={data && !data.passwordIsValid ? classes.invalid : ''}
        />
      </div>
      <button disabled={isSubmitting} className={classes.btn}>
        {isSubmitting ? 'Submitting...' : 'Sign up'}
      </button>
```

```
      <div className={classes.lineContainer}>
        <div className={classes.line}></div>
        <p>or</p>
      </div>
      <p>
        Already have an account? <Link to="/auth/login">Login</Link>
      </p>
    </Form>
  );
}
```

frontend/src/components/MainChat/ChatInfo.jsx

```
import { useContext, useEffect, useState } from 'react';
import classes from '../MainChat/ChatInfo.module.css';
import { UsersContext } from '../../store/users-context';
import ProfileImage from '../../images/profile-image.png';
import { CiUnlock } from 'react-icons/ci';
import { CiLock } from 'react-icons/ci';
import Modal from '../SettingsBar/Modal';
import { getFlag, getToken } from '../../utils/localStorageManipulation';
export default function ChatInfo({ isSecretChat }) {
  const ctx = useContext(UsersContext);
  const [isSecret, setIsSecret] = useState(isSecretChat);
  const [modalIsVisible, setModalIsVisible] = useState(false);
  const lock = isSecret ? <CiLock /> : <CiUnlock />;
  const token = getToken();
  const flag = getFlag();

  useEffect(() => {
    setIsSecret(isSecretChat);
  }, [isSecretChat]);

  async function postIsSecret(isSecretBoolean) {
    const response = await fetch(
      `http://localhost:3000/users/setChatIsSecret/${ctx.activeUser._id}`,
      {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${token}`,
        },
        body: JSON.stringify({ isSecret: isSecretBoolean }),
      }
    );
    const data = await response.json();
    setIsSecret(data.isSecret);
  }

  function setIsSecretHandler() {
    if (!isSecret) {
      setModalIsVisible(true);
    }
    if (isSecret) {
       postIsSecret(false);
    }
  }
function onClose() {
    setModalIsVisible(false);
  }

  function confirmIsSecret() {
    postIsSecret(true);
    onClose();
  }
```

```
  return (
    <div className={classes.flexWrapper}>
      <div className={classes.chatInfoWrapper}>
        <img
          src={
            ctx.activeUser.imagePath
              ? `http://localhost:3000/${ctx.activeUser.imagePath}`
              : ProfileImage
          }
          alt=""
        />
        <p className={classes.name}>{ctx.activeUser.username}</p>
      </div>
      <div className={classes.iconLock} onClick={setIsSecretHandler}>
        {!flag && lock}
      </div>
      {modalIsVisible ? (
        <Modal>
          <div className={classes.confirmModal}>
            <h4>Make Chat Secret</h4>
            <p>Are you sure you want to make this chat secret?</p>
            <p>
              By making this chat secret, it will not be accessible with a fake
              password. If you enter the fake password during login, all
              messages in this chat will be deleted.
            </p>
            <div className={classes.actionButtons}>
              <button onClick={onClose}>Cancel</button>
              <button onClick={confirmIsSecret}>Confirm</button>
            </div>
          </div>
        </Modal>
      ) : null}
    </div>
  );
}
```

frontend/src/components/MainChat/InputMessage.jsx

```
import { useContext, useState } from 'react';
import classes from '../MainChat/InputMessage.module.css';
import { BsSend } from 'react-icons/bs';
import { UsersContext } from '../../store/users-context';
import { getToken } from '../../utils/localStorageManipulation';
import { encryptMessage } from '../../security/encryptMessage';
export default function InputMessage({ onSend, sharedKey }) {
  const ctx = useContext(UsersContext);
  const token = getToken();
  const [message, setMessage] = useState('');
  const buttonIsDisabled = !ctx.activeUser._id || message.trim().length === 0;



async function sendMessageHandler() {
    const encryptedMessage = await encryptMessage(message, sharedKey);
    const response = await fetch(
      `http://localhost:3000/message/send/${ctx.activeUser._id}`,
      {
        method: 'POST',
        headers: {
          Authorization: `Bearer ${token}`,
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ message: encryptedMessage }),
      }
```

