КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій
(факультет)

Кафедра кібербезпеки та комп'ютерної інженерії
(назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

на тему:

Система захищеного обміну повідомленнями в реальному часі (веб-додаток)

Чаюк Сергій Олександрович
(прізвище, ім'я та по батькові здобувача повністю)

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Автоматизації і інформаційних технологій

(факультет)

Кафедра кібербезпеки та комп'ютерної інженерії

(назва кафедри)

ЗАТВЕРДЖУЮ

Завідувач кафедри

к.т.н., доцент Максим ДЕЛЕМБОВСЬКИЙ

_____

„___" _____20___ року

КВАЛІФІКАЦІЙНА РОБОТА ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

Система захищеного обміну повідомленнями в реальному часі (веб-додаток)

(назва)

Я як здобувач вищої освіти КНУБА розумію і підтримую політику закладу з академічної доброчесності. Я не надавав(-ла) і не одержував(-ла) недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Здобувач Чаюк Сергій Олександрович

(прізвище, ім'я та по батькові повністю)

125 Кібербезпека та захист інформації

(спеціальність)

Безпека інформаційних і комунікаційних систем

(освітня програма)

Група БІКСм-24

Керівник Делембовський М.М.

(прізвище та ініціали)

Кандидат технічних наук, доцент

(вчене звання, науковий ступінь)

Рецензент _____

(прізвище та ініціали)

Ідентичність підтверджую

Київ 2025 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій
Кафедра: кібербезпеки та комп'ютерної інженерії
Освітній рівень: магістр
Спеціальність: 125 кібербезпека та захист інформації
ОПП: безпека інформаційних і комунікаційних систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
к.т.н., доцент Максим ДЕЛЕМБОВСЬКИЙ

_____
„___" _____20___ року

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧА СТУПЕНЯ ВИЩОЇ ОСВІТИ МАГІСТР

Чаюк Сергій Олександрович

(прізвище, ім'я та по батькові здобувача)

1. Тема роботи Система захищеного обміну повідомленнями в реальному часі (веб-додаток), затверджена наказом ректора КНУБА №1635/23.2/25 від 30 вересня 2025 року
2. Керівник роботи Делембовський Максим Михайлович, канд. техн. наук, доцент кафедри кібербезпеки та комп'ютерної інженерії
(прізвище, ім'я та по батькові, науковий ступінь, вчене звання)
3. Термін подання здобувачем роботи до захисту 15 грудня 2025 року
4. Зміст пояснювальної записки за розділами:
Р. 1. Аналіз предметної області та постановка задачі
Р. 2. Методологія та засоби проектування захищеної веб-системи
Р. 3. Архітектурні та проектні рішення для захищеного обміну даними
5. Графічний матеріал за розділами:
Р. 1. рис. 4, табл. 2
Р. 2. рис. 12, табл. 1
Р. 3. рис. 23, табл. 10

6. Консультанти розділів кваліфікаційної випускної роботи

Розділи	Прізвища, ініціали та посади консультанта	Перевірив	
		дата	підпис
Розділ 1.			
Розділ 2.			
Розділ 3.			

7. Календарний план виконання роботи:

Види робіт та їх зміст	Дата виконання
Розділ 1.	20.10.2025 р.
Розділ 2.	16.11.2025 р.
Розділ 3.	09.12.2025 р.
Остаточне оформлення роботи	11.12.2025 р.
Направлення роботи на рецензування, перевірку на плагіат	12.12.2025 р.
Попередній захист роботи на кафедрі	15.12.2025 р.

8. Дата видачі завдання: 30 вересня 2025 року

Керівник _____ Делембовський М.М.
(підпис) (прізвище та ініціали)

Здобувач _____ Чаюк С.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Чаюк С.О. “Система захищеного обміну повідомленнями в реальному часі (веб-додаток)”.

Магістерська робота присвячена вирішенню проблеми забезпечення конфіденційності та цілісності даних у веб-орієнтованих системах комунікації. Об’єктом дослідження є процеси наскрізного шифрування в умовах недовіреного серверного середовища.

У роботі проведено аналіз існуючих протоколів та обгрунтовано доцільність використання гібридної криптографічної архітектури. Запропоновано та спроектовано архітектуру системи з принципами нульовим розголошенням, де сервер виконує роль виключно ретранслятора зашифрованих даних.

Розроблено програмний прототип ядра шифрування мовою TypeScript з використанням нативного Web Cryptography API. Були реалізовано такі механізми: генерації та безпечного зберігання ключів з використанням PBKDF2, шифрування масивів даних алгоритмом AES-GCM, інкапсуляції ключів за стандартом HPKE (RFC 9180), а також синхронізації історії повідомлень для відправника.

Результати роботи підтверджують можливість побудови ефективних та безпечних веб-додатків для обміну повідомленнями, які не поступаються нативним аналогам та забезпечують високий рівень захисту без необхідності довіри до інфраструктури провайдера послуг.

Ключові слова: наскрізне шифрування, інформаційна безпека, гібридне шифрування, Web Cryptography API, веб-додаток, PBKDF2, HPKE, AES-GCM.

SUMMARY

Chaiuk S.O. “Secure real-time messaging system (web application)”.

The master’s thesis is devoted to solving the problem of ensuring data confidentiality and integrity in web-based communication systems. The object of the study is the processes of End-to-End Encryption within an untrusted server environment.

The thesis analyzes existing protocols and substantiates the feasibility of using a hybrid cryptographic architecture. An architecture with “Zero-Knowledge” principles is proposed and designed, where the server acts exclusively as a relay for encrypted data.

A software prototype of the encryption core was developed using TypeScript and the native Web Cryptography API. The following mechanisms were implemented: key generation and secure storage using PBKDF2, encryption of datasets using the AES-GCM algorithm, key encapsulation according to the HPKE standard (RFC 9180), and message history synchronization for the sender.

The results confirm the feasibility of building efficient and secure web messaging applications that match native counterparts and provide a high level of protection without requiring trust in the service provider’s infrastructure.

Keywords: End-to-End Encryption, information security, hybrid encryption, Web Cryptography API, web application, PBKDF2, HPKE, AES-GCM.

РЕЗЮМЕ (SUMMARY) *до кваліфікаційної випускової роботи здобувача*	ПІБ *Чаюк Сергій Олександрович* *Chaiuk Serhii Oleksandrovych*		
ЗВО	Київський національний університет будівництва і архітектури		
Тема *(українською та англійською)*	Система захищеного обміну повідомленнями в реальному часі (веб-додаток) Secure real-time messaging system (web application)		
Освітній ступінь	Магістр		
Факультет	Автоматизації і інформаційних технологій		
Випускова кафедра	Кібербезпеки та комп'ютерної інженерії		
Спеціальність	125 кібербезпека та захист інформації		
Освітня програма	Безпека інформаційних і комунікаційних систем		
Керівник	Делембовський Максим Михайлович		
Обсяг роботи:	*Пояснювальна записка, стор.*	*Розділів*	*Презентація, кількість слайдів*
	122	3	30
Розділ 1	Аналіз предметної області та постановка задачі.		
Розділ 2	Методологія та засоби проектування захищеної веб-системи.		
Розділ 3	Архітектурні та проектні рішення для захищеного обміну даними в веб-середовищі.		
Висновки по роботі	У роботі спроектовано захищену клієнт-серверну архітектуру. Обгрунтовано та реалізовано гібридну архітектуру шифрування на базі HPKE та AES-GCM з використанням WebCryptoAPI. Вирішено задачі безпечного управління ключами та синхронізації сесій.		
Ключові слова: Keywords:	E2EE, гібридне шифрування, WebCryptoAPI, веб-додаток, PBKDF2, HPKE, AES-GCM E2EE, hybrid encryption, WebCryptoAPI, web application, PBKDF2, HPKE, AES-GCM		

Здобувач _____/ _____

Керівник _____/ _____

ЗМІСТ

ВСТУП

У сучасну цифрову епоху, коли обмін інформацією в реальному часі став ключовим інструментом для особистих та ділових комунікацій, питання захисту цифрового суверенітету та приватності набуло безпрецедентної гостроти. Глобальна статистика свідчить про невпинне зростання кіберзагроз. Лише за першу половину 2025 року було зафіксовано один з найбільших в історії витоків даних, що скомпрометував понад 16 мільярдів облікових записів від провідних сервісів, таких як Google, Apple, Telegram та інші [1].

Для України, яка перебуває в стані війни, ця проблема є ще більш нагальною. Кількість цілеспрямованих кібератак на державні органи та критичну інфраструктуру стрімко зростає. За даними Держспецзв'язку, якщо у 2023 році було зафіксовано близько 2500 кіберінцидентів, то у 2024 цей показник зріс майже на 70%, а з початку 2025 року фіксується в середньому 15 кібератак щодня [2, 3]. Водночас, месенджери стали основним каналом отримання новин та спілкування для абсолютної більшості українців. Станом на 2025 рік, Telegram використовують близько 88% українських користувачів смартфонів, а Viber — 72% [4]. Це перетворює їх на стратегічну ціль для зловмисників, що прагнуть отримати доступ до конфіденційної інформації.

На цьому тлі, технологія наскрізного шифрування E2EE є фундаментальною вимогою для будь-якої сучасної комунікаційної платформи. Протокол Signal, який став "золотим стандартом" для захисту розмов віч-на-віч, довів свою ефективність і був прийнятий за основу в Signal таWhatsApp [5]. Проте, коли мова заходить про групові чати, його архітектура, що базується на встановленні парних захищених сеансів між усіма учасниками, стикається з проблемами масштабування. Оновлення ключів чи зміна складу групи вимагає кількості операцій, що зростає лінійно (або навіть квадратично) відносно кількості учасників, що робить його неефективним для великих спільнот.

Саме для вирішення цієї проблеми IETF розробила та стандартизувала новий протокол — Messaging Layer Security (MLS) [6]. На відміну від Signal, MLS

використовує деревоподібну структуру ключів, що дозволяє оновлювати спільний секрет групи з логарифмічною складністю. Це робить його значно ефективнішим для груп, що налічують багато учасників і закладає основу для нового покоління захищених корпоративних та публічних комунікаційних систем. Паралельно, стандартизація високорівневих криптографічних примітивів, як-от Hybrid Public Key Encryption, спрощує створення надійних та перевірених криптосистем, мінімізуючи ризик помилок імплементації [7].

Таким чином, розробка системи захищеного обміну повідомленнями, яка б поєднувала надійність перевірених підходів із сучасними стандартизованими та ефективними протоколами, є надзвичайно актуальною науково-практичною задачею. Це дозволить не лише спроєктувати безпечний інструмент, але й дослідити архітектурні переваги нових стандартів для побудови масштабованих систем.

Метою даної дипломної роботи є проектування архітектури та механізму шифрування в веб-середовищі для захищеного обміну повідомленнями в реальному часі, що базується на сучасних криптографічних стандартах та забезпечує високий рівень конфіденційності, цілісності та автентичності даних.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- Проаналізувати сучасні загрози безпеці систем обміну повідомленнями та дослідити існуючі протоколи E2EE і алгоритми, що лежать в їх основі.
- Обгрунтувати вибір криптографічних алгоритмів та примітивів (симетричне/асиметричне шифрування, гешування, інкапсуляція ключів) з урахуванням можливостей веб-розробки.
- Спроектувати архітектуру механізму шифрування з адаптацією в веб-середовищі, що підтримує обмін повідомленнями в реальному часі та реалізує E2EE.
- Розробити прототип, що реалізує спроектовану архітектуру, з використанням сучасних веб-технологій.

- Провести тестування для перевірки функціональності та оцінки рівня захищеності.

Об’єктом дослідження є процес захищеного обміну даними в реальному часі, що реалізується в клієнт-серверних веб-системах.

Предметом дослідження є криптографічні методи, протоколи та архітектурні рішення для побудови веб-орієнтованих систем обміну повідомленнями з наскрізним шифруванням.

Аналіз науково-технічної літератури та існуючих рішень показує, що існують значні можливості для вдосконалення та дослідження.

Провідні протоколи, як Signal Protocol, є надзвичайно ефективними для розмов 1-на-1, але їхня логіка є складною та монолітною. Розробка, що базується на більш новому, модульному та стандартизованому підході як HPKE, може запропонувати простішу та більш гнучку архітектуру, яку легше верифікувати.

Як зазначалося, підхід Signal до групових чатів не є оптимальним для великих спільнот. Новий стандарт MLS був створений спеціально для вирішення цієї проблеми, але його практичні реалізації у веб-додатках ще не є поширеними. Дана робота заповнює цю прогалину, досліджуючи архітектуру такої системи.

Також, більшість популярних сервісів є централізованими системами із закритим вихідним кодом. Існує попит на відкриті, прозорі та гнучкі архітектури, які можна було б легко адаптувати для корпоративних потреб, інтегрувати з іншими системами та розгортати на власній інфраструктурі.

Дана дипломна робота спрямована на розробку відкритої архітектури, що використовує останні досягнення в стандартизації криптографії для вирішення реальних проблем масштабованості та гнучкості.

Для вирішення поставлених завдань у роботі використано комплекс методів: системний аналіз для огляду предметної області та існуючих рішень; об’єктно-орієнтоване проєктування для розробки архітектури системи; метод прототипування для ітеративної розробки програмного продукту; емпіричні методи, зокрема тестування, для перевірки працездатності та аналізу результатів.

Наукова новизна одержаних результатів полягає в адаптації та застосуванні комплексного поєднання криптографічних протоколів і стандартів для організації захищеного сеансу обміну повідомленнями в архітектурі веб-додатку. На відміну від класичних реалізацій E2EE, що часто використовують власні комплексні протоколи, запропонована модель базується на поєднанні декількох механізмів, що спрощує реалізацію та аудит, і використовує Web Cryptography API для виконання всіх криптографічних операцій на стороні клієнта. Це забезпечує надійне наскрізне шифрування в браузері, мінімізує довіру до сервера та підвищує загальний рівень конфіденційності.

Результати роботи можуть бути використані для створення захищених корпоративних систем комунікацій. Розроблені архітектура і прототип шифрування можуть бути використані для побудови комерційних продуктів або як освітній інструмент для демонстрації принципів роботи сучасних криптографічних систем. Відповідні рішення та модулі можуть бути інтегровані в існуючі веб-системи для підвищення їх безпеки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис проблеми та постановка задачі

Сучасний ландшафт цифрових комунікацій характеризується двома ключовими, але фундаментально суперечливими тенденціями. З одного боку, спостерігається нестримна міграція користувацької та корпоративної активності у веб-середовище. На відміну від нативних додатків, які потребують інсталяції та розробляються окремо для кожної платформи (iOS, Android, Windows), веб-додатки пропонують універсальну доступність [8]. Вони функціонують безпосередньо у браузері, що забезпечує крос-платформну сумісність “за замовчуванням”, миттєве розгортання оновлень без участі користувача та легкість доступу з будь-якого пристрою. Ця зручність зробила “web-first” підхід домінуючим для багатьох сервісів, включно з системами обміну повідомленнями [9].

З іншого боку, ця зручність має високу ціну з точки зору безпеки. Паралельно з ростом веб-платформ відбувається експоненційне зростання кіберзагроз. Атаки на ланцюги постачання, цілеспрямоване державне стеження та компрометація серверної інфраструктури стали повсякденною реальністю. У відповідь на ці загрози, наскрізне шифрування (End-to-End Encryption, E2EE) перетворилося з нішевої функції на базову вимогу для будь-якої комунікаційної системи, що претендує на конфіденційність. Модель E2EE гарантує, що повідомлення шифруються на пристрої відправника і можуть бути розшифровані виключно на пристрої одержувача. Це робить сервер “сліпим” ретранслятором, який не має доступу до відкритого тексту повідомлень, навіть якщо він буде зламаний або юридично примушений до співпраці [10].

Основна проблема, що вирішується у даній роботі, полягає у фундаментальному конфлікті між вимогою абсолютної безпеки та архітектурною природою сучасних веб-додатків. На відміну від нативного додатку, код якого встановлюється один раз і може бути перевірений, наприклад через механізми

відтворюваних збірок, код веб-додатку завантажується з сервера щоразу при ініціалізації сесії [11]. Це створює фатальну проблему довіри, яка детально описується нижче.

Технічне ядро проблеми: вороже середовище веб-браузера

Для реалізації E2EE у вебі, всі криптографічні операції (генерування ключів, шифрування, розшифрування) мають виконуватися на стороні клієнта, тобто безпосередньо в браузері. Це покладається на дві ключові технології: Web Cryptography API (WebCrypto) для виконання криптографічних операцій та механізми зберігання, як IndexedDB або Local Storage, для персистентності ключів [12]. Однак ця архітектура є вразливою за своєю суттю:

1. Загроза модифікації коду “на льоту”. Оскільки E2EE загрожує бізнес-моделям, що базуються на зборі даних або перешкоджає державному нагляду, сам сервер стає головним вектором атаки [11]. Зловмисний або скомпрометований сервер може в будь-який момент надіслати клієнту модифікований JavaScript-код. Цей код, виконуючись у тому ж контексті, що й легітимний, може:
   - Перехопити приватні ключі користувача, щойно вони завантажуються зі сховища у пам’ять.
   - Захопити відкритий текст повідомлень перед шифруванням або одразу після розшифрування.
   - Незаконно “злити” ці дані на сторонній сервер.
2. Відсутність безпечного сховища ключів. Хоча Web Cryptography API дозволяє генерувати ключі з атрибутом extractable: false, сподіваючись замкнути їх у браузері, це не є аналогом апаратного модуля безпеки HSM або Secure Enclave [13]. Ключі все одно існують у пам’яті процесу браузера і, хоч і не можуть бути експортовані через API, залишаються вразливими до атак на пам’ять або до модифікованого JavaScript-коду, який може використовувати цей ключ для розшифрування даних та

подальшої їх крадіжки. Сховища IndexedDB/Local Storage за своєю природою не є захищеними від доступу з боку JavaScript-коду того ж походження.

3. Вразливості самого веб-додатку. Навіть якщо припустити, що сервер є чесним, веб-додатки за своєю природою мають значно більшу поверхню атаки, ніж нативні. Будь-яка вразливість типу Cross-Site Scripting (XSS), навіть у сторонньому компоненті, може дозволити зловмиснику інжектувати свій скрипт і виконати ті ж самі дії: викрасти ключі або перехопити повідомлення.

Таким чином, будь-яка веб-система E2EE повинна виходити з моделі загроз, де сервер не є довіреним, а клієнтський код є потенційно скомпрометованим. Це створює набагато вищу планку для проектування безпечної архітектури порівняно з нативними системами.

Друга складова проблеми: неефективність масштабування існуючих протоколів

Протягом останнього десятиліття “золотим стандартом” для E2EE-комунікацій став протокол Signal, який поєднує протокол початкового встановлення ключів X3DH (Extended Triple Diffie-Hellman) та алгоритм “подвійного храповика” (Double Ratchet Algorithm). Ця комбінація забезпечує найсильніші гарантії безпеки: досконалу пряму секретність (Forward Secrecy) та безпеку після компрометації (Post-Compromise Security) для розмов 1-на-1.

Однак, коли мова заходить про групові чати, архітектура Signal виявляє значні проблеми з масштабуванням. Класичний підхід Signal до груп базується на використанні множинних парних (pairwise) сеансів Double Ratchet. Коли учасник надсилає повідомлення у групу, він фактично шифрує його окремо для кожного іншого учасника групи та надсилає ці окремі копії [14].

Це призводить до лінійної складності O(n) для операцій оновлення складу групи:

- Додавання учасника: Кожен існуючий учасник групи (n) повинен встановити новий парний сеанс з новим учасником.
- Видалення учасника: Група має згенерувати новий “груповий ключ”, і цей ключ має бути безпечно розповсюджений кожному з n−1 учасників, що залишилися, знову ж таки через їхні парні сеанси [15].

Для малих груп це є прийнятним, але для великих корпоративних чи публічних спільнот, що можуть налічувати сотні або тисячі учасників, така модель стає обчислювально та мережево неефективною, призводячи до значних затримок та навантаження.

Третя складова проблеми: складність нових стандартів та відсутність гнучких примітивів

Для вирішення проблеми масштабування груп, робоча група IETF (Internet Engineering Task Force) розробила та у 2023 році стандартизувала Messaging Layer Security (RFC 9420). MLS є проривним протоколом, який використовує асиметричну деревоподібну структуру ключів. Ця архітектура дозволяє виконувати операції додавання та видалення учасників з логарифмічною складністю O(logn), що робить його ідеальним для груп будь-якого розміру, аж до десятків тисяч [16]. Однак, MLS є надзвичайно складним, монолітним протоколом. Його повноцінна імплементація та формальна верифікація є масштабною задачею, особливо для веб-середовища з його обмеженнями.