```
    );
    const answer = await response.json();
    onSend({ ...answer.newMessage, message: message });
    setMessage('');
  }

  return (
    <>
      <div className={classes.wrapper}>
        <input
          type="text"
          placeholder="Message"
          value={message}
          onChange={(e) => {
            setMessage(e.target.value);
          }}
        />
        <div
          className={classes.sendBtn}
          onClick={buttonIsDisabled ? null : sendMessageHandler}
        >
          <BsSend className={classes.send} />
        </div>
      </div>
    </>
  );
}
```

frontend/src/components/MainChat/MainChat.jsx

```
import ChatInfo from './ChatInfo';
import classes from '../MainChat/MainChat.module.css';
import Message from './Message';
import InputMessage from './InputMessage';
import { useContext, useEffect, useState, useRef } from 'react';
import { UsersContext } from '../../store/users-context';
import { getPrivateKey, getToken } from '../../utils/localStorageManipulation';
import { SocketContext } from '../../store/socket-context';
import { generateSharedKey } from '../../security/sharedKey';
import { decryptMessage } from '../../security/decryptMessage';

export default function MainChat() {
  const ctx = useContext(UsersContext);
  const { socket } = useContext(SocketContext);
  const messagesEndRef = useRef(null);
  const [messages, setMessages] = useState([]);
  const [receiverPublicKey, setReceiverPublicKey] = useState('');
  const [sharedKey, setSharedKey] = useState();
  const token = getToken();
  const [privateKey, setPrivateKey] = useState(getPrivateKey());
  const [userInfo, setUserInfo] = useState();
  const [isSecret, setIsSecret] = useState(null);

  useEffect(() => {
    async function getReceiverPublicKey() {
      if (ctx.activeUser._id) {
        const response = await fetch(
          `http://localhost:3000/users/getPublicKey/${ctx.activeUser._id}`,
          { headers: { Authorization: `Bearer ${token}` } }
        );
        const receiverPublicKey = await response.json();
        if (receiverPublicKey.publicKey) {
          setReceiverPublicKey(JSON.parse(receiverPublicKey.publicKey));
        }
      }
```

```
  }
  getReceiverPublicKey();
}, [ctx.activeUser]);

useEffect(() => {
  async function getSharedKey() {
    try {
      if (receiverPublicKey && privateKey) {
        const sharedKey = await generateSharedKey(
          receiverPublicKey,
          privateKey
        );
        setSharedKey(sharedKey);
      }
    } catch (error) {
      console.log(error);
    }
  }
  getSharedKey();
}, [receiverPublicKey, privateKey]);

useEffect(() => {
  async function getUserInfo() {
    const response = await fetch(
      `http://localhost:3000/users/getLoggedUser`,
      {
        headers: {
          Authorization: `Bearer ${token}`,
        },
      }
    );
    const data = await response.json();
    setUserInfo(data.user);
  }
  getUserInfo();
}, [token]);

useEffect(() => {
  async function getMessages() {
    if (ctx.activeUser._id) {
      const response = await fetch(
        `http://localhost:3000/message/${ctx.activeUser._id}`,
        { headers: { Authorization: `Bearer ${token}` } }
      );
      // if (!response.ok) {
      //   console.log('no messages found');
      // }
      const messages = await response.json();
      const isSecret = messages.isSecret;
      setIsSecret(isSecret);

      if (sharedKey) {

         if (messages.chat && messages.chat.length > 0) {
          const decryptedMessages = await Promise.all(
            messages.chat.map(async (message) => {
              let copiedMessage = { ...message };
              copiedMessage.message = await decryptMessage(
                message.message,
                sharedKey
              );
              return copiedMessage;
            })
          );
          setMessages(decryptedMessages);
```

```
        } else {
          setMessages(messages.chat);
        }
      }
    }
  }
  getMessages();
}, [ctx.activeUser, token, sharedKey]);

useEffect(() => {
  if (socket) {
    socket.on('new-message', async (message) => {
      if (ctx.activeUser._id === message.senderId) {
        const decryptedMessage = await decryptMessage(
          message.message,
          sharedKey
        );
        setMessages((prevMessagges) => {
          return [
            ...prevMessagges,
            { ...message, message: decryptedMessage },
          ];
        });
      }
    });
  }
  return () => socket?.off('new-message');
}, [socket, messages, setMessages]);

useEffect(() => {
  if (messagesEndRef.current) {
    messagesEndRef.current.scrollIntoView({ behavior: 'auto' });
  }
}, [messages, ctx.activeUser]);

function addMessage(message) {
  setMessages((prevMessages) => {
    if (Array.isArray(prevMessages)) {
      return [...prevMessages, message];
    } else {
      return [message];
    }
  });
}

return (
  <div className={classes.mainPage}>
    {!ctx.activeUser._id ? (
      <p className={`${classes.messages} ${classes.fallbackText}`}>
        Select chat to start chatting
      </p>
    ) : (
      <>
        <ChatInfo isSecretChat={isSecret}></ChatInfo>
        {messages && messages.length > 0 ? (
          <div className={classes.messages}>
            {messages.map((message) => {
              return (
                <Message
                  userInfo={userInfo}
                  sharedKey={sharedKey}
                  key={message._id}
                  message={message}
                ></Message>
              );
```

```
              })}
              <div id="messages-end" ref={messagesEndRef} />
            </div>
          ) : (
            <p className={`${classes.messages} ${classes.fallbackText}`}>
              Send message to start chatting
            </p>
          )}

          <InputMessage
            sharedKey={sharedKey}
            onSend={addMessage}
          ></InputMessage>
        </>
      )}
    </div>
  );
}
```

frontend/src/components/MainChat/Message.jsx

```
import { useContext } from 'react';
import ProfileImage from '../../images/profile-image.png';
import classes from './Message.module.css';
import { UsersContext } from '../../store/users-context';
import formatTime from '../../utils/timeFormatter';

export default function Message({ message, userInfo }) {
  const ctx = useContext(UsersContext);
  const receiver = ctx.activeUser;
  const time = formatTime(message.createdAt);
  const messageFromOwner = message.receiverId === receiver._id;
  const receiverPic = receiver.imagePath
    ? `http://localhost:3000/${receiver.imagePath}`
    : ProfileImage;
  const senderPic = userInfo?.imagePath
    ? `http://localhost:3000/${userInfo.imagePath}`
    : ProfileImage;

  return (
    <div
      className={
        messageFromOwner
          ? `${classes.messageWrapper} ${classes.sender}`
          : `${classes.messageWrapper}`
      }
    >
      <img src={messageFromOwner ? senderPic : receiverPic} alt="" />
      <div className={classes.messageInfo}>
        <div className={classes.wrapperNameDate}>
          <p className={classes.name}>
            {messageFromOwner ? 'You' : receiver.username}
          </p>
          <p className={classes.time}>{time}</p>
        </div>
        <p className={classes.message}>{message.message}</p>
      </div>
    </div>
  );
}
```

frontend/src/components/SettingsBar/EditUser.jsx

```
import ProfileImage from '../../images/profile-image.png';
import { LuAsterisk } from 'react-icons/lu';
```

```
import classes from './EditUser.module.css';
import { useState } from 'react';
import { getFlag, getToken } from '../../utils/localStorageManipulation';
export default function EditUser({ onClose, userInfo, hideEditModal }) {
  const [showTooltip, setShowTooltip] = useState(false);
  const [image, setImage] = useState(null);
  const [validationError, setValidationError] = useState(null);
  const token = getToken();
  const flag = getFlag();
  const toggleTooltip = () => {
    setShowTooltip((prevTooltip) => !prevTooltip);
  };

  const changeImageHandler = (e) => {
    setImage(e.target.files[0]);
  };

  const currentUserPic = userInfo.imagePath
    ? `http://localhost:3000/${userInfo.imagePath}`
    : ProfileImage;

  async function editUserHandler(event) {
    event.preventDefault();
    const formData = new FormData(event.target);

    const response = await fetch(`http://localhost:3000/users/editUser`, {
      method: 'POST',
      headers: {
        Authorization: `Bearer ${token}`,
      },
      body: formData,
    });
    if (!response.ok) {
      const error = await response.json();
      setValidationError(error);
      return error;
    }