На іншому полюсі знаходиться інший важливий стандарт IETF — Hybrid Public Key Encryption (RFC 9180) [17]. На відміну від Signal або MLS, HPKE не є повноцінним протоколом керування сеансами. Натомість, це високорівневий, модульний “будівельний блок”. Він стандартизує комбінацію трьох примітивів:

- Механізм інкапсуляції ключів (KEM).
- Функція виведення ключа (KDF).
- Схема автентифікованого шифрування з додатковими даними (AEAD).

Перевага HPKE полягає у його гнучкості, простоті та аудитоздатності порівняно з комплексними протоколами. Він ідеально підходить для “одноразового” шифрування повідомлення для одержувача. Проте, HPKE сам по собі не забезпечує ані керування сеансами, ані прямої секретності , ані протоколів відновлення після компрометації, які є критичними для систем обміну повідомленнями.

Формулювання науково-технічної задачі

Таким чином, розробники сучасних веб-систем захищеного обміну повідомленнями опинилися перед обличчям технологічної та безпекової прогалини. Вони змушені обирати між:

- Використанням доведеного, але не масштабованого для груп протоколу (Signal).
- Спробою реалізації надскладного, але ефективного протоколу (MLS).
- Розробкою власного, нестандартного (і, ймовірно, небезпечного) криптографічного протоколу.

Виникає науково-технічна задача, яка полягає у вирішенні протиріччя між, з одного боку, високою потребою у доступних, крос-платформних, веб-орієнтованих системах обміну повідомленнями, здатних підтримувати ефективні та безпечні комунікації, та з іншого боку, відсутністю стандартизованих, гнучких та простих в аудиті архітектурних рішень. Ці рішення мають ефективно поєднувати перевірені протоколи, сучасні модульні примітиви та безпечно використовувати обмежені можливості клієнтського середовища для мінімізації довіри до сервера.

Відповідно, задача даної дипломної роботи полягає у проектуванні, розробці програмного прототипу та дослідженні архітектури веб-системи захищеного обміну повідомленнями, яка б:

- Базувалася на клієнт-серверній моделі з чітко визначеною моделлю загроз, що мінімізує довіру до сервера.

- Реалізовувала наскрізне шифрування для комунікацій виключно на стороні клієнта за допомогою стандартного Web Cryptography API.
- Адаптувала перевірені принципи генерації та керування ключами, їхнього захисту і передачі між пристроями
- Інтегрувала сучасний модульний стандарт HPKE для операцій асиметричного шифрування та інкапсуляції ключів, з метою спрощення реалізації, зниження ризику помилок імплементації та підвищення аудитоздатності системи.
- Поєднувала механізми симетричного шифрування з HPKE для досягнення ефективного виконання в веб-середовищі.

1.2 Об’єкт, предмет та мета дослідження

На основі аналізу проблеми, проведеного в підрозділі 1.1, який виявив суттєву науково-технічну прогалину між потребою у доступних веб-системах E2EE та відсутністю гнучких і масштабованих архітектур, ми формулюємо наступні ключові елементи дипломної роботи: об’єкт, предмет та мета дослідження.

Об’єкт дослідження

Об’єктом дослідження є процес захищеного обміну даними в реальному часі. Цей процес розглядається не ізольовано, а в контексті його реалізації в рамках сучасних клієнт-серверних веб-системах.

Об’єкт охоплює повний життєвий цикл захищеного повідомлення: від його створення, шифрування на клієнті відправника, транспортування через недовірену серверну інфраструктуру, яка виступає в ролі ретранслятора, до отримання та безпечного розшифрування на клієнті одержувача. Це включає також процеси встановлення початкової довіри, автентифікації, генерації, розподілу та ротації криптографічних ключів.

Предмет дослідження

Предметом дослідження є специфічні компоненти, що визначають безпеку та ефективність об’єкта дослідження. До предмету належить сукупність криптографічних методів, протоколів, архітектурних рішень та програмних інтерфейсів, що забезпечують проектування і реалізацію безпечної, ефективної, масштабованої веб-орієнтованої системи обміну повідомленнями з наскрізним шифруванням.

Предмет дослідження деталізується і включає наступні ключові компоненти:

1. Фундаментальні криптографічні механізми:
   - Асиметричне шифрування, дослідження систем шифрування з відкритим ключем як основи для встановлення ідентичності та початкового захищеного обміну даними. Це включає аналіз сучасних стандартів, зокрема HPKE, та його складових: механізму інкапсуляції ключів для ефективного встановлення спільного секрету.
   - Аналіз алгоритмів симетричного шифрування, зокрема режимів автентифікованого шифрування з додатковими даними AEAD, таких як AES-GCM. Вони є основою для швидкого та надійного шифрування безпосередньо повідомлень (даних у стані передачі) з використанням сеансових ключів.
   - Вивчення, протоколів обміну ключами, що дозволяють двом або більше сторонам встановити спільний таємний ключ через незахищений канал. Це включає аналіз протоколу Діффі-Хеллмана на еліптичних кривих ECDH та його використання у більш складних протоколах, як-от X3DH (Extended Triple Diffie-Hellman), для асинхронного встановлення сеансу.

- Криптографічні примітиви, такі як криптографічні геш-функції, наприклад SHA-256, для забезпечення цілісності даних та функцій виведення ключа KDF, як-от HKDF, для генерації безпечних криптографічних ключів із спільного секрету.

2. Протоколи захищених комунікацій та сеансів:
   - Керування сеансами E2EE, глибокий аналіз принципів роботи протоколів, що забезпечують розширені гарантії безпеки. В першу чергу, це стосується алгоритму Double Ratchet протоколу Signal, який забезпечує досконалу пряму секретність та безпеку після компрометації для комунікацій 1-на-1.
   - Протоколи автентифікації та авторизації, дослідження методів, за допомогою яких користувач підтверджує свою особу серверу (автентифікація, через пари логін/пароль або токени) та отримує права доступу до ресурсів. Важливою частиною є відокремлення автентифікації на сервері від наскрізної автентифікації між клієнтами.
3. Архітектура та системна реалізація:
   - Клієнт-серверна архітектура в контексті E2EE, аналіз архітектурних моделей, що мінімізують довіру до сервера. У цій моделі сервер виступає виключно як недовірений ретранслятор зашифрованих повідомлень та сховище для публічних криптографічних матеріалів користувачів.
   - Дослідження та порівняння підходів до реалізації E2EE, зокрема аналіз проблеми лінійної складності в підходах на базі Signal Protocol та вивчення альтернативних, більш масштабованих моделей
   - Програмні інтерфейси веб-браузера, вивчення можливостей, обмежень та аспектів безпеки Web Cryptography API як основного інструменту для виконання всіх криптографічних операцій безпосередньо на стороні клієнта.
4. Характеристики продуктивності та взаємодії:

- Масштабованість та продуктивність — оцінка того, як обрані криптографічні протоколи та архітектурні рішення впливають на продуктивність веб-додатку.
- Аспекти взаємодії “користувач-система” — дослідження точок перетину між користувацьким інтерфейсом та безпекою.

Мета дослідження

Метою даної дипломної роботи є вирішення науково-технічної задачі підвищення рівня безпеки, ефективності та масштабованості веб-орієнтованих систем обміну повідомленнями. Це досягається шляхом проектування архітектури та розробки програмного прототипу шифрування для захищеного обміну повідомленнями в реальному часі, що базується на поєднанні сучасних криптографічних стандартів та перевірених протоколів. Ціллю є створення архітектурного рішення, що забезпечує високий рівень конфіденційності, цілісності та автентичності даних при одночасному вирішенні проблем масштабування.

Крім того, мета виходить за рамки простої технічної реалізації. Вона охоплює ширші амбіції щодо стимуляції позитивних перетворень в системах обміну повідомленнями. Надаючи користувачам прозорість, інтуїтивно зрозумілі інтерфейси для управління даними та надійні механізми шифрування, кінцевою метою є формування довіри до цифровізації з подальшим розширенням цифрових екосистем для перенесення усіх можливих послуг на зручні онлайн-сервіси.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- Проаналізувати сучасні загрози безпеці для веб-систем обміну повідомленнями та провести глибоке дослідження існуючих протоколів наскрізного шифрування, ідентифікувавши їхні сильні сторони, недоліки та алгоритмічні основи.
- Обгрунтувати вибір конкретних криптографічних алгоритмів та примітивів, методів симетричного та асиметричного шифрування,

функцій гешування, механізмів інкапсуляції ключів, приділяючи особливу увагу їхній безпечній реалізації в середовищі веб-браузера з використанням Web Cryptography API.

- Спроектувати деталізовану клієнт-серверну архітектуру, що підтримує обмін повідомленнями в реальному часі. Архітектура має чітко визначати модель довіри, протоколи автентифікації користувачів, формати обміну ключами та механізми реалізації E2EE для комунікацій.
- Розробити програмний прототип шифрування, який втілює спроектовану архітектуру та реалізує ключові протоколи для захищених комунікацій. Прототип має бути створений з використанням сучасних веб-технологій, наприклад, JavaScript/TypeScript, WebSockets, WebCrypto API.
- Провести тестування розробленого програмного прототипу для верифікації його функціональної коректності та проведення оцінки рівня захищеності від базових векторів атак, визначених у моделі загроз.

1.3 Аналіз стану вирішення задачі

Для проведення грунтовного аналізу існуючих рішень та обгрунтування вибору архітектури для проектованої системи, необхідно спершу встанови “вимоги” або “критерії оцінки”. Цей підрозділ визначає фундаментальні принципи, на яких будується будь-яка захищена комунікаційна система, формулює модель загроз та описує сучасні властивості безпеки, які стали стандартом.

1.3.1 Фундаментальні вимоги, модель загроз та властивості безпеки

Будь-яка система, що працює з інформацією, традиційно оцінюється за трьома основними критеріями, відомими як “тріада CIA”: Конфіденційність, Цілісність та Доступність [18].

Конфіденційність (Confidentiality) — це властивість гарантує, що інформація не стане доступною або не буде розкритою неавторизованим особам, суб'єктам чи процесам. У контексті систем обміну повідомленнями, це означає, що зміст повідомлення має бути незрозумілим для будь-кого, окрім призначеного одержувача.

Цілісність (Integrity) — ця властивість гарантує точність і повноту даних та захищає їх від несанкціонованої модифікації або знищення. Для месенджера це означає, що одержувач має бути впевненим: повідомлення, яке він отримав, є саме тим повідомленням, яке відправив відправник, і воно не було змінене в дорозі.

Доступність (Availability) — це забезпечення своєчасного і надійного доступу до інформації та її використання авторизованими користувачами. Для месенджера це означає, що сервіс має бути здатним доставляти повідомлення та надавати доступ до історії листування.

Хоча всі три компоненти є важливими, у даній дипломній роботі фокус зміщено на Конфіденційність та Цілісність у ворожому середовищі, де доступність (тобто сама робота сервера) вважається гарантованою.

Модель загроз

Модель загроз визначає, від кого ми захищаємось. Для системи захищеного обміну повідомленнями ключовими є три типи супротивників:

- Пасивний супротивник — суб'єкт, який може лише прослуховувати мережевий трафік. Він може перехоплювати та записувати всі зашифровані повідомлення, але не може їх блокувати чи модифікувати.
- Активний супротивник (Man-in-the-Middle, MitM) — це більш потужний супротивник, який знаходиться між двома сторонами, що спілкуються. Він може не лише перехоплювати, але й активно втручатися в комунікацію: блокувати, видаляти, змінювати або

впроваджувати власні повідомлення. Класичним прикладом є атака з підміною публічних ключів під час їх початкового обміну [19, 20].

Рисунок 1.1 — Схема моделі загрози Man-in-the-Middle

- Скомпрометований сервер — це центральна загроза в моделі наскрізного шифрування. Ми виходимо з припущення, що супротивник має повний контроль над серверною інфраструктурою. Це може статися внаслідок хакерської атаки, інсайдерської загрози або через юридичний примус (наприклад, отримання ордеру на доступ до серверів). У цій моделі сервер є недовіреним і вважається частиною “ворожої” мережі.

Для захисту від цих загроз використовуються дві принципово різні моделі шифрування: TLS проти E2EE [21].

Транспортне шифрування (TLS/SSL)

Модель, яку використовує переважна більшість веб-сайтів (HTTPS) та багато “хмарних” рішень, захищає лише канал зв’язку між клієнтом та сервером. Повідомлення шифрується на пристрої відправника, надходить на сервер, де воно розшифровується і зберігається у відкритому вигляді. Потім, коли одержувач

підключається, сервер знову шифрує повідомлення і відправляє його через інший захищений TLS-канал.

Рисунок 1.2 — Схема шифрування в TLS

Ця модель повністю беззахисна перед скомпрометованим сервером. Сервер має повний доступ до відкритого тексту всіх повідомлень, що нівелює конфіденційність [22].

Наскрізне шифрування (E2EE)

Модель є відповіддю на загрозу скомпрометованого сервера. В E2EE дані шифруються на пристрої відправника (перший “кінець”) і можуть бути розшифровані лише на пристрої одержувача (другий “кінець”). Криптографічні ключі, необхідні для розшифрування, зберігаються виключно на кінцевих пристроях і ніколи не передаються серверу. Для сервера повідомлення виглядає як незрозумілий набір зашифрованих байтів.

Рисунок 1.3 — Схема наскрізного шифрування

E2EE є єдиною моделлю, що забезпечує справжню конфіденційність та цілісність даних, оскільки вона виключає сервер із “кола довіри” [22].

Розширені (сучасні) властивості безпеки

Простого шифрування даних недостатньо. Сучасні протоколи E2EE мають надавати значно сильніші гарантії безпеки, які захищають не лише поточні, але й минулі та майбутні комунікації.

Досконала пряма секретність (Perfect Forward Secrecy, PFS) — це властивість криптографічної системи, яка гарантує, що компрометація довгострокових приватних ключів не дозволить супротивнику розшифрувати минулі повідомлення. Уявіть, що супротивник роками пасивно записував ваш зашифрований трафік, а потім зміг викрасти приватний ключ вашого акаунту. У системі без PFS він зможе розшифрувати весь записаний архів. Система з PFS, навпаки, генерує унікальні, ефемерні (тимчасові) сеансові ключі для кожної розмови або навіть кожного повідомлення. Ці сеансові ключі неможливо відновити, знаючи лише довгострокові ключі, тому минулі повідомлення залишаються в безпеці [23].

Безпека після компрометації (Post-Compromise Security, PCS) — ця властивість є дзеркальним відображенням PFS і іноді називається "майбутньою секретністю" (Future Secrecy) [24]. PCS гарантує, що якщо супротивник миттєво

скомпрометував поточний стан вашого сеансу (наприклад, викрав ефемерний сеансовий ключ з пам’яті), протокол зможе "самовідновитися". Як тільки протокол оновлює сеансовий ключ (наприклад, при відправці наступного повідомлення), супротивник втрачає здатність розшифровувати майбутні повідомлення. Це критично важлива властивість для захисту від атак, що передбачають тимчасовий доступ до пристрою.

Можливість заперечення (Deniability) — це властивість, яка дозволяє відправнику повідомлення правдоподібно заперечувати своє авторство. У системі з можливістю заперечення, одержувач може бути впевнений в автентичності отриманого повідомлення, але він не може криптографічно довести третій стороні (наприклад, суду), що саме відправник його створив. Це досягається тим, що протокол дає можливість самому одержувачу підробити повідомлення так, ніби воно прийшло від відправника [25]. Це контрастує з системами на базі PGP, де цифровий підпис є незаперечним доказом авторства.

Ці вимоги — E2EE, PFS, PCS та Deniability — формують “золотий стандарт”, за яким ми будемо оцінювати всі існуючі протоколи в наступних підрозділах.

1.3.2 Еволюція протоколів E2EE

Історія сучасних протоколів захищеного обміну повідомленнями — це історія вирішення фундаментальних недоліків попередніх систем, зокрема, усунення фатальної вразливості, пов’язаної з компрометацією довгострокових ключів.

Pretty Good Privacy (PGP)

Першою широко розповсюдженою та криптографічно надійною системою E2EE був протокол Pretty Good Privacy (PGP) (та його відкрита реалізація

OpenPGP). Розроблений у 1990-х роках для захисту електронної пошти, PGP є класичним прикладом асинхронної криптографічної системи [26].

Його модель роботи базується на довгострокових парах ключів (публічному та приватному) [27]. Коли відправник хоче надіслати конфіденційне повідомлення, він:

1. Генерує випадковий, одноразовий сеансовий ключ.
2. Шифрує тіло повідомлення цим сеансовим ключем за допомогою симетричного шифру, наприклад AES.
3. Шифрує сам сеансовий ключ, використовуючи довгостроковий публічний ключ одержувача.
4. Надсилає зашифрований сеансовий ключ разом із зашифрованим повідомленням.
5. Одержувач, отримавши лист (можливо, через дні або тижні), використовує свій довгостроковий приватний ключ для розшифрування сеансового ключа, яким у свою чергу, розшифровує повідомлення.

Ця модель забезпечує надійну конфіденційність та автентичність (через цифрові підписи), але має два фундаментальні недоліки, що роблять її непридатною для сучасних месенджерів:

- Довгостроковий приватний ключ одержувача є “головним ключем” (master key), який захищає всі сеансові ключі всіх отриманих ним повідомлень. Якщо цей приватний ключ буде скомпрометований (наприклад, викрадений з комп’ютера), супротивник, який роками пасивно записував зашифровані листи, зможе повернутися назад і розшифрувати весь архів минулих комунікацій.
- Використання цифрових підписів для автентифікації створює незаперечний криптографічний доказ того, що конкретний відправник є автором конкретного повідомлення. Це є прямою протилежністю до властивості заперечення, яка є бажаною для приватних розмов.

Off-the-Record (OTR) та революція у реальному часі

У 2004 році Нікіта Борисов, Ян Аврум Голдберг та Ерік Брюер представили протокол Off-the-Record [28]. OTR був розроблений для синхронних (real-time) чатів (Instant Messaging) і став революційним, оскільки вперше впровадив у практичне використання дві найважливіші властивості, відсутні в PGP [29]:

- OTR не покладається на довгострокові ключі для шифрування. Натомість, на початку кожного сеансу він використовує протокол обміну ключами Діффі-Хеллмана для генерації нових, ефемерних (тимчасових) сеансових ключів. Більше того, він постійно "прокручує" (ratchets) ці ключі вперед під час розмови. Таким чином, компрометація будь-якого ключа (навіть довгострокового ключа ідентифікації) не дозволяє розшифрувати жодної минулої розмови.
- OTR відмовився від цифрових підписів. Для автентифікації повідомлень він використовує коди автентифікації повідомлень, які генеруються на основі спільного секрету, відомого обом сторонам [30]. Оскільки і відправник, і одержувач можуть згенерувати однаковий MAC, одержувач не може довести третій стороні, що повідомлення створив саме відправник (оскільки він міг створити його сам).

Однак, OTR мав такий самий фундаментальний недолік, як і PGP, але дзеркальний: він був розроблений для синхронних комунікацій. Протокол вимагав, щоб обидва учасники були одночасно онлайн для встановлення сеансу та обміну ключами [31]. Це робило його абсолютно непридатним для сучасного світу, де користувачі постійно перебувають офлайн, а повідомлення мають доставлятися асинхронно.

Silent Circle Instant Messaging Protocol (SCIMP)

Розуміючи обмеженість OTR, розробники почали шукати шляхи для адаптації його властивостей до асинхронного середовища. Однією з таких спроб

був Silent Circle Instant Messaging Protocol (SCIMP) [32]. SCIMP, який багато в чому базувався на OTR та ZRTP (протоколі для VoIP), намагався вирішити проблему асинхронності. Він використовував інший стиль “храповика” — геш-храповик (hash ratchet), де кожний наступний ключ повідомлення був гешем попереднього, що забезпечувало чудову пряму секретність [33].

Проте, як зазначалося в аналізі, що призвів до створення протоколу Signal, підхід SCIMP забезпечував добру пряму секретність, але мав погану безпеку після компрометації. Компрометація поточного ключа сеансу могла призвести до компрометації всіх майбутніх повідомлень у цьому сеансі.

Порівняльний аналіз ранніх протоколів

Для наочного узагальнення еволюції та ключових недоліків, що призвели до необхідності створення нових рішень, наведемо порівняльну таблицю (Таблиця 1).

Таблиця 1.1 — Порівняльний аналіз властивостей безпеки протоколів PGP, OTR та SCIMP

Властивість	**PGP**	**OTR**	**SCIMP**
Основне призначення	Асинхронна e-mail	Синхронний IM чат	Асинхронний IM чат
E2EE	Так	Так	Так
PFS	Ні	Так	Так (через геш-ratchet)
PCS	Ні	Так (через DH-ratchet)	Ні
Deniability	Ні (використовує підписи)	Так (використовує MAC)	Так
Підтримка асинхронності	Так (створений для неї)	Ні (вимагає онлайн-статусу)	Так

Цей аналіз та порівняльна таблиця чітко демонструють технологічну прогалину станом на початок 2010-х років:

- PGP був безпечним для асинхронних даних, але не мав ані PFS, ані Deniability.
- OTR забезпечив PFS та Deniability, але працював лише в синхронному режимі 1-на-1.
- SCIMP спробував вирішити асинхронність, але пожертвував безпекою після компрометації.