    await response.json();
    hideEditModal();
  }
  return (
    <div className={classes.editWrapper}>
      <h1>Edit profile</h1>
      <form onSubmit={editUserHandler}>
        <div className={classes.editPhotoWrapper}>
          <div>
            <h6>Profile photo</h6>
            <div className={classes.actionPicButtons}>
              <label htmlFor="file-upload" className={classes.customFileUpload}>
                Change
              </label>
              <input
                name="image"
                id="file-upload"
                type="file"
                accept="image/*"
                hidden
                onChange={changeImageHandler}
              />
              {/* <button onClick={deleteImageHandler}>Delete</button> */}
            </div>
          </div>
          <img
            className={classes.profilePic}
```

```
            src={image ? URL.createObjectURL(image) : currentUserPic}
            alt=""
          />
        </div>
        <div className={classes.line}></div>
        <div className={classes.errorMessage}>
          {validationError ? validationError.message : null}
        </div>
        <div className={classes.inputField}>
          <label htmlFor="username">Username</label>
          <input
            type="text"
            name="username"
            id="username"
            placeholder={userInfo.username}
          />
        </div>
        <div className={classes.inputField}>
          <label htmlFor="email">Email</label>
          <input
            type="text"
            name="email"
            id="email"
            disabled
            placeholder={userInfo.email}
          />
        </div>
        <div className={classes.inputField}>
          <label htmlFor="old-password">Old password</label>
          <input type="password" name="old-password" id="old-password" />
        </div>
        <div className={classes.inputField}>
          <label htmlFor="new-password">New password</label>
          <input type="password" name="new-password" id="new-password" />
        </div>
        {!flag && (
          <div className={classes.inputField}>
            <label htmlFor="fake-password">Fake password</label>
            <LuAsterisk
              className={classes.hint}
              onMouseEnter={toggleTooltip}
              onMouseLeave={toggleTooltip}
            />
            {showTooltip && (
              <div className={classes.tooltip}>
                Fake password is a special password that you can set for your
                account. Requiring a password adds an extra level of security,
                ensuring that only trusted individuals can access sensitive
                information. After entering the fake password during login, all
                messages in secret chats will be deleted. Be careful during
                login!
              </div>
            )}
            <input type="password" name="fake-password" id="fake-password" />
          </div>
        )}
        <div className={classes.actionBtn}>
          <button onClick={onClose}>Cancel</button>
          <button>Save</button>
        </div>
      </form>
    </div>
  );
}
```

frontend/src/components/SettingsBar/Logout.jsx

```
import { FiLogOut } from 'react-icons/fi';
import classes from '../SettingsBar/SettingsBar.module.css';
import { deleteDataFromStorage } from '../../utils/localStorageManipulation';

import { useNavigate } from 'react-router-dom';
export default function Logout() {
  const navigate = useNavigate();
  function logout() {
    deleteDataFromStorage();
    navigate('/auth/login');
  }
  return (
    <>
      <FiLogOut className={classes.iconLogout} onClick={logout} />
    </>
  );
}
```

frontend/src/components/SettingsBar/Modal.jsx

```
import classes from './Modal.module.css';
export default function Modal(props) {
  return (
    <>
      <div className={classes.backdrop} onClick={props.onClose}></div>
      <dialog open className={classes.modal}>
        {props.children}
      </dialog>
    </>
  );
}
```

frontend/src/components/SettingsBar/Settings.jsx

```
import { IoMdSettings } from 'react-icons/io';
import classes from './SettingsBar.module.css';

export default function Settings({ onOpen }) {
  return <IoMdSettings className={classes.iconSettings} onClick={onOpen} />;
}
```

frontend/src/components/SettingsBar/SettingsBar.jsx

```
import { getToken } from '../../utils/localStorageManipulation';
import classes from '../SettingsBar/SettingsBar.module.css';
import EditUser from './EditUser';
import Logout from './Logout';
import Modal from './Modal';
import Settings from './Settings';
import UserPic from './UserPic';

import { useState, useEffect } from 'react';
export default function SettingsBar() {
  const [modalIsVisible, setModalIsVisible] = useState(false);
  const [userInfo, setUserInfo] = useState({});
  const token = getToken();

  function hideEditModal() {
    setModalIsVisible(false);
  }
  function openEditModal() {
    setModalIsVisible(true);
  }
```