Світу був потрібен протокол, який би поєднав у собі найкраще з усіх світів: досконалу пряму секретність та безпеку після компрометації, можливість заперечення, а також був би розроблений з нуля для асинхронного середовища. Цим протоколом і став Signal.

1.3.3 “Золотий стандарт” для 1-на-1

Аналіз PGP, OTR та SCIMP виявив чітку технологічну прогалину: відсутність протоколу, який би одночасно забезпечував досконалу пряму секретність, безпеку після компрометації та можливість заперечення, і при цьому був би розроблений для асинхронного середовища (де користувачі не обов’язково одночасно перебувають онлайн).

Цю прогалину заповнив протокол Signal, розроблений Тревором Перріном та Моксі Марлінспайком. Це не єдиний монолітний протокол, а комбінація двох окремих механізмів, кожен з яких вирішує свою задачу:

- X3DH (Extended Triple Diffie-Hellman) — протокол для початкового асинхронного встановлення сеансового ключа [34].
- Double Ratchet (DR) — алгоритм для подальшого керування сеансом та обміну повідомленнями після того, як сеанс встановлено [35].

Протокол OTR вимагав, щоб обидва користувачі були онлайн. X3DH вирішує цю проблему, дозволяючи серверу виступати в ролі “сховища” публічних ключів, при цьому сервер залишається недовіреним [36].

Після того, як X3DH встановив початковий спільний секрет, у гру вступає алгоритм Double Ratchet для керування подальшим обміном повідомленнями. Його мета — забезпечити найсильніші гарантії PFS та PCS.

Назва “подвійний” (Double) походить від того, що він комбінує два “храповики” (ratchet — механізм, що рухається лише в одному напрямку), які походять від OTR та SCIMP:

- Symmetric-key Ratchet (Геш-храповик) — це, по суті, ідея з SCIMP. Коли Аліса надсилає повідомлення, вона використовує KDF (функцію виведення ключа) для отримання нового ключа повідомлення зі свого “ланцюжка відправки”. Наступне повідомлення вона зашифрує ключем, отриманим з наступного кроку KDF. Те саме робить Боб зі своїм “ланцюжком отримання”. Цей механізм забезпечує досконалу пряму секретність для кожного окремого повідомлення.
- Diffie-Hellman Ratchet (Асиметричний храповик) — це, по суті, ідея з OTR. Кожного разу, коли користувач не просто надсилає повідомлення, а й отримує відповідь, до його повідомлення прикріплюється новий ефемерний публічний ключ DH. Коли інша сторона отримує цей ключ, обидва учасники виконують новий обмін. Результат цього обміну робить абсолютно нові симетричні ланцюжки.

Поєднання цих двох механізмів дає властивість самовідновлення або безпеку після компрометації, якої бракувало SCIMP.

Нехай зловмисник отримує повний доступ до пам’яті пристрою Аліси та викрадає її поточні симетричні ключі (з геш-храповика). З Double Ratchet, зловмисник може читати наступні повідомлення Аліси. Але щойно Боб відповість, він прикріпить до свого повідомлення новий публічний ключ DH. Аліса та Боб виконають новий обмін DH, який створить абсолютно новий спільний секрет, невідомий зловмиснику. З цього моменту всі подальші симетричні ключі будуть новими, і зловмисник буде “викинутий” з сеансу.

Комбінація X3DH та Double Ratchet стала “золотим стандартом” і була прийнята у WhatsApp, Google Messages та багатьох інших. Вона вирішила практично всі проблеми для комунікацій 1-на-1.

Однак, фундаментальний недолік цього підходу виявляється при спробі масштабувати його на групові чати [37]. Класична реалізація груп у Signal (та його похідних) базується на “парних” (pairwise) сеансах. Коли користувач хоче відправити повідомлення у групу з n учасників, він не надсилає одне повідомлення; він фактично:

- Підтримує n−1 окремих сеансів Double Ratchet з кожним членом групи.
- Шифрує повідомлення n−1 разів, окремо для кожного учасника.
- Відправляє n−1 окремих повідомлень.

Хоча цей підхід надійний, він має лінійну складність O(n). Для невеликих груп це прийнятно, але для великих корпоративних чи публічних груп це стає вкрай неефективним. Операції, що змінюють склад групи (додавання або видалення учасника), вимагають розсилки оновлень кожному з n учасників окремо, що створює величезне обчислювальне та мережеве навантаження.

Саме ця проблема — неефективність та неможливість масштабування протоколу Signal для великих груп — стала головним мотиватором для розробки наступного покоління протоколів, таких як MLS.

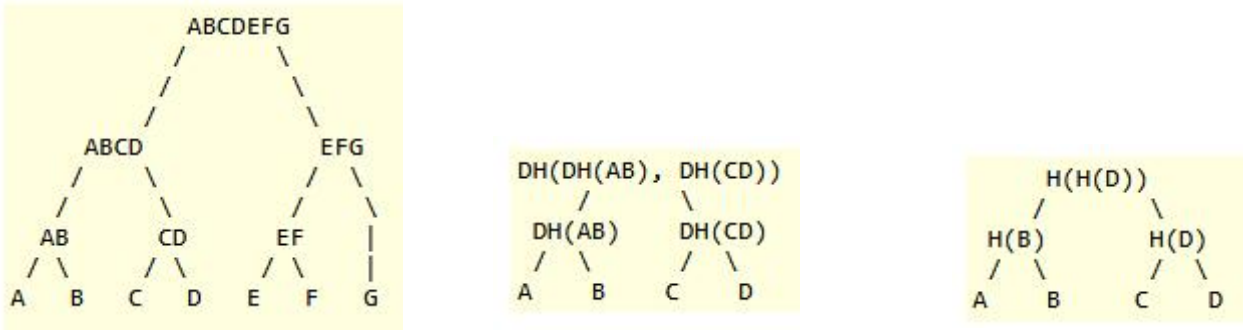
1.3.4 Аналіз стандарту IETF для груп

Відповіддю на проблему масштабування став протокол Messaging Layer Security (MLS), який розроблявся робочою групою в IETF (Internet Engineering Task Force) та був офіційно стандартизований у липні 2023 року як RFC 9420. У його розробці брали участь провідні компанії та дослідники з Google, Mozilla, Cisco, Wire, Оксфордського університету та INRIA [38].

MLS з самого початку проектувався з метою забезпечити ті ж самі високі гарантії безпеки, що й Signal, але з ефективністю для груп будь-якого розміру, аж до 50,000 учасників і більше [39].

Асинхронні дерева з храповиком (ART)

Фундаментальна відмінність MLS від підходу Signal полягає у структурі даних, що використовується для управління ключами. Замість підтримки n−1 окремих парних сеансів, MLS організовує всіх n учасників групи як листки у бінарному дереві. Ця структура називається “Асинхронне дерево з храповиком” (Asynchronous Ratcheting Tree, ART), або в поточній специфікації, TreeKEM [40].



а) б) в)

Рисунок 1.4 — Структури дерев у протоколі MLS: а) Логічна структура дерева; б) ART; в) TreeKEM

Логарифмічна складність O(logn)

Перевага такої деревоподібної структури стає очевидною при зміні складу групи (commit-операції):

1. Учасник використовує груповий секрет (з кореня) для шифрування повідомлень. Ця операція має складність O(1).
2. Щоб видалити D, один з учасників надсилає “update” транзакцію. Йому не потрібно надсилати n−1 повідомлень. Замість цього він генерує нові ключі лише для тих вузлів, що лежали на шляху D до кореня. Потім він шифрує ці нові ключі для “спів-шляху” (copath) — тобто для вузлів, які були “сусідами” цього шляху.

3. В результаті, інші учасники (A, B, C) отримують одне компактне повідомлення про оновлення. Вони використовують свої старі ключі, щоб розшифрувати потрібну їм частину оновлення, і обчислюють новий кореневий ключ. Видалений учасник D, не маючи нових ключів, більше не може обчислити новий кореневий секрет.

Найголовніше, кількість обчислень та розмір повідомлення про оновлення залежать не від кількості учасників n, а від висоти дерева, яка є O(logn). Якщо розрахувати кількість операцій для групи з 100000 учасників, то для підходу Signal (O(n)) оновлення ключа вимагає ≈100000 окремих операцій та повідомлень, а для MLS (O(logn)) $\log_2(100000) \approx 17$ операцій.

Недоліки

MLS — це безперечно майбутнє захищених групових комунікацій, і такі компанії, як Google та Apple, вже оголосили про його впровадження. Однак він має один суттєвий недолік, який є критичним для нашого дослідження, а саме, надзвичайна складність.

Специфікація RFC 9420 є об'ємною та складною для розуміння. Керування станом дерева, обробка транзакцій (commits), забезпечення узгодженості стану між тисячами асинхронних клієнтів — все це є монументальною інженерною задачею, з чого виходить високий поріг реалізації. Хоча складність є логарифмічною, емпіричні дослідження показують, що реальні обчислювальні витрати на обробку оновлень є значними. Створення безпечної, коректної та сумісної реалізації MLS (як, наприклад, OpenMLS на Rust) [41] є завданням для цілої команди інженерів-криптографів, а не для однієї дипломної роботи.

Хоча MLS є найкращим рішенням для дуже великих груп, він може бути надлишковим та надто складним для малих і середніх груп (до 100-200 учасників), де простіша, хоч і менш ефективна, архітектура могла б бути достатньою та набагато простішою в реалізації.

1.3.5 Огляд криптографічних примітивів та стандартів для вебу

Складність монолітних протоколів, таких як Signal або MLS, значною мірою полягає в тому, що вони тісно пов’язують логіку керування сеансами з криптографічними операціями. Сучасний підхід, навпаки, полягає у виділенні надійних криптографічних “примітивів” у окремі стандарти.

Hybrid Public Key Encryption (RFC 9180)

Традиційне шифрування з відкритим ключем (наприклад, пряме шифрування за допомогою RSA) є повільним та має відомі недоліки. Тому на практиці завжди використовується гібридний підхід: генерується випадковий симетричний ключ, ним шифрується повідомлення, а потім цей симетричний ключ шифрується або “огортається” публічним ключем одержувача.

Проблема полягала в тому, що існувало безліч способів зробити це “огортання”, і багато з них були небезпечними. HPKE, стандартизований як RFC 9180, вирішує цю проблему, пропонуючи єдиний, безпечний та гнучкий стандарт для виконання цієї операції [42].

HPKE — це не протокол сеансу, а “конструктор”, що поєднує три компоненти [43]:

- KEM (Key Encapsulation Mechanism) — механізм інкапсуляції ключа. Замість того, щоб відправник створював ключ і шифрував його, KEM виконує асиметричну операцію для генерації спільного секрету. Цей підхід є більш надійним.
- KDF (Key Derivation Function) — функція виведення ключа (наприклад HKDF). Вона бере спільний секрет з KEM і безпечно “розтягує” його для отримання одного або декількох симетричних ключів шифрування.
- AEAD (Authenticated Encryption with Associated Data) — автентифіковане шифрування. Це симетричний шифр (наприклад AES-

GCM або ChaCha20-Poly1305), який одночасно шифрує дані та забезпечує їх цілісність (захист від модифікації).

Основні переваги — це простота та гнучкість. HPKE значно простіший за повний протокол Signal. Він виконує одне чітке завдання — “безпечно надіслати зашифрований блок даних одержувачу, маючи його публічний ключ”. Це легко реалізувати та перевірити. Також, стандарт дозволяє легко замінювати компоненти (наприклад KEM або AEAD) без зміни загальної логіки.

Найважливіший недолік HPKE полягає в тому, що він сам по собі не є сеансовим протоколом. Він не надає властивостей досконалої прямої секретності або безпеки після компрометації з коробки. Якщо просто використовувати HPKE з довгостроковим публічним ключем, ми отримаємо ту ж проблему, що й у PGP.

Таким чином, HPKE є ідеальним примітивом, наприклад, для шифрування повідомлень, що ініціюють сеанс, або для передачі нових ключів у групі, але він має бути доповнений сеансовою логікою, такою як Double Ratchet, щоб отримати PFS та PCS.

Web Cryptography API (WebCrypto)

Для того, щоб веб-додаток міг реалізувати E2EE, всі криптографічні операції (шифрування, генерація ключів, підпис) мають виконуватися на стороні клієнта, тобто безпосередньо у браузері. Спроби реалізувати це за допомогою чистого JavaScript історично були провальними через проблеми з безпекою та продуктивністю.

Web Cryptography API — це низькорівневий інтерфейс (API), вбудований у сучасні браузери, який надає веб-сторінкам доступ до криптографічних примітивів, реалізованих нативно. Доступ до нього здійснюється через об’єкт window.crypto.subtle [44].

Найважливіша функція — це можливість генерувати або імпортувати CryptoKey об’єкти з прапорцем extractable: false [45]. Це означає, що JavaScript-код може використовувати цей ключ для операцій, але він ніколи не може

прочитати сам приватний ключ. Ключ залишається в захищеній області пам’яті браузера.

WebCrypto вирішує проблему виконання криптографії, але не вирішує фундаментальну проблему веб-середовища — проблему довіри до коду. Як було зазначено у підрозділі 1.1, скомпрометований сервер може надіслати зловмисний JavaScript. Навіть якщо приватний ключ має прапор extractable: false, цей зловмисний JS може просто перехопити відкритий текст повідомлення перед тим, як воно буде передано на шифрування, або після того, як воно буде отримане з функції розшифрування. Хоча CryptoKey можна зберігати в IndexedDB, саме сховище не є захищеним від зловмисного JS того ж походження (origin).

Навіть врахувавши недоліки, можна зробити висновок, що у нас є потужні “будівельні блоки”. WebCrypto дає нам можливість виконувати криптографію у браузері, а HPKE дає нам простий та надійний стандарт для асиметричного шифрування.

1.3.6 Архітектурний аналіз існуючих рішень

Аналіз теоретичних протоколів є неповним без вивчення їх практичних реалізацій. Саме в комерційних та open-source продуктах виявляються ключові архітектурні компроміси.

Таблиця 1.2 — Порівняльний аналіз архітектур існуючих месенджерів

Додаток	**E2EE за замовчуванням**	**Протокол E2EE**	**Ключова архітектурна особливість / Недолік**
Signal	Так	Signal Protocol (Відкритий)	Еталон. Повний E2EE. Сервер - недовірений. Недолік: маштабування.
WhatsApp	Так	Signal Protocol (Fork)	Найбільше впровадження E2EE. Вразливість: опціональні нешифровані бекапи.
Telegram	Ні	MTProto (Власний)	Гібридна модель. E2EE лише в “Секретних чатах” (1-на-1).

Продовження таблиці 2

iMessage	Так (Apple-to-Apple)	Apple (Власний, Закритий)	“Закрита екосистема”. Вразливість: iCloud Backup (ключі E2EE зберігаються у Apple).
Viber	Так	Власний (Закритий)	E2EE за замовчуванням, але протокол є “чорною скринькою”, що ускладнює незалежний аудит.
Rocket.Chat	Ні (Опціонально)	Власний (на базі Signal)	Open-source / Веб-орієнтований. E2EE не за замовчуванням.

Signal є стандартом та еталонною реалізацією E2EE. Його архітектура побудована на базі централізованого сервера, який виступає виключно як недовірений ретранслятор повідомлень та сховище “пакетів” ключів (Pre-Key Bundles) для асинхронної комунікації. Будучи розробником Signal Protocol , він застосовує E2EE за замовчуванням для абсолютно всіх комунікацій. Signal надає найвищий рівень безпеки (PFS, PCS), але його архітектура та проблеми масштабованості безпосередньо впливають на користувальницький досвід.

WhatsApp, що належить Meta, є найпопулярнішим месенджером у світі, який використовує E2EE. Його архітектура подібна до Signal, і він використовує власну реалізацію (fork) Signal Protocol [46]. E2EE застосовується за замовчуванням для всіх комунікацій. Однак WhatsApp часто критикують за резервні копії: хоча зараз вони можуть бути захищені E2EE, ця опція є відносно новою та не завжди ввімкненою користувачами, що створює вразливість.

Telegram вирізняється своєю гібридною та часто неправильно зрозумілою моделлю безпеки. Він використовує власний протокол MTProto [47]. Найважливіше те, що E2EE не використовується за замовчуванням. Архітектура розділена на два типи чатів: “хмарні” та “секретні”. “Хмарні чати” (всі звичайні чати 1-на-1 та всі групові чати) використовують лише транспортне шифрування, і сервери Telegram мають повний доступ до відкритого тексту. E2EE застосовується лише в опціональних “секретних чатах” [48], які доступні лише

для 1-на-1. Це є яскравим прикладом архітектурного компромісу на користь зручності (миттєва синхронізація історії) ціною безпеки.

iMessage від Apple є прикладом E2EE, глибоко інтегрованого у власну “закриту екосистему” [49]. Він використовує протокол E2EE за замовчуванням для всіх повідомлень між пристроями Apple. Однак його головною архітектурною вразливістю є резервні копії iCloud. Якщо користувач вмикає стандартне резервне копіювання (що є типовою поведінкою), копія приватних ключів, необхідних для розшифрування повідомлень, завантажується на сервери Apple. Це повністю нівелює E2EE, оскільки Apple (або будь-хто з ордером) може отримати доступ до всіх повідомлень.

Viber — ще один популярний месенджер, який декларує повне E2EE за замовчуванням для всіх чатів. Проте, на відміну від Signal, він використовує власний, closed-source протокол [50]. Хоча компанія стверджує, що він базується на тих самих принципах, що й Signal, закритість коду робить неможливим повноцінний незалежний аудит, що вважається у криптографічній спільноті гіршою практикою порівняно з відкритими стандартами.

Rocket.Chat представляє сегмент відкритих (open-source) корпоративних платформ, які можна розгортати на власній інфраструктурі, що робить його популярною альтернативою для багатьох компаній. Ця платформа веб-орієнтована. Її E2EE є опціональним. Для його реалізації використовується власна імплементація, що значною мірою натхненна Signal Protocol, але з урахуванням компромісу зручності використання [51].

1.4 Обгрунтування цілей дослідження

Проведений у підрозділі 1.3. грунтовний аналіз стану вирішення задачі виявив фундаментальний архітектурний конфлікт, що є центральною проблемою всієї галузі захищених комунікацій. Цей конфлікт між зручністю використання та криптографічною надійністю є головним обгрунтуванням необхідності даного дипломного дослідження.

Аналіз практичних реалізацій чітко окреслив два протилежні полюси. З одного боку, модель максимальної зручності Telegram, яка ідеально використовує переваги веб-середовища. Користувач отримує миттєву синхронізацію всієї історії повідомлень між необмеженою кількістю пристроїв. Ціною такої зручності є повна відмова від E2EE за замовчуванням, оскільки сервер має повний доступ до відкритого тексту.

З іншого боку, модель максимальної безпеки Signal пропонує стандарт криптографічного захисту — Signal Protocol, що гарантує досконалу пряму секретність та безпеку після компрометації. Ціною такої безпеки є зручність, особливо у вебі. Оскільки сервер є недовіреним, такі базові функції, як синхронізація історії або підключення нового веб-клієнта, перетворюються на складну, нетривіальну задачу.

Зміна парадигми веб-розробки та зростання вимог до конфіденційності користувацьких даних диктують необхідність переходу від server-side encryption до архітектур із “нульовим розголошенням” (Zero-Knowledge Architecture), де сервер виконує виключно роль транспорту та сховища зашифрованих даних, як “сліпий” ретранслятор.

Для додатків масового застосування, ця проблема стоїть найгостріше. Користувачі очікують від додатку “увійшов і працюєш”. Існуючі E2EE-моделі можуть прямо суперечити цьому очікуванню. Це створює чітку технологічну прогалину: відсутність стандартизованої, аудитоздатної архітектури для систем, яка б змогла запропонувати гібридний підхід — надійну безпеку, що не знищує зручність.

Дана дипломна робота спрямована на заповнення саме цієї прогалини. Замість того, щоб обирати одну з крайнощів, ціллю дослідження є проектування та розробка програмного прототипу з гібридною архітектурою E2EE, що пропонує користувачеві компроміс безпеки і зрочномті.

Вибір підходу до реалізації системи захищеного обміну повідомленнями базується на принципах гібридного шифрування та моделі Envelope Encryption, що дозволяє поєднати ефективність симетричних алгоритмів з безпекою

розподілу ключів асиметричних систем. Такий вибір обгрунтовується необхідністю забезпечення балансу між криптографічною стійкістю, продуктивністю у веб-середовищі та масштабованістю.

Обгрунтування вибраного підходу базується на синергії проаналізованих примітиві, стандартів та протоколів і поєднує наступні ключові компоненти:

1) Схема захищеного транспортування повідомлень — процес обміну даними реалізується через дворівневу схему шифрування, що дозволяє оптимізувати продуктивність та забезпечити криптографічну стійкість:
   - Рівень даних (Payload Layer) — безпосереднє шифрування тіла повідомлення (текст, медіа) здійснюється унікальним випадковим симетричним ключем сесії з використанням алгоритму автентифікованого шифрування, наприклад AES-GCM. Це забезпечує конфіденційність та цілісність даних, а також генерує унікальний вектор ініціалізації (IV) для кожного повідомлення.
   - Рівень транспорту ключа (Key Transport Layer), де симетричний ключ використаний на попередньому етапі, шифрується (інкапсулюється) для публічного ключа отримувача з використанням стандарту HPKE. Такий підхід дозволяє зберігати у базі даних структуру, що містить: зашифрований контент, вектор ініціалізації та інкапсульований ключ HPKE Ciphertext, роблячи неможливим розшифрування даних сервером або третіми особами.
2) Портативна система управління ключами (Encrypted Key Storage) для забезпечення мобільності користувача, можливість доступу з різних пристроїв, без компрометації безпеки, застосовується механізм віддаленого зберігання зашифрованих приватних ключів. Генерація пари асиметричних ключів відбувається виключно на стороні клієнта, в браузері. Приватний ключ перед відправкою на сервер шифрується симетричним ключем, що отриманий з пароля користувача за допомогою алгоритму деривації ключа Password-Based Key Derivation Function 2 з високою кількістю ітерацій. Сервер зберігає лише

зашифрований контейнер, не маючи технічної можливості отримати доступ до оригінального приватного ключа. Розшифрування відбувається виключно в оперативній пам’яті пристрою користувача після успішної автентифікації.

3) Використання Web Cryptography API, де реалізація всіх криптографічних примітивів, генерація ключів, гешування, симетричне та асиметричне шифрування, базується на стандарті W3C Web Crypto. Це дозволяє використовувати нативні механізми браузера, що забезпечують вищу швидкість виконання операцій та кращий захист від атак порівняно з JavaScript-реалізаціями

Наукова новизна дослідження полягає у проектуванні архітектури, яка адаптує механізм симетричного шифрування з використанням стандарту HPKE для середовища веб-браузера, забезпечуючи при цьому безпечну синхронізацію ідентичності користувача між пристроями через недовірений сервер.

Практична цінність роботи полягає у розробці прототипу шифрування, що реалізує повний цикл захищеного обміну даними: від генерації та захисту ключів паролем до реалізації протоколу обміну, де HPKE виступає безпечним транспортом для симетричних ключів шифрування даних.

Таким чином, метою роботи є створення та дослідження захищеної архітектури обміну повідомленнями у веб-середовищі, що базується на комбінації симетричного шифрування даних та асиметричної інкапсуляції ключів.

Планується досягти конкретних наукових та прикладних результатів:

- Розроблений механізм безпечного зберігання та відновлення приватних ключів користувача з використанням PBKDF2.
- Спроектувану структуру даних для зберігання зашифрованих повідомлень.
- Реалізуваний програмний модуль для шифрування повідомлень та інкапсуляції ключів за допомогою HPKE.
- Результати тестування прототипу на предмет коректності відновлення даних та стійкості до перехоплення.

2 МЕТОДОЛОГІЯ ТА ЗАСОБИ ПРОЕКТУВАННЯ ЗАХИЩЕНОЇ ВЕБ-СИСТЕМИ

2.1 Аналіз архітектурних підходів до реалізації E2EE у веб-додатках

Для розв’язання науково-технічної задачі, сформульованої у першому розділі, яка полягає у подоланні конфлікту між безпекою наскрізного шифрування та зручністю використання у веб-середовищі, необхідно спершу провести декомпозицію та аналіз фундаментальних архітектурних підходів, що використовуються для побудови таких систем.

Кожен із цих підходів “методів” пропонує власний спосіб розв’язання тріади завдань: встановлення сеансу, шифрування повідомлень та керування криптографічними ключами. Вибір конкретного методу безпосередньо впливає на властивості безпеки системи, її масштабованість та кінцеву зручність для користувача.

Модель “Довіреного сервера” на базі TLS

Найпоширенішою архітектурою в сучасних веб-додатках є модель, що покладається виключно на шифрування транспортного рівня. Цей підхід забезпечує захищений канал між клієнтом (браузером) та сервером, запобігаючи прослуховуванню мережевого трафіку пасивним супротивником.

Повідомлення шифрується на пристрої відправника, передається на сервер, де розшифровується і зберігається у відкритому вигляді. Коли одержувач підключається, сервер знову шифрує повідомлення та відправляє його через інший TLS-канал.

Яскравим прикладом такої архітектури є чати месенджера Telegram. Ця модель забезпечує максимальну зручність: миттєва синхронізація історії, доступ з будь-якого пристрою, відсутність потреби у складному керуванні ключами.

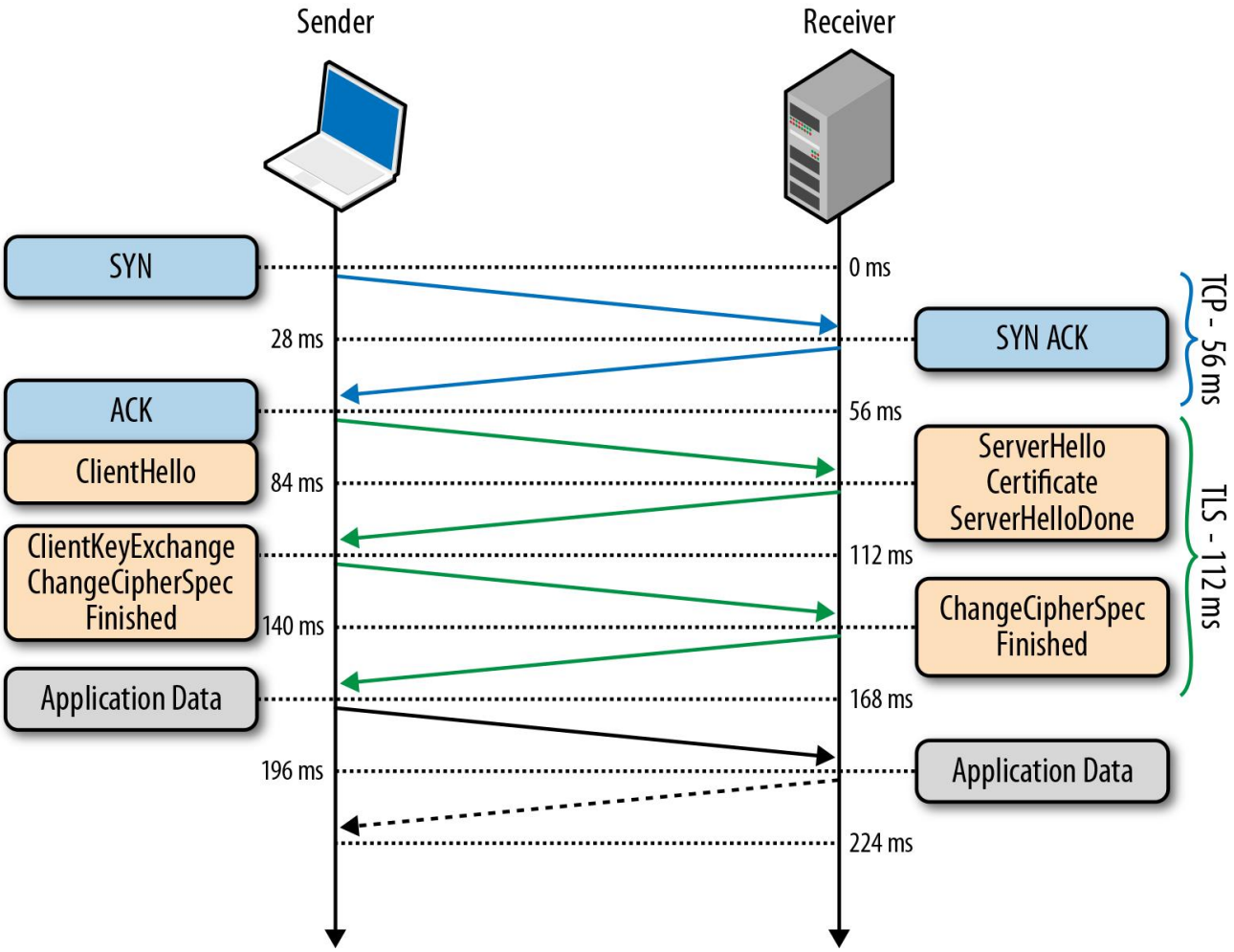


Рисунок 2.1 — TLS-рукостискання

Цей підхід фундаментально не є наскрізним шифруванням. Він є абсолютно беззахисним перед загрозою скомпрометованого сервера. Оскільки сервер має повний доступ до відкритого тексту всіх повідомлень, він стає єдиною точкою відмови, що нівелює конфіденційність у разі хакерської атаки, інсайдерської загрози або юридичного примусу.

“Класична” модель E2EE на базі PGP

Історично першою життєздатною моделлю E2EE став протокол Pretty Good Privacy. Цей підхід є асинхронним за своєю природою і базується на використанні довгострокових пар ключів (приватного та публічного).

Для відправки повідомлення генерується одноразовий симетричний ключ, яким шифрується саме повідомлення. Після цього, цей симетричний ключ шифрується або “огортається” довгостроковим публічним ключем одержувача. Одержувач використовує свій довгостроковий приватний ключ для розшифрування симетричного ключа, яким, у свою чергу, розшифровує повідомлення.

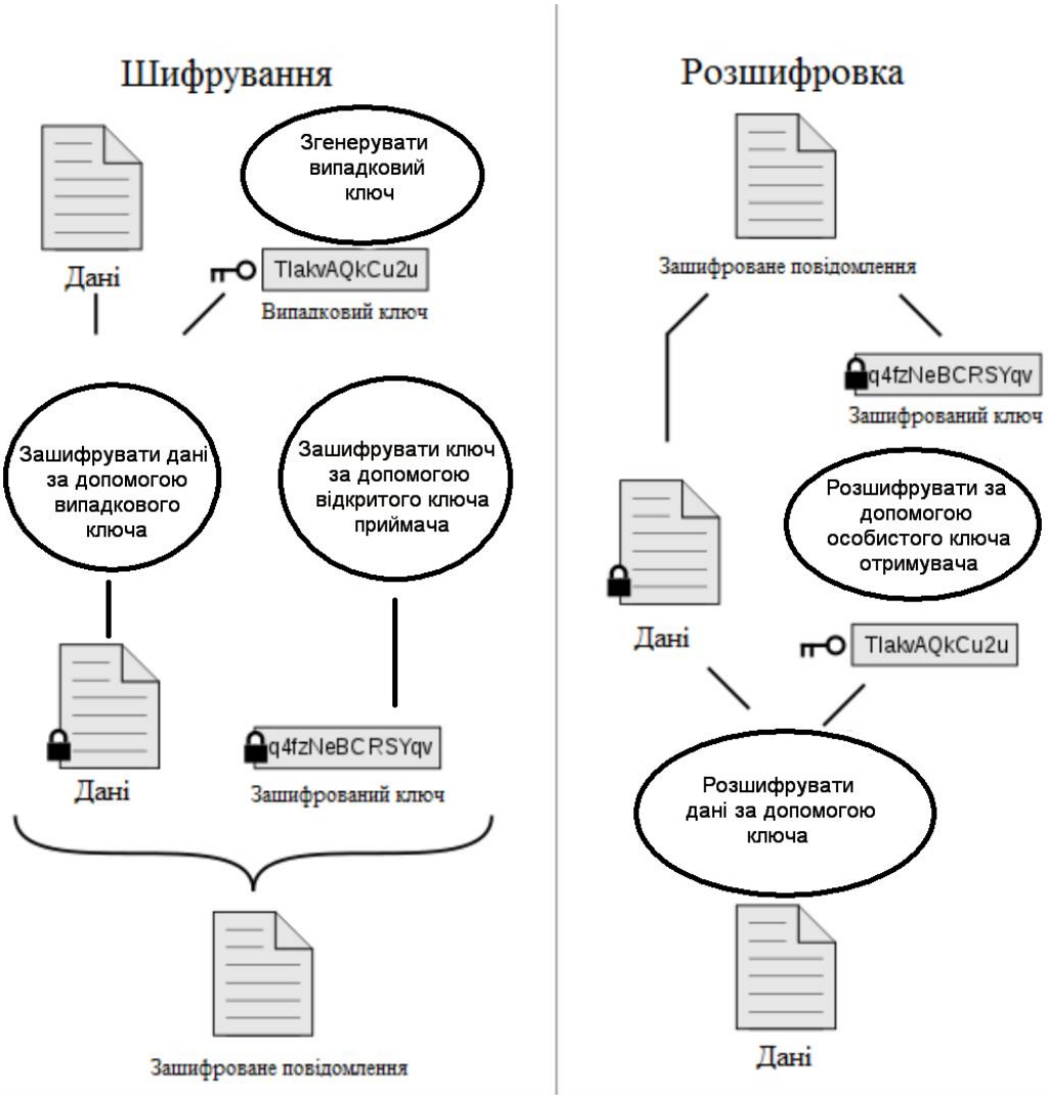


Рисунок 2.2 — Робота PGP шифрування

Головною вразливістю цієї моделі є те, що довгостроковий приватний ключ є “головним ключем” master key. Його компрометація дозволяє супротивнику, який пасивно записував трафік, розшифрувати весь архів минулих повідомлень.

Модель PGP не забезпечує ані досконалої прямої секретності, ані безпеки після компрометації.

Модель “Повного сеансу” на базі Signal Protocol

Сучасним “золотим стандартом” для комунікацій 1-на-1 є протокол Signal, що поєднує механізм асинхронного встановлення сеансу X3DH та алгоритм подвійного храповика Double Ratchet.

Ця архітектура вирішує ключові недоліки PGP. Вона використовує ефемерні тимчасові ключі для кожного сеансу та постійно оновлює їх за допомогою двох “храповиків” — симетричного та асиметричного. Ця комбінація забезпечує найвищі гарантії безпеки. Компрометація довгострокових ключів не дозволяє розшифрувати минулі повідомлення. Компрометація поточного сеансового ключа не дозволяє розшифрувати майбутні повідомлення, оскільки протокол “самовідновлюється” при наступному обміні DH.

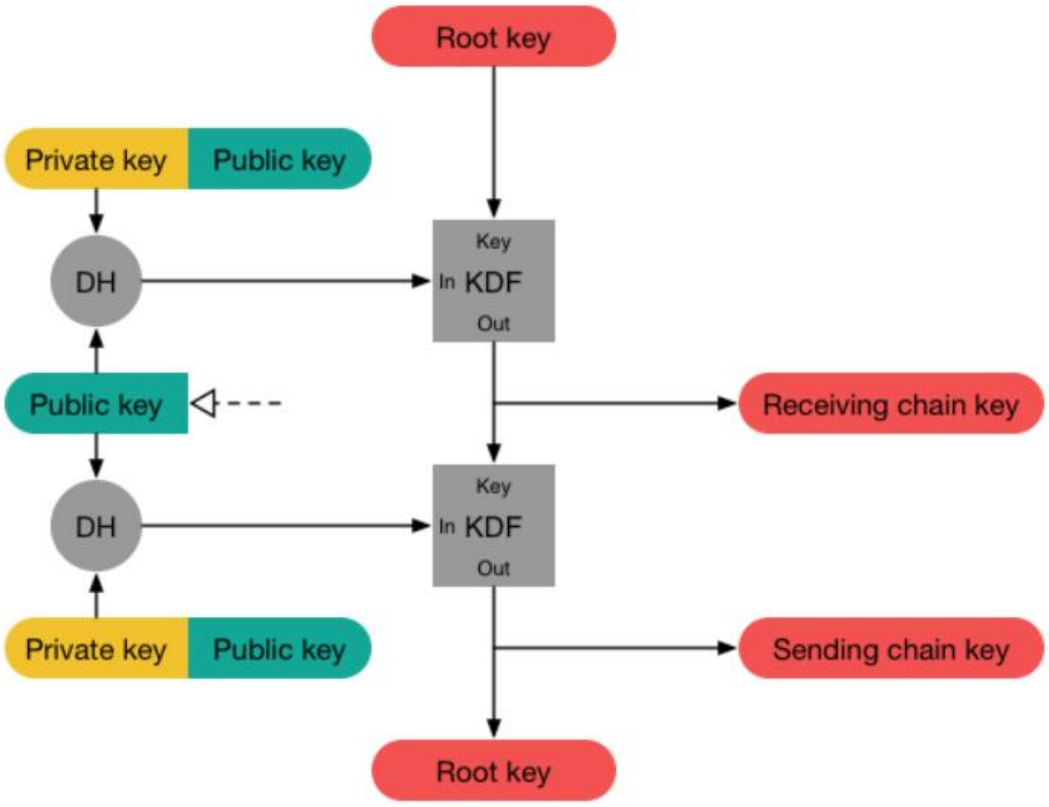


Рисунок 2.3 — Повний крок ratchet в алгоритмі Double Ratchet

Попри свої переваги, ця модель має суттєві архітектурні недоліки у контексті масштабування та зручності використання:

- Класичний підхід Signal до груп має лінійну складність O(n), де кожне повідомлення шифрується та відправляється n-1 разів окремо для кожного учасника . Це робить його вкрай неефективним для великих груп.
- Signal Protocol є монолітним та складним протоколом, що підвищує ризик помилок імплементації.
- Оскільки сервер є “сліпим” ретранслятором , такі базові функції, як синхронізація історії повідомлень між кількома пристроями, стають вкрай нетривіальною та складною інженерною задачею.

Модель “Модульних примітивів” на базі HPKE

Відповіддю на складність монолітних протоколів стала поява стандартизованих, високорівневих “будівельних блоків”. Ключовим серед них є Hybrid Public Key Encryption.

HPKE — це не протокол сеансу, а гнучкий конструктор, що стандартизує перевірений часом гібридний підхід, аналогічний PGP, але модернізований. Він чітко визначає комбінацію трьох примітивів: механізму інкапсуляції ключа KEM, функції виведення ключа KDF та схеми автентифікованого шифрування AEAD.

Головна перевага полягає у простоті, гнучкості та аудитоздатності. Він виконує одне чітке завдання: безпечно надіслати зашифрований блок даних одержувачу, маючи його публічний ключ.

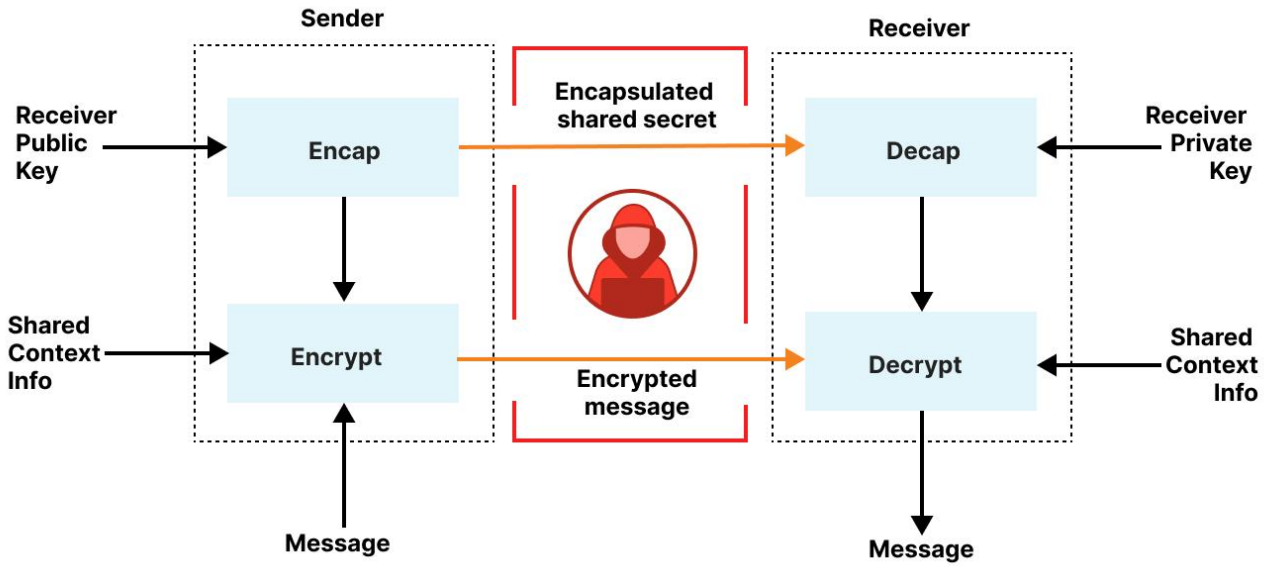


Рисунок 2.4 — Огляд HPKE

HPKE сам по собі не є сеансовим протоколом. Він не надає властивостей PFS або PCS з коробки. Якщо використовувати HPKE з тим самим довгостроковим публічним ключем, ми повертаємося до класичної моделі PGP з її фатальним недоліком — компрометація приватного ключа розкриває всі минулі повідомлення, що були зашифровані цим ключем.