```
  useEffect(() => {
    async function getUserInfo() {
      if (modalIsVisible) {
        const response = await fetch(
          `http://localhost:3000/users/getEditUser`,
          {
            headers: { Authorization: `Bearer ${token}` },
          }
        );
        const data = await response.json();
        setUserInfo(data.user);
      }
    }
    getUserInfo();
  }, [token, modalIsVisible]);
  return (
    <div className={classes.main}>
      <UserPic></UserPic>
      {modalIsVisible ? (
        <Modal onClose={hideEditModal}>
          <EditUser
            hideEditModal={hideEditModal}
            userInfo={userInfo}
            onClose={hideEditModal}
          ></EditUser>
        </Modal>
      ) : null}
      <Settings onOpen={openEditModal}></Settings>
      <Logout></Logout>
    </div>
  );
}
```

frontend/src/components/SettingsBar/UserPic.jsx

```
import { useEffect, useState } from 'react';
import ProfileImage from '../../images/profile-image.png';
import classes from './SettingsBar.module.css';
import { getToken } from '../../utils/localStorageManipulation';

export default function UserPic() {
  const [userInfo, setUserInfo] = useState();
  const token = getToken();
  useEffect(() => {
    async function getUserInfo() {
      const response = await fetch(
        `http://localhost:3000/users/getLoggedUser`,
        {
          headers: {
            Authorization: `Bearer ${token}`,
          },
        }
      );
      const data = await response.json();
      setUserInfo(data.user);
    }
    getUserInfo();
  }, [token]);
  return (
    <img
      src={
        userInfo && userInfo.imagePath
          ? `http://localhost:3000/${userInfo.imagePath}`
          : ProfileImage
```

```
      }
      className={classes.userPic}
      alt=""
    />
  );
}
```

frontend/src/components/Sidebar/SearchPanel.jsx

```
import classes from './SearchPanel.module.css';
import { BiSearch } from 'react-icons/bi';

export default function SearchPanel({ search, setSearch, findChats }) {
  return (
    <>
      <div className={classes.search}>
        <input
          type="text"
          placeholder="Search"
          value={search}
          onChange={(e) => {
            setSearch(e.target.value);
          }}
        />
        <BiSearch className={classes.icon} onClick={findChats} />
      </div>
    </>
  );
}
```

frontend/src/components/Sidebar/Sidebar.jsx

```
import SearchPanel from './SearchPanel';
import classes from './Sidebar.module.css';
import UserChat from './UserChat';
import { useState } from 'react';
import { useContext } from 'react';
import { UsersContext } from '../../store/users-context';

export default function Sidebar() {
  const ctx = useContext(UsersContext);
  const [search, setSearch] = useState('');
  const [users, setUsers] = useState(ctx.sidebarUsers.users);

  function findChats() {
    if (!search || !search.trim()) {
      setUsers(ctx.sidebarUsers.users);
      return;
    }
    const filteredUsers = users.filter((user) =>
      user.username.toLowerCase().startsWith(search.toLowerCase())
    );
    setUsers(filteredUsers);
  }
  return (
    <div className={classes.sidebarContainer}>
      <SearchPanel
        search={search}
        setSearch={setSearch}
        findChats={findChats}
      ></SearchPanel>
      <div className={classes.chats}>
        {users && users.length > 0 ? (
          users.map((user) => <UserChat key={user._id} user={user}></UserChat>)
        ) : (
```

```
          <p className={classes.fallbackText}>No user found</p>
        )}
      </div>
    </div>
  );
}
```

frontend/src/components/Sidebar/UserChat.jsx