Таблиця 2.1 — Порівняльний аналіз архітектурних підходів

Критерії/Метод	**Модель TLS**	**Модель PGP**	**Модель Signal**	**Модель HPKE**
Справжнє E2EE	Ні	Так	Так	Так
Захист від MitM	Так (на каналі)	Так (автентичність)	Так	Так
Захист від сервера	Ні	Так	Так	Так
PFS	Ні	Ні	Так	Ні
PCS	Ні	Ні	Так	Ні
Підтримка асинхронності	Так	Так	Так	Так
Зручність	Висока	Низька	Дуже низька	Середня/Висока
Складність реалізації	Тривіальна	Середня	Дуже висока	Низька

Існуючі методи не пропонують оптимального рішення, що одночасно задовольняє вимоги абсолютної безпеки та високої зручності, особливо у контексті веб-додатків. Це створює технологічну прогалину та підводить до необхідності обгрунтування модифікованої архітектурної моделі, яка б поєднувала переваги проаналізованих підходів для вирішення поставленої задачі.

2.2 Обгрунтування вибору методу (методики) дослідження

Проведений у аналіз продемонстрував, що жоден з існуючих “чистих” архітектурних підходів не надає повного розв’язання фундаментального протиріччя між криптографічною надійністю та зручністю використання у веб-середовищі. Модель Signal Protocol є надто складною для реалізації та має суттєві обмеження у синхронізації історії, тоді як модель на базі HPKE сама по собі не забезпечує розширених гарантій безпеки, таких як PFS та PCS.

Для досягнення мети дипломної роботи, яка полягає у побудові архітектури та проектуванні системи, що збалансує ці суперечливі вимоги, пропонується методологія на основі гібридної модифікованої моделі.

Ця модель є свідомим компромісом, що синтезує надійність перевірених протоколів та гнучкість сучасних модульних стандартів. Вона складається з трьох ключових методик: керування ідентичністю, захищеного керування ключами та гібридної реалізації протоколів E2EE.

2.2.1 Методика керування ідентичністю та сесіями

У будь-якій клієнт-серверній системі процес автентифікації (підтвердження особи користувача) та авторизації (надання прав доступу) є базовим. У моделі E2EE цей процес має критичне доповнення: він повинен бути криптографічно відокремленим від процесів наскрізного шифрування. Сервер автентифікує користувача для надання доступу до сервісу, наприклад доставки повідомлень,

але ця автентифікація не повинна надавати серверу доступ до змісту самих повідомлень.

Для реалізації цього завдання обирається механізм на базі JSON Web Tokens (JWT) [52] з структурою на рисунку 2.5.

Рисунок 2.5 — Структура JSON Web Tokens (JWT)

На відміну від класичних сесій, що зберігають стан на сервері, JWT є stateless механізмом. Серверу не потрібно звертатися до бази даних для валідації кожної сесії; він перевіряє криптографічний підпис токена. Це ідеально підходить для сучасних API та підключень у реальному часі з WebSockets.

Зберігання JWT на клієнті є окремою проблемою безпеки. Зберігання у LocalStorage [53] робить токен вразливим до атак типу Cross-Site Scripting XSS, що є неприпустимим у “ворожому веб-середовищі”.

Обраний метод зберігання, де токени access та refresh [54] будуть зберігатися у HttpOnly cookies [55]. Цей прапор забороняє будь-який доступ до cookie з боку JavaScript, ефективно захищаючи сесію користувача від викрадення через XSS-атаки.

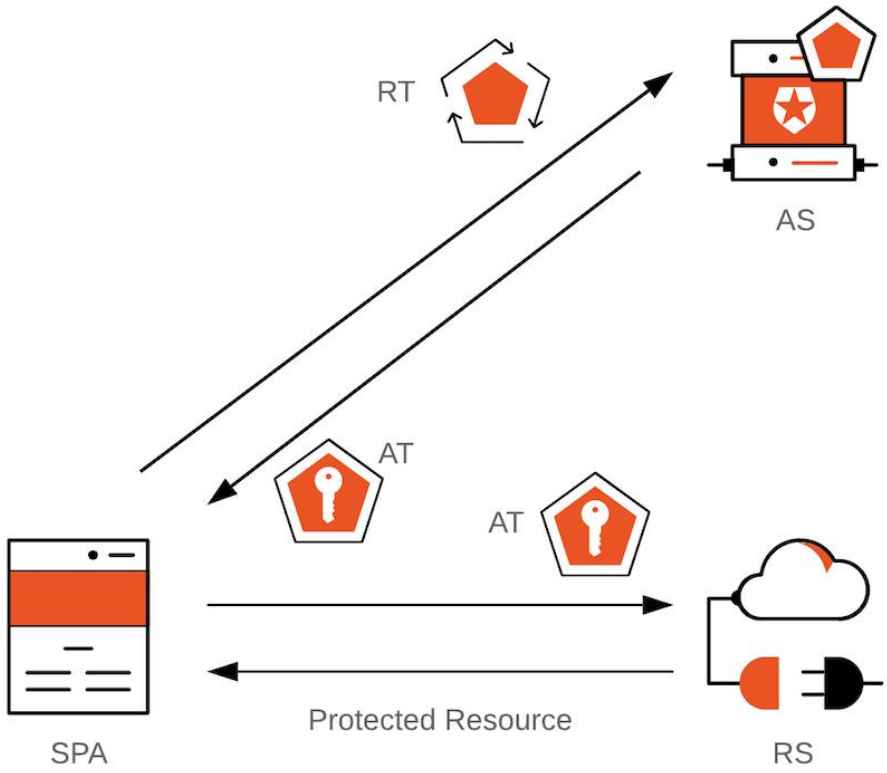


Рисунок 2.6 — Принцип роботи refresh token

2.2.2 Методика захищеного керування ключами на клієнті

Найскладнішим завданням E2EE у вебі є керування приватним ключем ідентичності користувача.

Проблема полягає в запитанні, де зберігати приватний ключ:

- Зберігання лише у браузері є незручним — користувач втратить доступ до всіх своїх чатів при очищенні кешу або вході з нового пристрою.
- Зберігання ключа на сервері у відкритому вигляді суперечить самій ідеї E2EE.

Для вирішення цієї дилеми використано методику “зашифрованого серверного сховища” (Server-Side Encrypted Storage). Вона дозволяє серверу виступати у ролі “сліпого” сховища для приватних ключів, не маючи до них доступу.

Методика складається з наступних етапів, що виконуються виключно на клієнті:

- Під час реєстрації, Web Cryptography API генерує довгострокову пару ключів, наприклад на базі Curve25519 для HPKE.
- Користувач вводить свій пароль. Цей пароль пропускається через криптографічну функцію виведення ключа на основі пароля для отримання майстер-ключа шифрування.
- Приватний ключ SK шифрується на клієнті за допомогою майстер-ключа з використанням симетричного алгоритму, наприклад AES-GCM. Результатом є зашифрований “блоб” CSK.
- Клієнт відправляє на сервер для зберігання лише свій публічний ключ PK та зашифрований приватний ключ CSK.

Для кроку 2 обирається стандарт PBKDF2 (Password-Based Key Derivation Function 2) [56]. На відміну від простих геш-функцій, як SHA-256, PBKDF2 спеціально розроблений для протидії атакам перебору brute-force [57]. Він використовує два параметри: сіль (salt) унікальне випадкове значення для кожного користувача та кількість ітерацій, наприклад більше 100000. Це робить процес виведення ключа обчислювально повільним, що унеможливлює швидкий перебір паролів.

В результаті, сервер ніколи не бачить ані пароль користувача, ані його приватний ключ. При вході з нового пристрою, користувач вводить пароль, клієнт завантажує CSK з сервера, локально генерує майстер-ключ і розшифровує свій приватний ключ. Ця методика забезпечує зручність, синхронізацію, не жертвуючи безпекою.

2.2.3 Методика реалізації гібридних протоколів E2EE

Як було обгрунтовано в підрозділі 1.4, вибір між максимальною безпекою та зручністю є хибним. Тому, основою побудови системи захищеного обміну повідомленнями обрано методологію гібридного шифрування, яка реалізується

через архітектурний патерн Digital Envelope. Цей підхід дозволяє вирішити фундаментальне протиріччя криптографічних систем: необхідність забезпечення високої швидкості обробки даних, притаманної симетричним алгоритмам, та безпечного розподілу ключів, що забезпечується асиметричною криптографією.

Концептуальна модель обробки даних

Методика передбачає повну відмову від використання одного ключа для шифрування і даних, і каналу зв'язку. Натомість, процес захисту інформації розділяється на два незалежних, але послідовних рівня.

На рівні інкапсуляції даних забезпечується конфіденційність безпосереднього змісту повідомлення. Для кожного окремого повідомлення або сесії генерується унікальний ефемерний (тимчасовий) симетричний ключ. Використання симетричних криптоалгоритмів на цьому етапі обумовлено їхньою високою обчислювальною ефективністю, що є критичним для веб-додатків, які працюють у браузері користувача та можуть обробляти великі обсяги даних.

Рівень інкапсуляції ключа відповідає за безпечну передачу симетричного ключа, згенерованого на попередньому етапі, до отримувача. Замість шифрування самих даних, асиметричний алгоритм, на базі сучасного стандарту HPKE, використовується виключно для шифрування або обгортання короткого симетричного ключа.

Застосування методики в асинхронному веб-середовищі

Ключовою особливістю розроблюваної системи є необхідність підтримки асинхронного зв'язку, де відправник і отримувач можуть не бути в мережі одночасно. Класичні сесійні протоколи вимагають складного узгодження станів в реальному часі.

Запропонована методика вирішує цю проблему шляхом трактування кожного повідомлення як автономного криптографічного об'єкта. Структура

такого об’єкта містить: шифротекст, симетричний ключ зашифрований публічним ключем отримувача, метадані.

Така архітектура перетворює сервер на “сліпе” сховище. Сервер бази даних оперує лише зашифрованими контейнерами, не маючи математичної можливості отримати доступ до ключів дешифрування, оскільки операція розгортання конверта можлива виключно на стороні клієнта за наявності приватного ключа отримувача.

Таким чином, методика гібридного шифрування є оптимальним базисом для побудови захищеного веб-додатку, забезпечуючи баланс між безпекою, продуктивністю клієнтської сторони та архітектурною гнучкістю.

2.3 Вибір засобів та технологій реалізації програмного прототипу

Для практичної реалізації та валідації обгрунтованої гібридної архітектурної моделі необхідна розробка програмного прототипу. Вибір технологічного стеку для цього прототипу є критичним, оскільки він має не лише забезпечувати необхідну функціональність, але й відповідати суворим вимогам безпеки, продуктивності та сучасної веб-розробки. Кожен компонент стеку обирався, виходячи з його здатності вирішувати специфічні завдання нашої архітектури.

2.3.1 Клієнтська частина та засоби криптографії

Клієнтська частина є ядром всієї системи, оскільки саме у браузері користувача виконуються всі криптографічні операції.

Мовою розробки як клієнтської, так і серверної частини обрано TypeScript [58]. Робота з критографією вимагає абсолютної точності при маніпуляції даними, наприклад чітке розрізнення між форматами string, ArrayBuffer та Uint8Array. Статична типізація TypeScript допомагає запобігти цілим класам помилок під час виконання, що є фундаментальною вимогою для побудови безпечної системи.

Для побудови користувацького інтерфейсу обрано прогресивний фреймворк Vue.js [59]. Його ключовою перевагою для нашого завдання є компонентна архітектура. Вона дозволяє інкапсулювати складну, стан-залежну логіку у самодостатні блоки. Наприклад, логіка шифрування в чаті, що реалізує алгоритм HPKE, може бути повністю ізольована в окремому компоненті, не впливаючи на решту додатку.

Стилізація сторінок виконується завдяки Tailwind CSS [60] — це utility-first підхід, що дозволяє максимально прискорити розробку прототипу та зосередитись на функціональності, а не на написанні CSS.

Для керування станом додатку обрано Pinia [61], офіційний менеджер стану для Vue. У нашій архітектурі Pinia виконує критично важливу функцію: вона керує глобальним станом, зокрема тимчасовим зберіганням розшифрованого приватного ключа користувача в оперативній пам’яті. Pinia забезпечує реактивне та безпечне сховище для цих даних, доступне лише під час активної сесії користувача.

Основою для будь-яких криптографічних операцій слугує Web Cryptography API (WebCrypto). Це низькорівневий API, вбудований у сучасні браузери, що надає доступ до криптографічних примітивів (AES-GCM, PBKDF2, ECDH). Його вибір є безальтернативним, оскільки він виконує обчислювально-інтенсивні операції у захищеному, нативному контексті, що значно безпечніше та продуктивніше за будь-які реалізації на чистому JavaScript.

Крім самостійно реалізованого шифрування в чаті з Web Cryptography API, що базується на HPKE, буде використано спеціалізовану бібліотеку, наприклад hpke-js [62]. Ця бібліотека є абстракцією, що коректно реалізує даний стандарт, використовуючи WebCrypto API під капотом. Це дозволяє нам уникнути помилок при низькорівневій реалізації протоколу, що є поширеною причиною вразливостей. Але рекомендується використовувати @hpke/core разом з модулями розширення (такими як @hpke/chacha20poly1305) замість hpke-js.

Для реалізації протоколу, що базується на Double Ratchet, може бути використана бібліотека @privacyresearch/libsignal-protocol-typescript [63], яка

реалізує протокол секретності з механізмом рачетування, що працює в синхронних і асинхронних середовищах обміну повідомленнями. Ця бібліотека використовує WebCrypto API для симетричної криптографії та генерації випадкових чисел. Вона використовує реалізацію інтерфейсу AsyncCurve в curve25519-typescript для операцій з відкритим ключем. Для кожного з них передбачені функціональні значення за замовчуванням, але ви можете задати свої власні, наприклад з міркувань продуктивності або безпеки.

2.3.2 Серверна частина та керування даними

Сервер у нашій архітектурі виконує роль недовіреного ретранслятора повідомлень та “сліпого” сховища для зашифрованих даних, приватних ключів та історії.

В якості серверного середовища обрано Node.js [64], щоб зберегти однорідність мови програмування клієнту і серверу. Його асинхронна, керована подіями модель вводу-виводу є галузевим стандартом для побудови високопродуктивних додатків реального часу, зокрема чатів. Node.js здатний ефективно обробляти тисячі одночасних WebSocket-з’єднань, що є ключовою вимогою для нашої системи. Використання TypeScript на сервері, як і на клієнті, дозволяє створити єдиний технологічний стек, що спрощує розробку та дає змогу спільно використовувати моделі валідації.

Для взаємодії з даними обрано PostgreSQL [65] як реляційну СУБД та Prisma [66] як ORM (Object-Relational Mapper).

PostgreSQL обрано через його надійність та здатність ефективно керувати структурованими даними та зв’язками (користувачі, чати, учасники чатів і т.д.). PostgreSQL є найнадійнішим та найпотужнішим відкритим рішенням для визначених завдань.

Prisma забезпечує повністю типобезпечний доступ до бази даних, що ідеально інтегрується з TypeScript. Крім того, Prisma автоматично захищає від SQL-ін’єкцій, що є базовим рівнем безпеки серверної частини.

2.3.3 Протоколи комунікації та механізми безпеки

Архітектура клієнт-серверних веб-додатків обміну даними вимагає двох різних моделей комунікації: синхронної, для автентифікації та отримання даних і асинхронної, для миттєвої доставки повідомлень.

Синхронна комунікація (REST API)

Синхронна комунікація реалізована за допомогою мінімалістичного фреймворку Express.js [67] для Node.js, який використовується для реалізації REST API [68]. Цей API використовується для операцій, що не вимагають реального часу: реєстрація, автентифікація, завантаження публічних ключів користувачів та отримання зашифрованої історії.

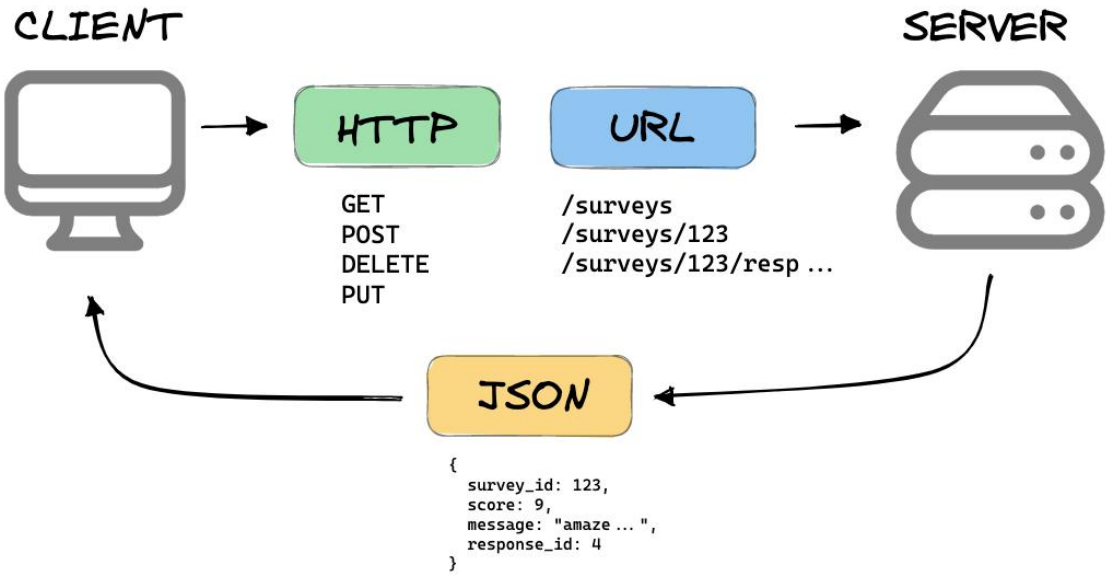


Рисунок 2.7 — Базовий REST API

Для реалізації методики автентифікації використовується комбінація бібліотек jsonwebtoken та cookie-parser.

Jsonwebtoken — це бібліотека для реалізації методики автентифікації на базі JWT Використовується на сервері для генерації та валідації токенів доступу.

Cookie-parser — це middleware для Express.js, необхідний для роботи з HttpOnly cookies. Він дозволяє серверу зчитувати JWT, що надійно зберігаються у cookie клієнта, куди їх поміщає клієнтський Axios.

Axios — це HTTP-клієнт на основі Promise, що використовується для взаємодії з REST API сервера (автентифікація, реєстрація, отримання списків контактів та зашифрованої історії) [69].

Асинхронна комунікація (Real-time Transport)

Асинхронна комунікація є транспортним рівнем на WebSockets [70] для миттєвої доставки E2EE-повідомлень. Для цього обрано бібліотеку Socket.io [71] (клієнтська та серверна частини) яка відповідає за організацію двонаправленого зв'язку в реальному часі між клієнтами.

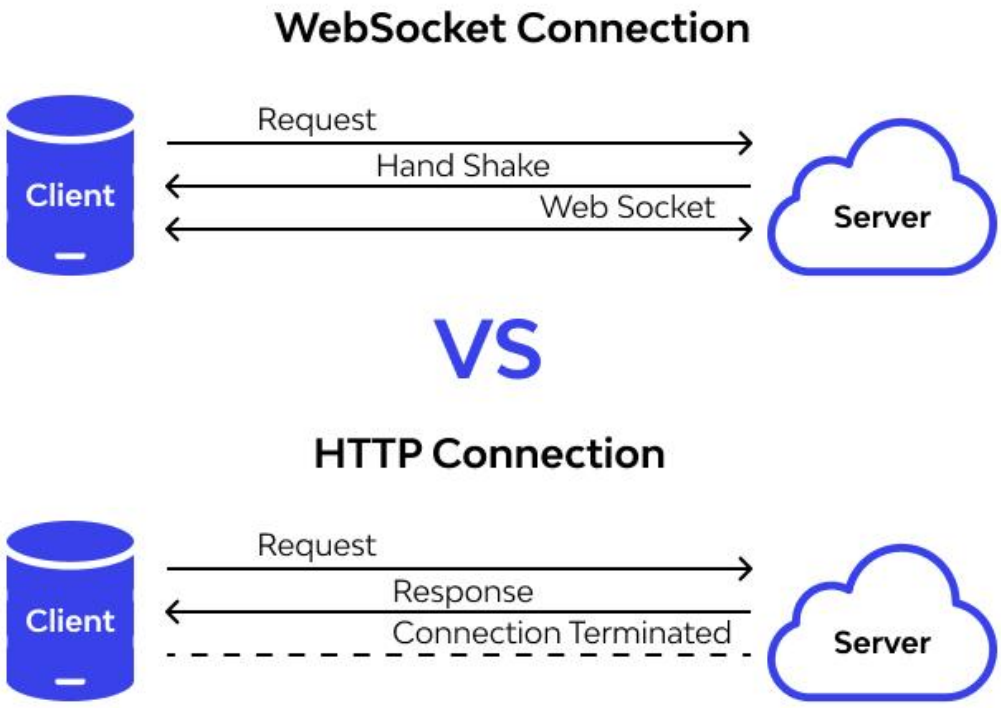


Рисунок 2.8 — WebSocket проти HTTP з'єднання

На відміну від чистих WebSockets, Socket.io надає критично важливий шар абстракції, що керує автоматичним перепідключенням, має механізми відкату fallback до HTTP long-polling та, що найважливіше, підтримує “кімнати” rooms, що є ідеальним механізмом для групових чатів.

Також, для забезпечення цілісності та безпеки наскрізного потоку даних використовуються CORS та Zod [72].

CORS є обов’язковим middleware для Express.js, що керує політикою доступу до ресурсів з інших доменів. Це обов’язковий елемент, оскільки дозволяє веб-клієнту, розміщеному на одному домені, безпечно взаємодіяти з API, розміщеним на іншому.

Zod — це бібліотека для валідації схем даних. Використовується як на клієнті, так і на сервері для валідації схем всіх вхідних даних, наприклад, тіла запиту API або вмісту WebSocket-повідомлення. Це створює файрвол на рівні додатку, що відкидає будь-які невалідні або шкідливі дані ще до їх обробки.

2.4 Аналіз і узагальнення фактичного матеріалу (Стандарти та API)

Методологія та вибір засобів базуються на фундаментальних властивостях конкретних криптографічних стандартів та програмних інтерфейсів. Для підтвердження коректності нашого підходу необхідно провести їх глибокий аналіз. Цей аналіз слугує узагальненням того фактичного матеріалу, на якому будується вся архітектура.

2.4.1 Декомпозиція Web Cryptography API

Центральним елементом нашої архітектури є Web Cryptography API, оскільки він єдиний надає стандартизований доступ до криптографічних операцій у “ворожому середовищі” веб-браузера. Усі операції виконуються через об’єкт window.crypto.subtle, що підкреслює їхню чутливість.

Ключовим об’єктом в WebCrypto є CryptoKey. Цей API дозволяє генерувати ключі, як симетричні, так і асиметричні, з прапорцем extractable: false. Теоретично, цей прапор означає, що JavaScript-код може використовувати ключ для операцій шифрування/розшифрування, але ніколи не може прочитати (експортувати) сам матеріал приватного ключа. Це виглядає як ідеальне рішення для безпечного зберігання ключів у браузері.

Однак, фундаментальний аналіз моделі загроз вебу показує, що цього недостатньо. Проблема полягає у довірі до коду. Скомпрометований сервер може надіслати клієнту модифікований JavaScript. Навіть якщо приватний ключ має прапор extractable: false і зберігається в IndexedDB, зловмисний код, виконуючись у тому ж походженні origin, може:

- Перехопити відкритий текст повідомлення перед тим, як воно буде передано на шифрування.
- Перехопити відкритий текст після того, як він буде отриманий з функції розшифрування.
- Використовувати сам ключ, не експортуючи його, для розшифрування даних та подальшої їх крадіжки.

Саме цей аналіз доводить, що просте покладання на extractable: false є нежиттєздатним у нашій моделі загроз. Це напряму обгрунтовує необхідність нашої методики: ми не можемо довіряти браузерному сховищу, натомість ми маємо застосувати додатковий шар захисту, зашифрувавши приватний ключ за допомогою PBKDF2 та пароля користувача.

2.4.2 Декомпозиція HPKE

Стандарт Hybrid Public Key Encryption є основою нашого захищеного спілкування. Його перевага в тому, що він є не монолітним протоколом, а конструктором, що поєднує три перевірені криптографічні примітиви: KEM, KDF та AEAD.

KEM (Key Encapsulation Mechanism) — механізм інкапсуляції ключа. Це асиметрична операція. Замість того, щоб відправник генерував ключ і шифрував його публічним ключем одержувача, як у PGP, KEM виконує операцію, наприклад обмін Діффі-Хеллмана, в результаті якої обидві сторони отримують спільний секрет.

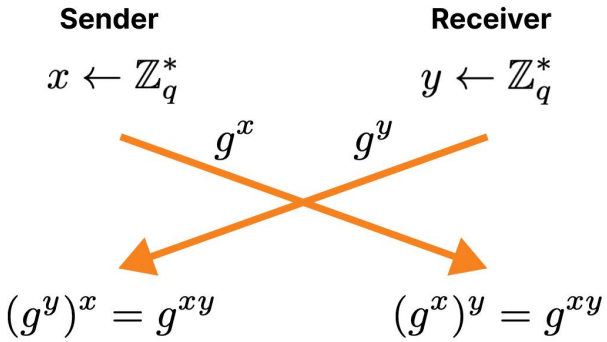


Рисунок 2.9 — Неавтентифікований обмін ключами за алгоритмом Діффі-Хеллмана

KDF (Key Derivation Function) — функція виведення ключа, наприклад HKDF. Ця функція бере спільний секрет, отриманий від KEM, і безпечно “розтягує” його, генеруючи з нього один або декілька надійних симетричних ключів.

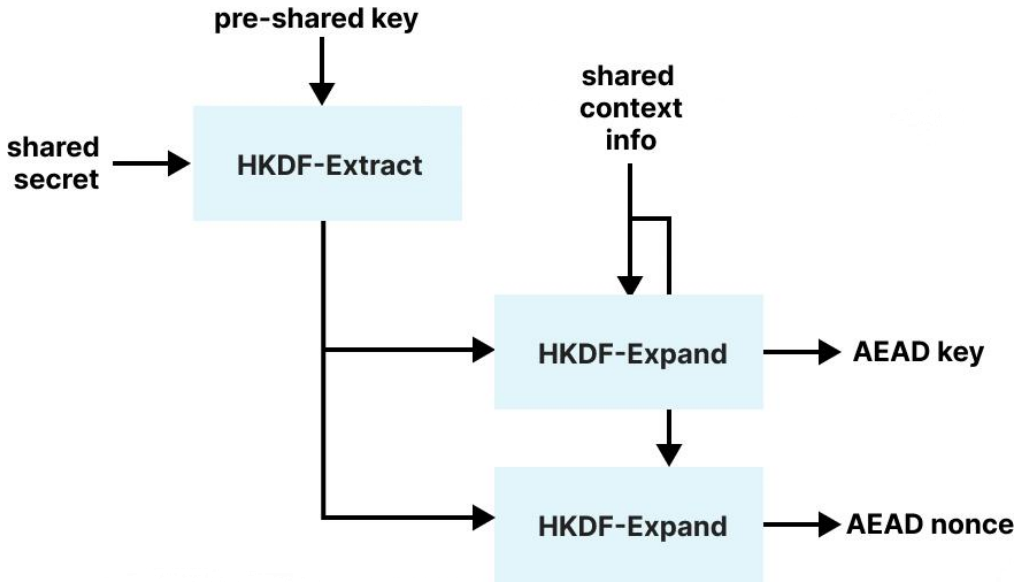


Рисунок 2.10 — Спрощений графік ключів HPKE

AEAD (Authenticated Encryption with Associated Data) — автентифіковане шифрування. Це симетричний шифр, наприклад AES-GCM або ChaCha20-Poly1305, який бере ключ від KDF і шифрує саме повідомлення.

Коли наш додаток виконує операцію шифрування, він послідовно виконує ці три кроки, створюючи єдиний, безпечний зашифрований пакет. Простота та модульність цього стандарту робить його ідеальним для асинхронного E2EE, де кожне повідомлення може бути зашифроване незалежно.

2.4.3 Аналіз PBKDF2 та AES-GCM

Ці два примітиви є основою нашої методики захисту приватного ключа, яка описана в підрозділі 2.2.2.

PBKDF2 (Password-Based Key Derivation Function 2) — це стандарт NIST, обраний для перетворення пароля користувача на майстер-ключ. Його вибір замість простого гешу, як SHA-256, є критичним. PBKDF2 розроблений для протидії атакам перебору brute-force завдяки двом механізмам:

- Сіль (Salt) — це унікальне випадкове значення, що додається до пароля перед гешуванням. Це робить неможливим використання попередньо обчислених rainbow tables.
- Ітерації, за якими функція виконує гешування тисячі разів. Це робить обчислення одного ключа повільним для користувача, але непомітним, проте вкрай повільним для зловмисника, що намагається перебрати мільярди паролів.

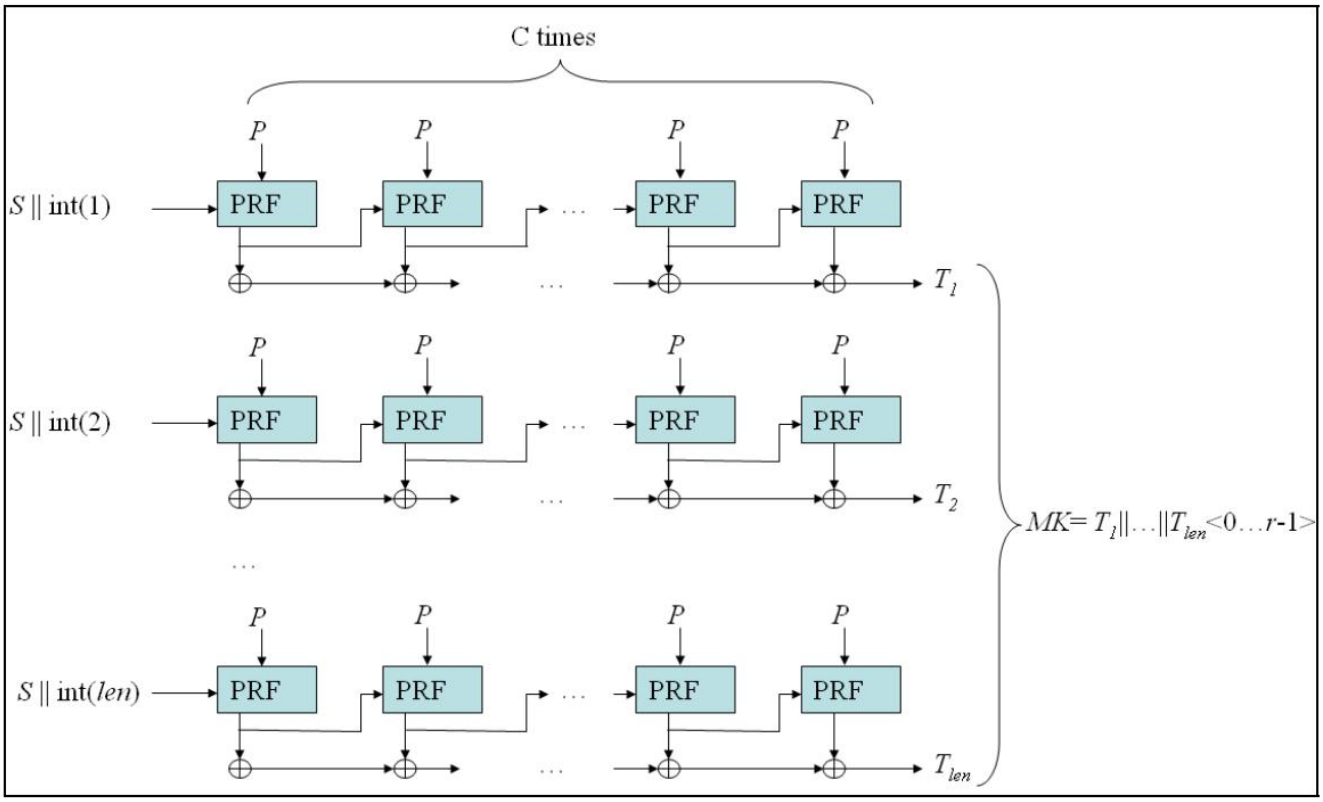


Рисунок 2.11 — Алгоритмічне представлення ітеративного процесу PBKDF2

AES-GCM (Galois/Counter Mode) — це режим симетричного шифрування (AEAD) [73], обраний для шифрування приватного ключа користувача і можливо повідомлень за допомогою майстер-ключа, отриманого з PBKDF2. Його вибір також не є випадковим. На відміну від старих режимів, як CBC, AES-GCM надає дві гарантії одночасно:

- Конфіденційність, адже приватний ключ або повідомлення надійно зашифровані.
- Цілісність та Автентичність, адже завдяки тегу автентифікації, GCM гарантує, що зашифровані дані не були змінені або модифіковані зловмисником під час зберігання на сервері.

GCM поєднує AES з режимом лічильника для шифрування та режимом Галуа для автентифікації. Це забезпечує як конфіденційність, так і цілісність даних.

У AES/GCM режим лічильника CTR перетворює блоковий шифр у потоковий шифр шляхом шифрування послідовних значень лічильника. Цей

метод дозволяє здійснювати паралельну обробку та ефективні операції шифрування/дешифрування.

Режим Galois забезпечує автентифікацію повідомлень шляхом генерації тегу, який перевіряє справжність даних. Це гарантує, що дані не були підроблені.

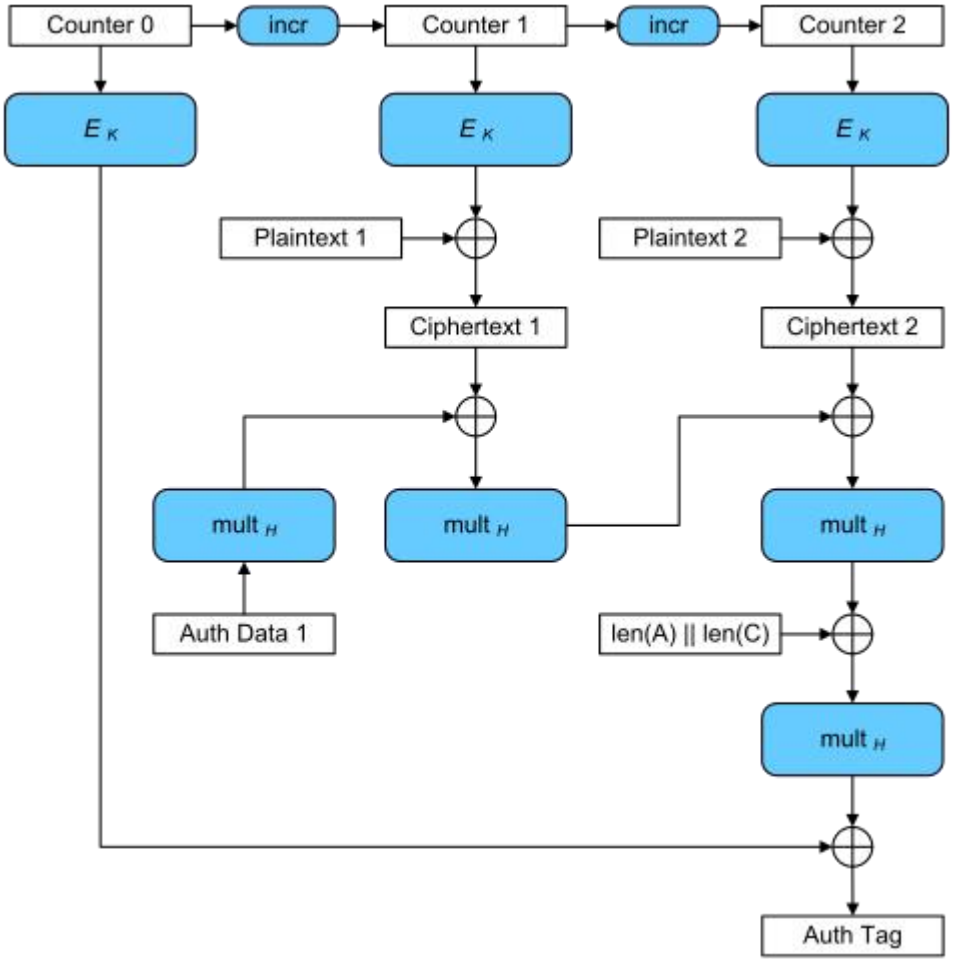


Рисунок 2.12 — Алгоритм шифрування в режимі GCM

Таким чином, аналіз цих стандартів підтверджує, що обрані інструменти надають точні технічні можливості, необхідні для реалізації нашої гібридної, захищеної та водночас зручної архітектури.

Можна зробити висновок, що у даному розділі було проведено глибокий аналіз та обгрунтування методології, необхідної для розв’язання поставленої науково-технічної задачі.

Було виконано порівняльний аналіз існуючих архітектурних підходів до реалізації E2EE-систем, включаючи модель на базі TLS, модель PGP, сеансову модель Signal та модульний підхід на базі HPKE. Цей аналіз виявив, що жоден з існуючих методів, в повній мірі, не пропонує оптимального рішення, яке б одночасно задовільняло вимоги абсолютної безпеки та високої зручності у веб-середовищі.

У відповідь на цю прогалину, було обгрунтовано вибір на користь гібридної архітектурної моделі. Ця методологія є ядром дослідження і включає методику керування ідентичністю, методику захищеного керування клієнтськими ключами та реалізацію E2EE.

Обрано та детально аргументовано технологічний стек, необхідний для реалізації програмного прототипу. Цей стек включає клієнтські та серверні технології, а також засоби комунікації, що повністю відповідають вимогам обгрунтованої методології.

Проведено декомпозицію і аналіз стандартів та API. Цей аналіз підтвердив, що обрані примітиви надають необхідні властивості та гарантії безпеки, на яких базується вся запропонована архітектура.

Таким чином, у поточному розділі було повністю сформовано методологічну та інструментальну базу дослідження. Визначені підходи та засоби є основою для наступного етапу роботи — детального проектування архітектури і прототипу шифрування в веб-середовищі, що буде викладено у наступному розділі.

3 АРХІТЕКТУРНІ ТА ПРОЕКТНІ РІШЕННЯ ДЛЯ ЗАХИЩЕНОГО ОБМІНУ ДАНИМИ В ВЕБ-СЕРЕДОВИЩІ

3.1 Проектування гібридної криптографічної архітектури

3.1.1 Стратегія управління ключами та захисту ідентичності

Основою системи захищеного обміну повідомленнями є криптографічна ідентичність користувача. На відміну від традиційних веб-додатків, де ідентичність визначається записом у базі даних сервера, у спроектованій E2EE-системі ідентичність — це володіння парою асиметричних криптографічних ключів.

Генерація ключів та вибір криптографічного примітиву Для реалізації механізму інкапсуляції ключів KEM в рамках стандарту HPKE було обрано еліптичну криву NIST P-256 (secp256r1). Цей вибір обгрунтований її широкою підтримкою у веб-браузерах через Web Cryptography API та відповідністю стандартам FIPS.

Процес генерації ключів відбувається виключно на стороні клієнта, в браузері, і базується на математичних властивостях еліптичних кривих над скінченними полями. Алгоритм генерації виглядає наступним чином:

1) Генерація ентропії.
2) Формування приватного ключа.
3) Обчислення публічного ключа.

За допомогою криптографічно стійкого генератора випадкових чисел CSPRNG, доступного через метод window.crypto.getRandomValues, браузер генерує випадкове ціле число d. Це число повинно знаходитися в діапазоні:

$$1 < d < n - 1,$$

де n — порядок базової точки G кривої P-256.

Згенероване число d стає приватним ключем користувача. Його безпека залежить виключно від якості ентропії та недоступності для сторонніх осіб.

Публічний ключ Q обчислюється шляхом скалярного множення базової точки кривої G на приватний ключ d:

$$Q = d \times G,$$

де Q — публічний ключ;

d — приватний ключ;

G — базова точка кривої.

Операція скалярного множення на еліптичній кривій є “односторонньою функцією”: знаючи d і G, легко обчислити Q, але знаючи Q і G, обчислити d практично неможливо, адже проблема дискретного логарифмування.

Отримані об’єкти CryptoKey зберігаються в оперативній пам’яті браузера.

Проблема персистентності та зберігання приватного ключа

Після генерації ключів виникає критична архітектурна проблема: де і як безпечно зберігати приватний ключ identity key між сесіями? Веб-браузер, за своєю природою, є ненадійним середовищем для довгострокового зберігання секретів. Локальні сховища вразливі до XSS-атак, а повне видалення даних браузера призводить до безповоротної втрати ідентичності. Це вимагає впровадження механізму експорту та відновлення ключа.

Використання мнемонічної фрази (Seed Phrase / BIP-39). Цей метод, запозичений з криптовалютних гаманців, передбачає перетворення ентропії приватного ключа у читабельну послідовність з 12 або 24 слів згідно зі стандартом BIP-39. Система генерує слова, а користувач зобов’язаний фізично записати їх на папір. Для входу на новому пристрої користувач вручну вводить ці слова, а система математично відновлює з них приватний ключ.