```
import { useContext } from 'react';
import ProfileImage from '../../images/profile-image.png';
import classes from './UserChat.module.css';
import { UsersContext } from '../../store/users-context';
import { SocketContext } from '../../store/socket-context';
export default function UserChat({ user }) {
  const { activeUser, selectChatHandler } = useContext(UsersContext);
  const { onlineUsers } = useContext(SocketContext);
  const isOnline = onlineUsers.find((userId) => userId === user._id);
  return (
    <div
      className={`${classes.userChatContainer}
      ${activeUser._id === user._id ? classes.selected : ''}
        `}
      onClick={() => selectChatHandler(user)}
    >
      <div className={classes.imageContainer}>
        <img
          src={
            user.imagePath
              ? `http://localhost:3000/${user.imagePath}`
              : ProfileImage
          }
          alt=""
        />
        <div className={isOnline ? classes.online : null}></div>
      </div>
      <div className={classes.userInfo}>
        <p className={classes.name}>{user.username}</p>
        {/* <p className={classes.lastMessage}></p> */}
      </div>
    </div>
  );
}
```

frontend/src/pages/Error.jsx

```
export default function Error() {
  return (
    <div>
      <h1>An error occured!</h1>
      <p>Could not find this page!</p>
    </div>
  );
}
```

frontend/src/pages/LoginPage.jsx

```
import LoginForm from '../components/Auth/LoginForm';
import classes from './AuthPage.module.css';
import { validateEmail, validatePassword } from '../utils/validateInput';
import {
  getPrivateKey,
  getPublicKey,
  storeData,
} from '../utils/localStorageManipulation';
```

```
import { redirect } from 'react-router-dom';
import { generateKeyPair } from '../security/keyPairGeneration';

export default function LoginPage() {
  return (
    <div className={classes.authPage}>
      <LoginForm />
    </div>
  );
}

export async function action({ request }) {
  const data = await request.formData();
  const submitData = {
    email: data.get('email'),
    password: data.get('password'),
  };

  const emailIsValid = validateEmail(submitData.email);
  const passwordIsValid = validatePassword(submitData.password, null);

  if (!emailIsValid || !passwordIsValid) {
    return { emailIsValid, passwordIsValid };
  }
  let privateKeyJwk;
  let publicKeyJwk;
  try {
    privateKeyJwk = getPrivateKey();
    publicKeyJwk = getPublicKey();
    if (!privateKeyJwk || !publicKeyJwk) {
      ({ publicKeyJwk, privateKeyJwk } = await generateKeyPair());
      console.log(publicKeyJwk, privateKeyJwk);
    }
  } catch (error) {
    console.log(error);
  }

  const response = await fetch('http://localhost:3000/auth/login', {
    method: request.method,
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      submitData,
      publicKeyJwk: JSON.stringify(publicKeyJwk),
    }),
  });

  if (!response.ok) {
    return response;
  }

  const resultData = await response.json();
  const { userId, token, isFake } = resultData;

  storeData(userId, token);
  localStorage.setItem('privateKey', JSON.stringify(privateKeyJwk));
  localStorage.setItem('publicKey', JSON.stringify(publicKeyJwk));
  localStorage.setItem('flag', isFake);
  const expiration = new Date();
  expiration.setTime(expiration.getTime() + 7 * 24 * 60 * 60 * 1000);
  localStorage.setItem('tokenExpiration', expiration.toISOString());
  return redirect('/user');
}
```

frontend/src/pages/Main.jsx

```
import classes from './Main.module.css';
import Sidebar from '../components/Sidebar/Sidebar';
import MainChat from '../components/MainChat/MainChat';
import {
  deleteDataFromStorage,
  getToken,
} from '../utils/localStorageManipulation';
import { UsersContext } from '../store/users-context';
import { redirect, useLoaderData } from 'react-router-dom';
import { useState } from 'react';
import SettingsBar from '../components/SettingsBar/SettingsBar';
import SocketContextProvider from '../store/socket-context';

export default function Main() {
  const sidebarUsers = useLoaderData();
  const [activeChat, setActiveChat] = useState({});
  function selectChatHandler(user) {
    setActiveChat(user);
  }
  const ctx = {
    sidebarUsers: sidebarUsers,
    activeUser: activeChat,
    selectChatHandler: selectChatHandler,
  };
  return (
    <UsersContext.Provider value={ctx}>
      <SocketContextProvider>
        <div className={classes.container}>
          <SettingsBar></SettingsBar>
          <Sidebar></Sidebar>
          <MainChat></MainChat>
        </div>
      </SocketContextProvider>
    </UsersContext.Provider>
  );
}