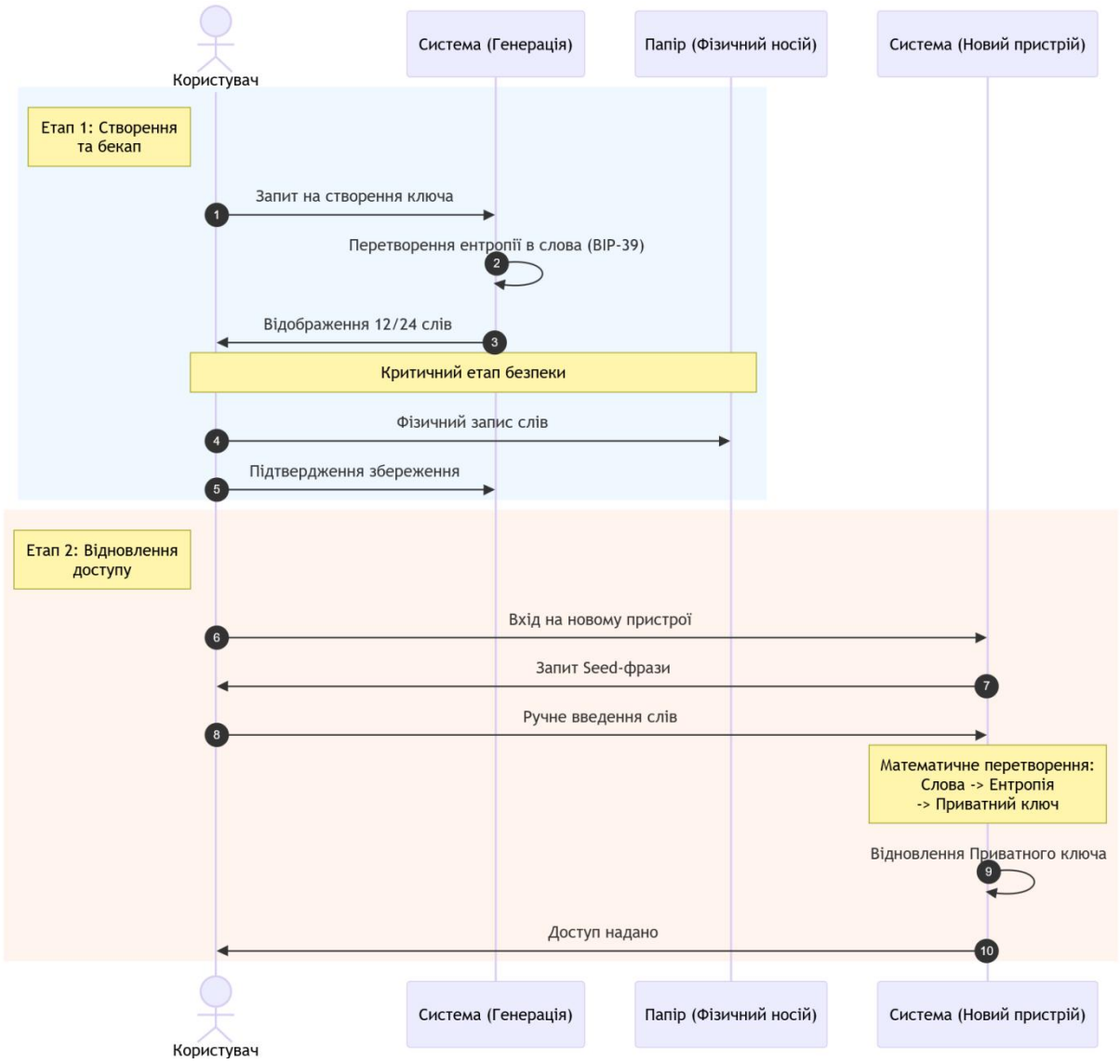


Рисунок 3.1 — Алгоритм використання Seed Phrase

Приватний ключ або фразу можна зберегти як Keystore File. Метод передбачає експорт зашифрованого приватного ключа у файл, наприклад .json або бінарний формат, який користувач завантажує на свій пристрій. Для входу в систему користувач повинен завантажити цей файл назад у веб-додаток та ввести пароль для розшифрування.

Потрібно враховувати, що більшість користувачів не мають можливості керувати власними ключами. Вони допустять помилку при переписуванні, втратять свою основну фразу або файл, а потім назавжди заблокують доступ до свого облікового запису та даних.

Прив’язка пристроїв через QR-код. У цьому підході приватний ключ ніколи не покидає пристрій, на якому був створений, у довгостроковій перспективі. Новий пристрій, наприклад десктоп, генерує власну ефемерну пару ключів і показує QR-код. Основний пристрій, де користувач вже автентифікований, сканує код і через захищений канал передає зашифровану копію свого ключа ідентичності. Якщо користувач втрачає свій єдиний пристрій, або всі свої пристрої, ключ втрачається назавжди. Ця модель вимагає, щоб користувач мав принаймні один активний, довірений пристрій, щоб додати новий.

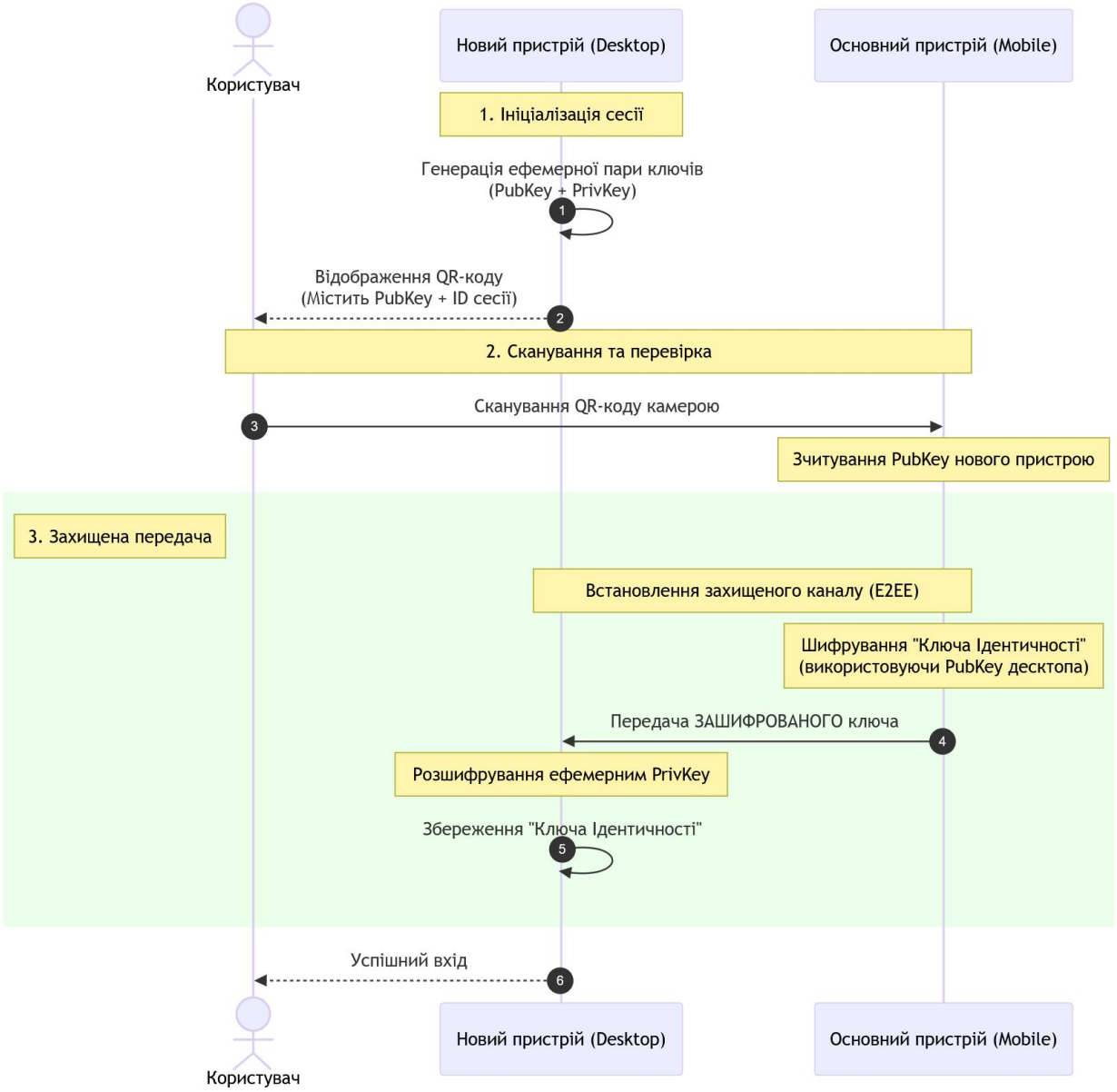


Рисунок 3.2 — Алгоритм прив’язки через QR-код

Серверне зберігання зашифрованого приватного ключа — це підхід, який є компромісом між зручністю хмарної синхронізації та безпекою Zero-Knowledge. Він передбачає зберігання приватного ключа на сервері, але у вигляді криптографічного контейнера, зашифрованого ключем, похідним від пароля користувача. Сервер виступає лише як сховище байтів і не має математичної можливості розшифрувати ключ. Алгоритм дій:

1) Користувач вводить пароль.
2) На клієнті з пароля та випадкової солі генерується симетричний ключ шифрування Wrapping Key.
3) Приватний ключ шифрується цим симетричним ключем.
4) Сервер отримує і зберігає лише шифротекст. Сервер не знає пароля і не має математичної можливості розшифрувати ключ.
5) При вході клієнт завантажує шифротекст з сервера і розшифровує його локально, повторно генеруючи ключ з введеного пароля.

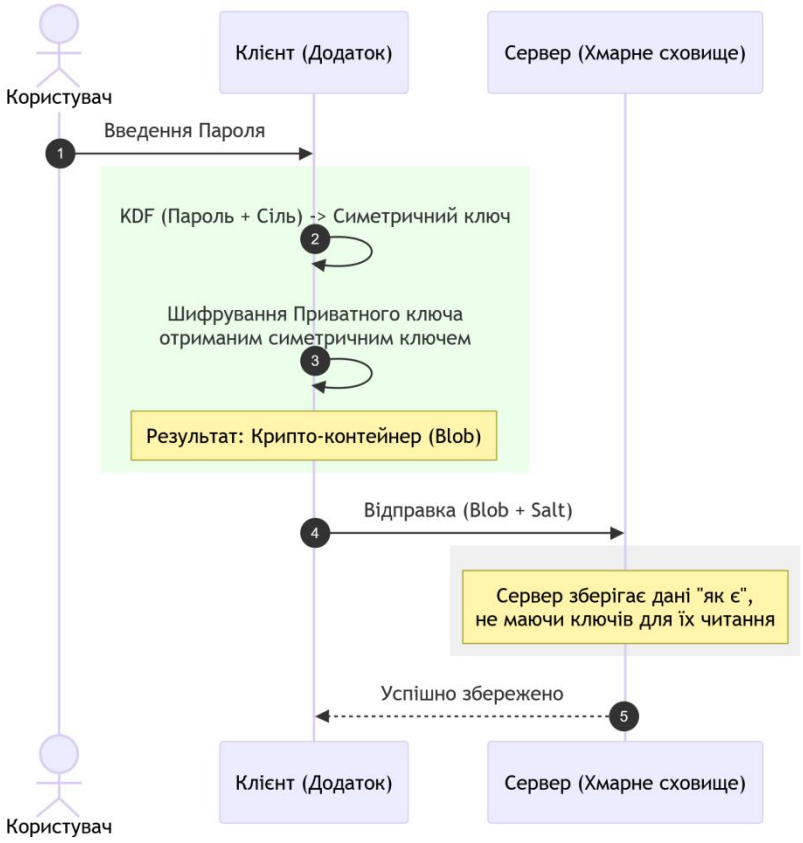


Рисунок 3.3 — Шифрування та збереження приватного ключа

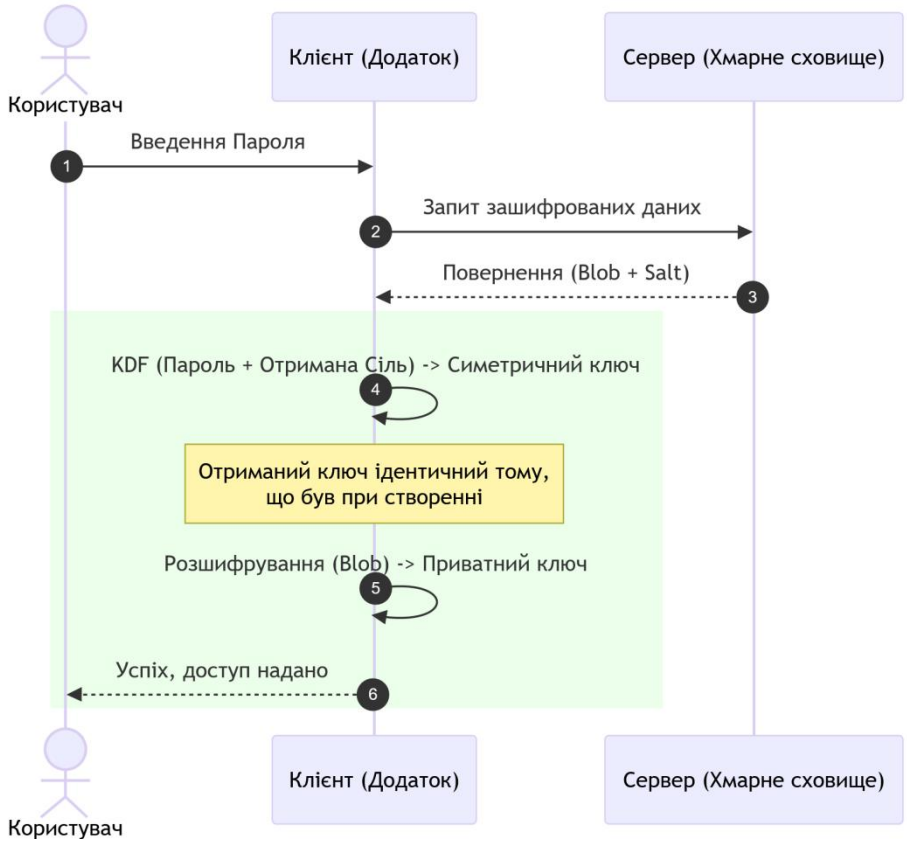


Рисунок 3.4 — Вхід та розшифрування приватного ключа

Для реалізації в рамках дипломної роботи було обрано саме цей варіант серверного зберігання зашифрованого приватного ключа. Вибір обумовлений наступними факторами:

- На відміну від Seed-фраз, які користувачі часто гублять, або Keystore-файлів, які прив’язують до конкретного пристрою, вхід за логіном та паролем є стандартом для веб-додатків. Це знижує поріг входження.
- Користувач може отримати доступ до своїх повідомлень з будь-якого пристрою, просто знаючи свій пароль. Немає потреби переносити файли чи мати під рукою основний пристрій для сканування QR.
- За умови використання надійних алгоритмів, цей метод забезпечує властивість Zero-Knowledge. Сервер не знає пароля і не бачить ключа.

Для захисту від атак повного перебору brute-force на стороні сервера або у випадку витоку бази даних, критично важливим є вибір параметрів алгоритму деривації ключа. Згідно з рекомендаціями OWASP 2025, у системі застосовується алгоритм деривації ключа PBKDF2 з використанням геш-функції SHA-256,

кількістю ітерацій не менше 600 000 та унікальною випадковою сіллю для кожного користувача. Це робить атаку перебором обчислювально нерентабельною навіть на сучасному обладнанні. Безпосереднє шифрування контейнера виконується алгоритмом AES-GCM, який забезпечує не лише конфіденційність, але й цілісність зашифрованого ключа.

3.1.2 Розробка протоколу захищеного обміну даними

Після вирішення задачі управління ідентичністю, генерації та збереження ключів P-256, наступним кроком є проектування механізму транспортування повідомлень. Оскільки система проектується для роботи в реальному часі та передбачає групові комунікації, пряме застосування базових алгоритмів є недостатнім. Розробка протоколу відбувалася шляхом послідовного вирішення проблем масштабування та синхронізації.

У найпростішому сценарії комунікації “один-на-один”, наприклад користувач Б (Боб) хоче надіслати повідомлення користувачу А (Аліса), ми розглядаємо використання стандарту HPKE у його базовому режимі Base Mode. Цей стандарт не є монолітним алгоритмом, а являє собою конструкцію, що послідовно поєднує три криптографічні примітиви:

- KEM — механізм інкапсуляції ключа.
- KDF — функція деривації (виведення) ключів.
- AEAD — автентифіковане симетричне шифрування.

Логіка процесу формування захищеного повідомлення виглядає наступним чином:

1) Користувач Б, знаючи довгостроковий публічний ключ користувача А, генерує ефемерну (одноразову) пару ключів для цього конкретного повідомлення. Використовуючи свій ефемерний приватний ключ та публічний ключ користувача А, він виконує операцію Діффі-Хеллмана на еліптичній кривій ECDH для обчислення спільного секрету Shared Secret. Одночасно з цим формується інкапсульований ключ (enc) — це

фактично ефемерний публічний ключ користувача Б, який дозволить користувачу А пізніше відтворити ту саму операцію і отримати той самий спільний секрет.

$$enc, sharedSecret \leftarrow KEM(PK_A),$$

де *KEM* — функція інкапсуляції ключа;

PK_A — публічний ключ отримувача;

enc — інкапсульований ключ, публічна частина ефемерного ключа яку треба передати отримувачу;

sharedSecret — прихований спільний секрет, який не віддається напряму, а одразу переходить в KDF.

2) Отриманий спільний секрет сам по собі ще не є ключем шифрування. Він передається у функцію деривації, наприклад HKDF-SHA256. Ця функція "розгортає" секрет у криптографічний контекст (Context). Цей контекст містить набір параметрів, необхідних для симетричного шифрування, зокрема:
   - симетричний ключ для AEAD.
   - унікальне число (вектор ініціалізації), яке запобігає атакам повторення.

$$senderContext \leftarrow KDF,$$

де *KDF* — функція виведення ключів;

sernderContext — об'єкт який зберігає в всередині себе налаштовані ключі.

3) Використовуючи параметри з отриманого контексту, користувач Б шифрує текст повідомлення M та, опціонально, додаткові відкриті дані, наприклад заголовки. Результатом є шифротекст, який неможливо прочитати без контексту, і який захищений від будь-яких непомітних модифікацій (цілісність).

$$Ciphertext \leftarrow Context.Seal(M),$$

де *Context.Seal* — метод шифрування даних з контекстом відправника;

M — дані які треба шифрувати;

Ciphertext — результат шифрування, шифротекст.

4) Користувач Б відправляє користувачу А пару значень (enc, Ciphertext), яка дозволить відновити спільний секрет, контекст і зашифровані дані.

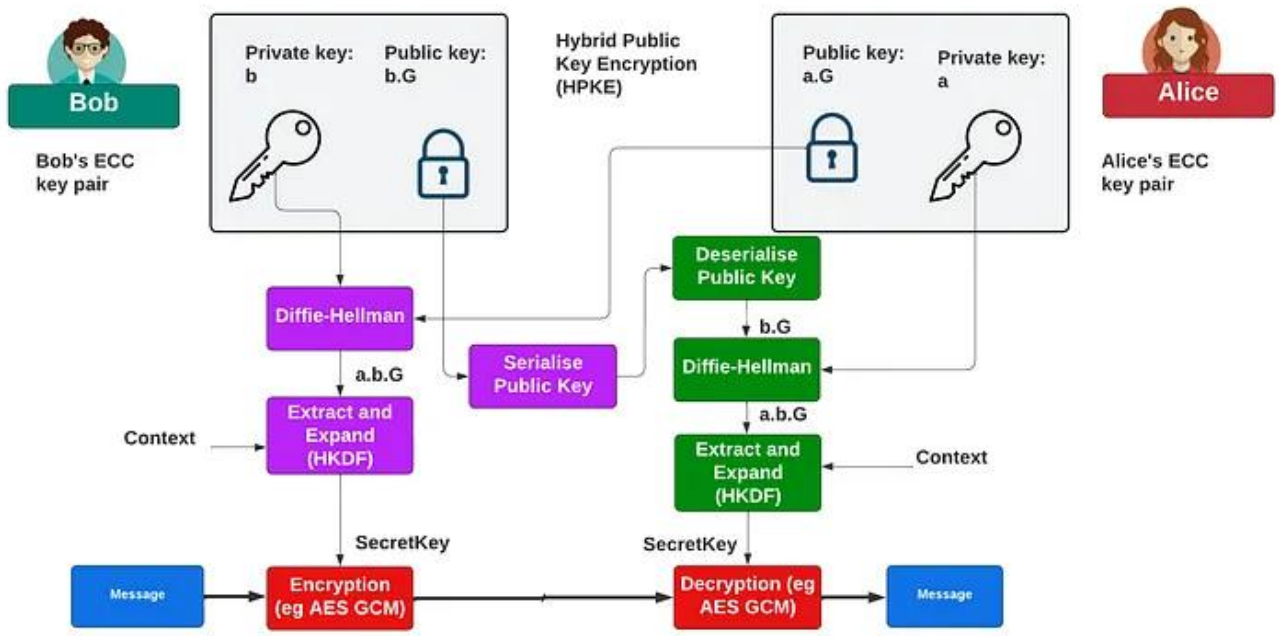


Рисунок 3.5 — Базова модель шифрування через HPKE

Хоча цей підхід є криптографічно стійким, він ефективний лише для двох учасників. Оскільки спільний секрет, а отже, і контекст, унікально прив’язаний до пари ключів учасників, користувач Б не може використати той самий шифротекст для третього учасника. Для кожного отримувача процедуру шифрування самого повідомлення, найважчу операцію, якщо повідомлення — це великий об’єм даних, довелося б виконувати заново. Це призводить до лінійного зростання навантаження O(N×Розмір даних) як за обчислювальними ресурсами, так і за обсягом трафіку, що робить цей метод непридатним для масових розсилок.