export async function loader({ request }) {
  const token = getToken();
  if (token === 'Token is expired') {
    deleteDataFromStorage();
    return redirect('/auth/login');
  }
  if (!token) {
    return redirect('/auth/login');
  }
  const response = await fetch(`http://localhost:3000/users/`, {
    headers: {
      method: request.method,
      Authorization: `Bearer ${token}`,
    },
  });

  return response;
}
```

frontend/src/pages/SignupPage.jsx

```
import SignupForm from '../components/Auth/SignupForm';
import classes from './AuthPage.module.css';
import {
  validateEmail,
  validatePassword,
  validateUsername,
} from '../utils/validateInput';
```

```
import { redirect } from 'react-router-dom';
export default function SignupPage() {
  return (
    <div className={classes.authPage}>
      <SignupForm />
    </div>
  );
}

export async function action({ request }) {
  const data = await request.formData();
  const submitData = {
    email: data.get('email'),
    username: data.get('username'),
    password: data.get('password'),
    confirmedPassword: data.get('confirmed-password'),
  };

  const emailIsValid = validateEmail(submitData.email);
  const usernameIsValid = validateUsername(submitData.username);
  const passwordIsValid = validatePassword(
    submitData.password,
    submitData.confirmedPassword
  );

  if (!emailIsValid || !passwordIsValid || !usernameIsValid) {
    return { emailIsValid, passwordIsValid, usernameIsValid };
  }

  const response = await fetch('http://localhost:3000/auth/signup', {
    method: request.method,
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(submitData),
  });

  if (!response.ok) {
    console.log(response);
    // throw json({ message: response.message }, { status: 500 });
    return response;
  }
  return redirect('/auth/login');
}
```

frontend/src/security/decryptMessage.js

```
export async function decryptMessage(message, sharedKey) {
  try {
    const decryptedBase64 = atob(message);
    const decryptedBytes = new Uint8Array(
      [...decryptedBase64].map((char) => char.charCodeAt(0))
    );
    const decryptedMessage = await window.crypto.subtle.decrypt(
      {
        name: 'AES-GCM',
        iv: new Uint8Array(13),
      },
      sharedKey,
      decryptedBytes
    );

    const decodedMessage = new TextDecoder().decode(decryptedMessage);

    return decodedMessage;
  } catch (error) {
    console.log(error);
```

```
    return `Decryption failed. The key may be incompatible.`;
  }
}
```

frontend/src/security/encryptMessage.js

```
export async function encryptMessage(message, sharedKey) {
  const encodedMessage = new TextEncoder().encode(message);
  const encryptedMessage = await window.crypto.subtle.encrypt(
    { name: 'AES-GCM', iv: new Uint8Array(13) },
    sharedKey,
    encodedMessage
  );

  const encryptedBytes = new Uint8Array(encryptedMessage);
  const encryptedText = String.fromCharCode.apply(null, encryptedBytes);
  const encryptedBase64 = btoa(encryptedText);

  return encryptedBase64;
}
```

frontend/src/security/keyPairGeneration.js

```
export async function generateKeyPair() {
  const keys = await window.crypto.subtle.generateKey(
    {
      name: 'ECDH',
      namedCurve: 'P-384',
    },
    true,
    ['deriveKey']
  );

  const publicKeyJwk = await window.crypto.subtle.exportKey(
    'jwk',
    keys.publicKey
  );

  const privateKeyJwk = await window.crypto.subtle.exportKey(
    'jwk',
    keys.privateKey
  );

  return { publicKeyJwk, privateKeyJwk };
}
```

frontend/src/security/sharedKey.js

```
export async function generateSharedKey(publicKeyJwk, privateKeyJwk) {
  const publicKey = await window.crypto.subtle.importKey(
    'jwk',
    publicKeyJwk,
    {
      name: 'ECDH',
      namedCurve: 'P-384',
    },
    true,
    []
  );

  const privateKey = await window.crypto.subtle.importKey(
    'jwk',
    privateKeyJwk,
    {
      name: 'ECDH',
```

```
      namedCurve: 'P-384',
    },
    true,
    ['deriveKey']
  );

  return await window.crypto.subtle.deriveKey(
    { name: 'ECDH', public: publicKey },
    privateKey,
    { name: 'AES-GCM', length: 128 },
    true,
    ['encrypt', 'decrypt']
  );
}
```

frontend/src/store/socket-context.jsx

```
import { useState, createContext, useEffect } from 'react';
import { getToken, getUserId } from '../utils/localStorageManipulation';
import openSocket from 'socket.io-client';

export const SocketContext = createContext({
  onlineUsers: [],
  socket: null,
});

export default function SocketContextProvider({ children }) {
  const [socket, setSocket] = useState(null);
  const [onlineUsers, setOnlineUsers] = useState([]);
  const token = getToken();
  const userId = getUserId();

  useEffect(() => {
    if (token) {
      const socket = openSocket('http://localhost:3000', {
        query: {
          userId,
        },
      });

      setSocket(socket);

      socket.on('online-users', (onlineIds) => {
        setOnlineUsers(Object.keys(onlineIds));
      });
      return () => socket.close();
    } else {
      if (socket) {
        socket.close();
        setSocket(null);
      }
    }
  }, [token]);

  const ctx = {
    onlineUsers: onlineUsers,
    socket: socket,
  };
  return (
    <SocketContext.Provider value={ctx}>{children}</SocketContext.Provider>
  );
}
```

frontend/src/store/users-context.jsx

```
import { createContext } from 'react';

export const UsersContext = createContext({
  sidebarUsers: [],
  activeUser: {},
  selectChatHandler: () => {},
});
```

frontend/src/utils/localStorageManipulation.js

```
export function storeData(userId, token) {
  localStorage.setItem('token', token);
  localStorage.setItem('userId', userId);
}

export function deleteDataFromStorage() {
  localStorage.removeItem('token');
  localStorage.removeItem('userId');
  localStorage.removeItem('tokenExpiration');
}

export function getPrivateKey() {
  const privateKey = localStorage.getItem('privateKey');
  const parsedPrivateKey = JSON.parse(privateKey);
  return parsedPrivateKey;
}

export function getPublicKey() {
  const publicKey = localStorage.getItem('publicKey');
  const parsedPublicKey = JSON.parse(publicKey);
  return parsedPublicKey;
}

export function getFlag() {
  const flag = localStorage.getItem('flag');
  return flag === 'true';
}

export function getDuration() {
  const expiration = localStorage.getItem('tokenExpiration');
  const expirationDate = new Date(expiration);
  const now = new Date();
  const duration = expirationDate.getTime() - now.getTime();
  return duration;
}

export function getToken() {
  const token = localStorage.getItem('token');
  if (!token) {
    return null;
  }
  const duration = getDuration();
  if (duration < 0) {
    return 'Token is expired';
  }
  return token;
}

export function checkAuth() {
  const token = getToken();
  if (!token || token === 'Token is expired') {
    return redirect('/auth/login');
  }
  return null;
}
```

```
export function getUserId() {
  const userId = localStorage.getItem('userId');
  return userId;
}
```

frontend/src/utils/timeFormatter.js

```
export default function formatTime(time) {
  const date = new Date(time);
  const hours = date.getHours().toString().padStart(2, '0');
  const minutes = date.getMinutes().toString().padStart(2, '0');
  return `${hours}:${minutes}`;
}
```

frontend/src/utils/validateInput.js

```
export function validateEmail(email) {
  const index = email.indexOf('@');
  const dotIndex = email.substring(index + 1);
  if (!email) {
    return false;
  } else if (index === -1 || index === 0 || index === email.length - 1) {
    return false;
  } else if (dotIndex.indexOf('.') === -1) {
    return false;
  }
  return true;
}

export function validatePassword(password, corfirmedPassword) {
  if (!password) {
    return false;
  } else if (password.length < 8) {
    return false;
  } else if (corfirmedPassword) {
    if (password !== corfirmedPassword) {
      return false;
    }
  }
  return true;
}

export function validateUsername(username) {
  if (!username.trim()) {
    return false;
  }
  return true;
}
```