Гібридна схема

Для вирішення проблеми масштабування було застосовано патерн Digital Envelope. Ідея полягає у розділенні шифрування даних і шифрування ключів.

У цій схемі ми вводимо поняття симетричного сеансового ключа повідомлення. Алгоритм трансформується наступним чином:

1) Аліса генерує випадковий симетричний ключ K_msg, наприклад 32 байти для AES-256 та вектор ініціалізації IV.
2) Тіло повідомлення шифрується один раз цим ключем за допомогою швидкого симетричного алгоритму AES-GCM:

$$C_{payload} \leftarrow AES\_GCM(K_{msg}, IV, M),$$

де $C_{payload}$ — зашифрований блок даних;

AES_GCM — методи симетричного шифрування;

K_{msg} — тимчасовий симетричний ключ;

IV — вектор ініціалізації;

M — дані які шифрують.

3) Симетричний ключ K_msg шифрується (інкапсулюється) окремо для кожного учасника за допомогою їхніх публічних ключів через HPKE:
   - Для Боба HPKE.Seal(PK_Bob, K_msg)
   - Для Керол HPKE.Seal(PK_Carol, K_msg)
4) В результаті пакет містить список отримувачів та їх персональні зашифровані копії ключа і один великий зашифрований блок даних.

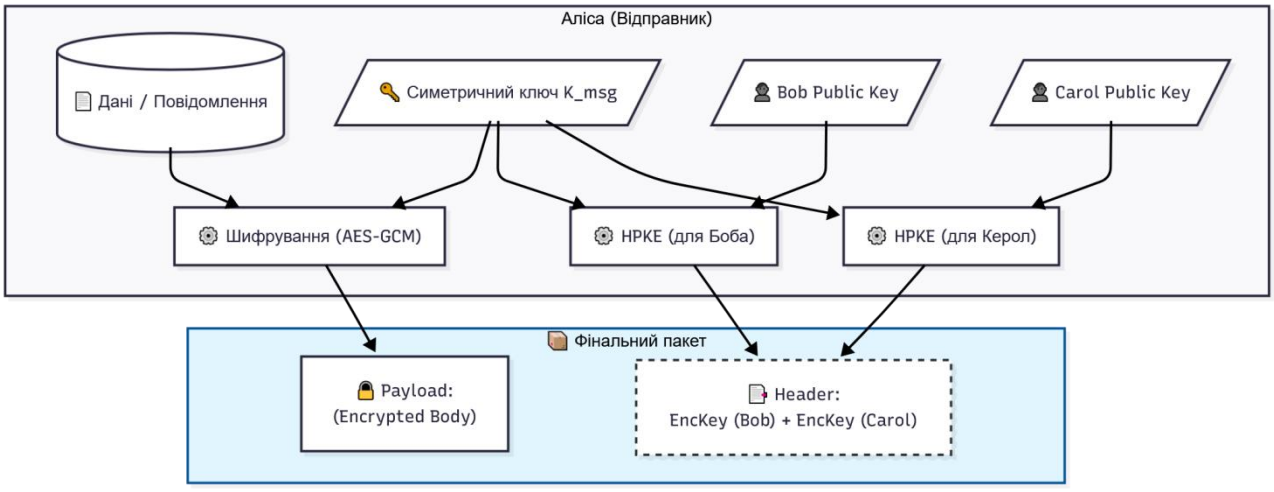


Рисунок 3.6 — Схема алгоритму гібридного шифрування з підтримкою множини отримувачів

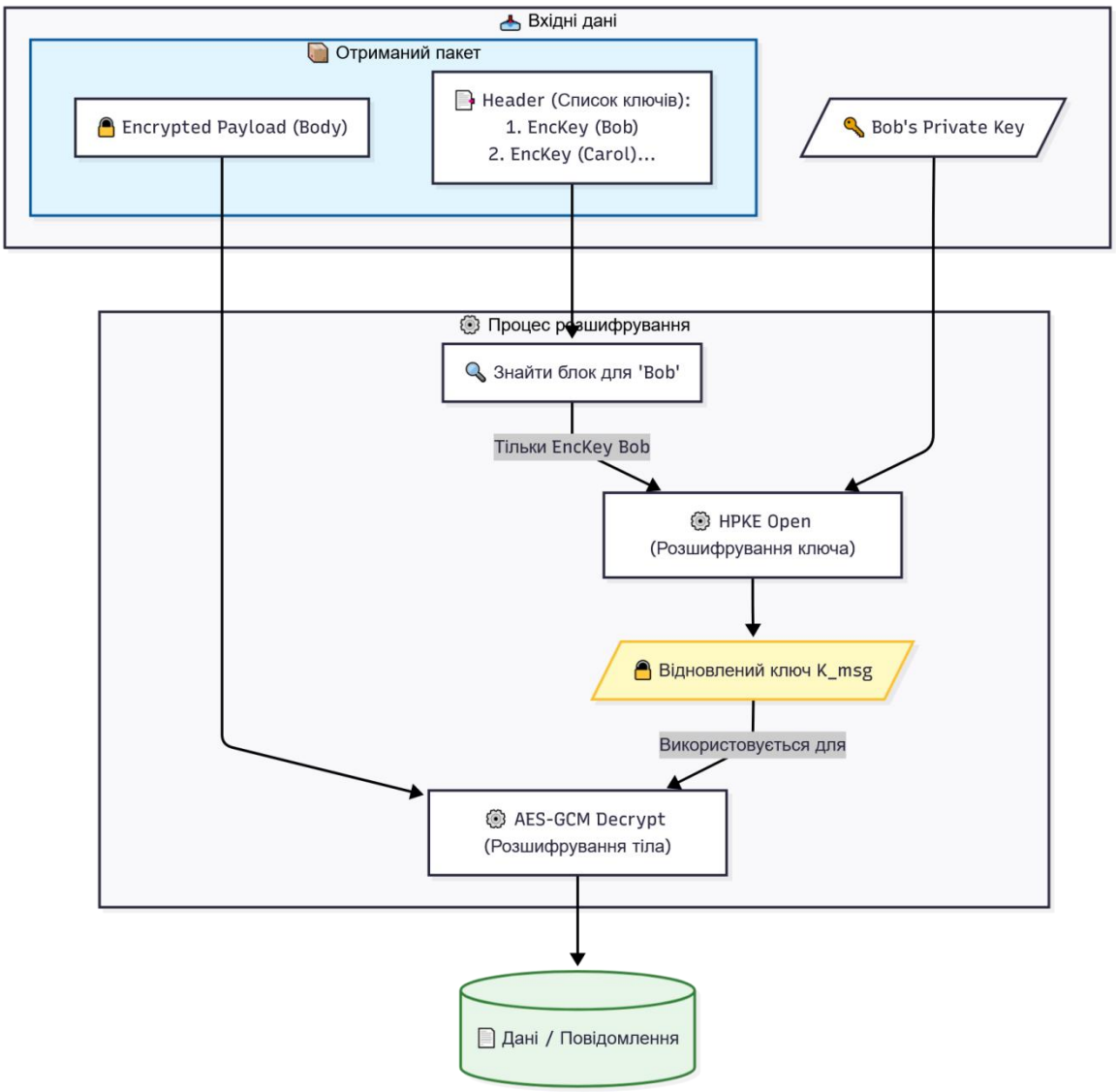


Рисунок 3.7 — Алгоритм відновлення сеансового ключа та дешифрування даних

Це дозволяє серверу зберігати лише одну копію зашифрованого файлу чи тексту, незалежно від кількості отримувачів, що економить дисковий простір та трафік.

Проблема Self-Decryption

Під час аналізу архітектури виникла специфічна проблема E2EE-систем пов’язана з розшифруванням для самого відправника, відома як Self-Decryption. У класичній клієнт-серверній архітектурі відправник бачить свої повідомлення,

просто завантажуючи їх з бази даних сервера. Однак у нашій моделі сервер зберігає лише зашифровані дані і ключі отримувачів.

Якщо користувач зайде в систему з іншого пристрою, він завантажить історію повідомлень, але не зможе розшифрувати власні повідомлення, оскільки:

- Він не має приватних ключів інших користувачів.
- Оригінальний симетричний ключі, яким він шифрував дані, був видалений з пам’яті після відправки.

Для вирішення проблеми синхронізації було прийнято рішення трактувати відправника як рівноправного отримувача повідомлення. Протокол був модифікований таким чином, що при формуванні списку отримувачів клієнтський додаток автоматично додає до нього публічний ключ самого відправника.

Таким чином, фінальний алгоритм формування захищеного повідомлення виглядає так:

1) Клієнт генерує ефемерний симетричний ключ K_msg.
2) Клієнт шифрує повідомлення за допомогою K_msg.
3) Клієнт формує список отримувачів який буде включати самого відправника.
4) Для кожного отримувача зі списку клієнт виконує HPKE-інкапсуляцію ключа K_msg.

В результаті, коли користувач відкриває історію чату на новому пристрої, додаток знаходить у заголовку повідомлення блок, зашифрований для його публічного ключа, використовує свій відновлений приватний ключ для отримання K_msg і за допомогою нього розшифровує текст повідомлення. Це забезпечує повну синхронізацію історії між усіма довіреними пристроями користувача без компрометації принципів E2EE.

3.1.3 Оптимізація обробки великих масивів даних

Одним із критичних викликів при реалізації наскрізного шифрування у веб-середовищі є обмеження ресурсів клієнтського пристрою, зокрема оперативної

пам’яті. Стандартна реалізація методів Web Cryptography API, наприклад window.crypto.subtle.encrypt, працює за принципом “все або нічого”: для виконання операції весь масив даних має бути завантажений у пам’ять у вигляді об’єкта ArrayBuffer.

При спробі зашифрувати файл значного обсягу, наприклад файл розміром 2 ГБ, одним викликом функції виникають наступні проблеми:

- Обмеження двигуна JavaScript, V8 у Chrome або SpiderMonkey у Firefox, не дозволяють виділяти настільки великі безперервні блоки пам’яті, що призводить до аварійного завершення роботи вкладки.
- Навіть якщо пам’яті вистачає, синхронна підготовка такого масиву даних блокує інтерфейс користувача.
- Обмеження алгоритму AES-GCM, адже згідно зі специфікацією NIST SP 800-38D, існує ліміт на кількість блоків, зашифрованих одним ключем та вектором ініціалізації IV, перевищення якого знижує криптографічну стійкість.

Для вирішення цих проблем було розроблено алгоритм потокового шифрування, який розширює спосіб використання алгоритму гібридного шифрування.

Суть методу полягає у віртуальному розділенні вхідного потоку даних на послідовність сегментів (чанків) фіксованого розміру. Кожен сегмент можна сприймати як окреме повідомлення яке надсилає користувач, що не вимагає зміни алгоритму гібридного шифрування.

Ключовою вимогою безпеки при шифруванні потоку чанків з одним симетричним ключем є унікальність пари Key і IV для кожного сегмента. Оскільки ключ сесії залишається незмінним для всього масиву даних, унікальність має забезпечуватися варіацією вектора ініціалізації.

Розроблений алгоритм працює наступним чином:

- Масив даних зчитується послідовно, сегмент за сегментом.
- Для кожного сегмента обчислюється унікальний вектор ініціалізації.

- Кожен сегмент шифрується незалежно з використанням алгоритму AES-GCM, утворюючи зашифрований сегмент.
- Отримані зашифровані сегменти послідовно передаються на сервер.

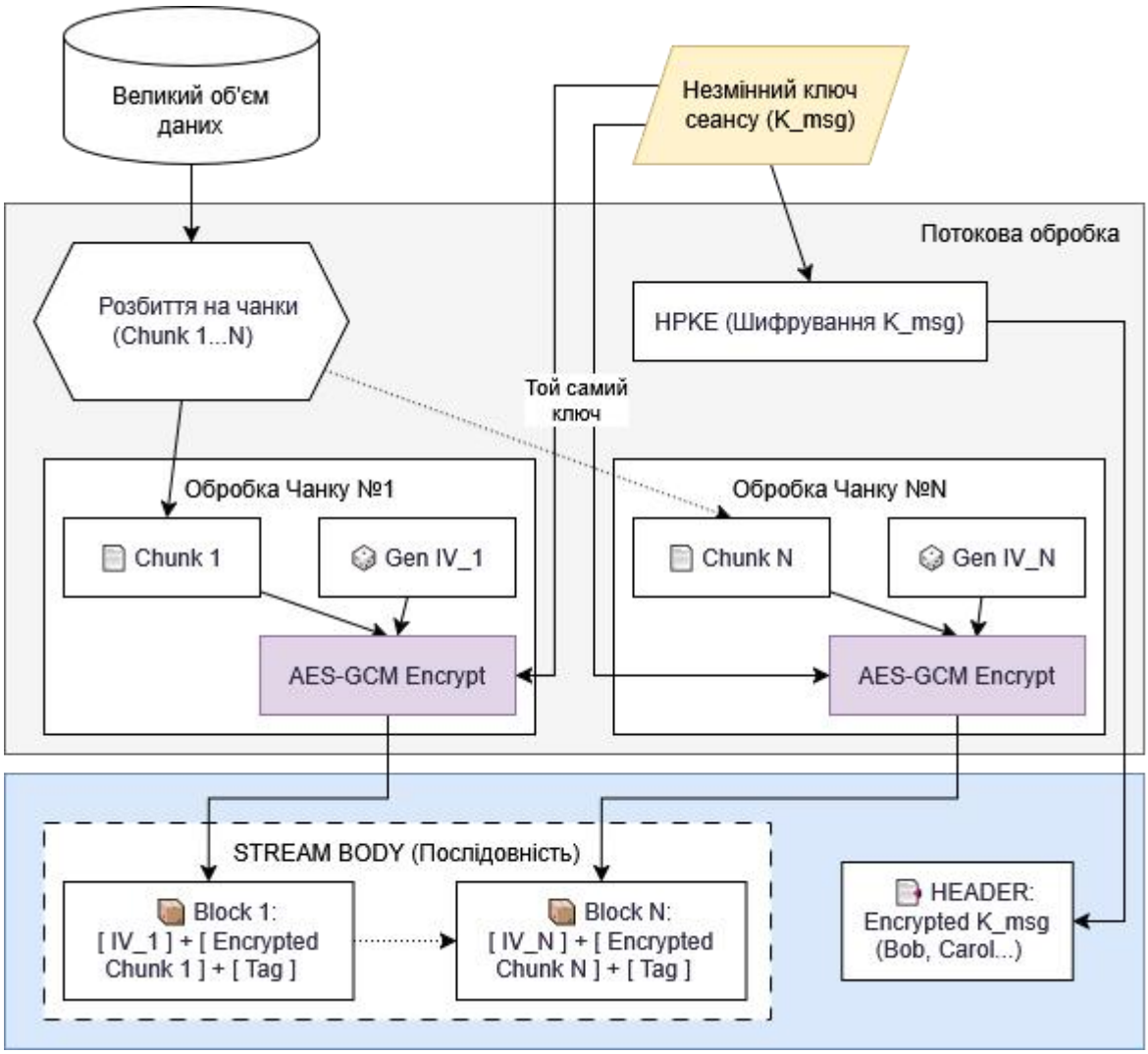


Рисунок 3.8 — Алгоритм потокового шифрування

Такий підхід дозволяє обробляти масиви даних будь-якого розміру використовуючи вже розроблене гібридне шифрування і не змінюючи його архітектуру. Незалежно від розміру даних, споживання оперативної пам’яті браузера залишається стабільним і дорівнює розміру одного чанка плюс службові витрати. Використання сегментів дає можливість вибирати необхідні частинки для перегляду без потреби розшифровувати весь масив даних. Також, з’являється стійкість до розривів з’єднання, адже процес завантаження або вивантаження

може бути відновлений з останнього успішно обробленого сегмента без необхідності повторного шифрування всього масиву даних.

3.2 Адаптація архітектури шифрування в веб-середовищі і модель даних

3.2.1 Загальна архітектура клієнт-серверної взаємодії

Для впровадження механізмів шифрування і їх ефективного використання в веб-системах необхідно забезпечити надійність, масштабованість та легку підтримку коду. Тому, архітектура програмного комплексу спроектована на основі принципів слабкої з’язності компонентів Loose Coupling, чіткого розмежування відповідальності Separation of Concerns та розшарування Layered Architecture. Такий підхід дозволяє ізолювати логіку шифрування від інтерфейсу користувача та відокремити бізнес-правила від деталей реалізації.

Архітектура клієнтської частини

Архітектура клієнтської частини базується на моделі односторінкового додатку Single Page Application, що дозволяє забезпечити динамічне оновлення контенту без перезавантаження сторінки. Для організації кодової бази та забезпечення її масштабованості обрано компонентно-орієнтований підхід із чітким розділенням на логічні шари. Така структура гарантує, що зміни в інтерфейсі не впливатимуть на бізнес-логіку або ядро шифрування.

Загальну схему взаємодії компонентів та напрямки потоків даних наведено на рисунку 3.9. На схеми видно, що архітектура побудована за радіальним принципом, де центральним вузлом виступає шар бізнес-логіки та управління станом. Він діє як диспетчер, координуючи обмін інформацією між інтерфейсом користувача, мережевим рівнем та ізольованими сервісами безпеки. Такий підхід дозволяє абстрагувати візуальну частину від процесів передачі та шифрування даних, забезпечуючи модульність системи.

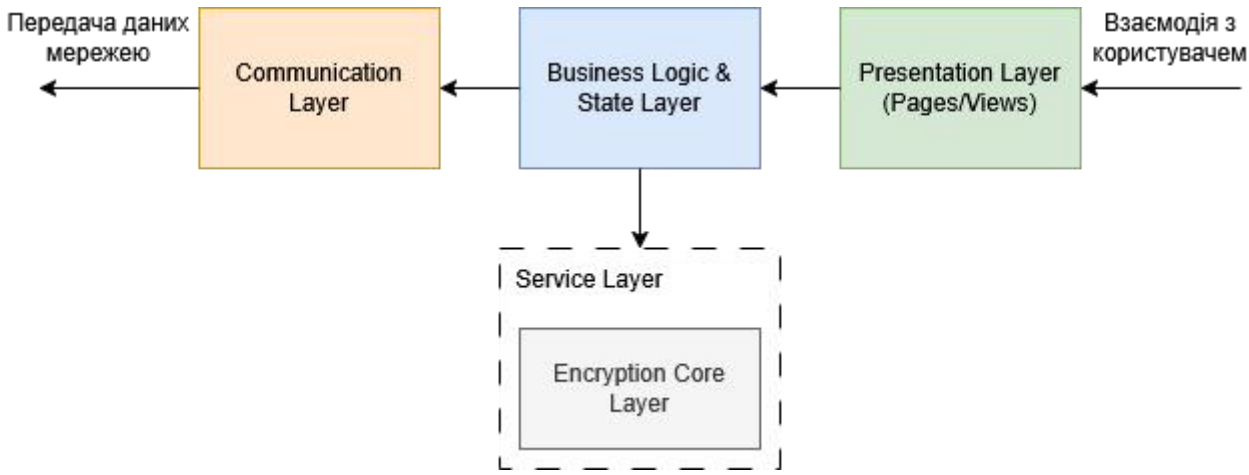


Рисунок 3.9 — Архітектура клієнтської частини

Шар представлення відповідає за візуалізацію даних та інтерфейс взаємодії з користувачем. Його головна задача — відображати стан системи та транслювати дії користувача на нижчі рівні, не містячи при цьому складної бізнес-логіки. В основі цього шару лежить ієрархічна система компонентів, що базується на структурі компонентів Vue.js.

На верхньому рівні ієрархії знаходяться Layouts макети, компоненти-обгортки. Вони визначають спільну структуру для групи сторінок, наприклад навігаційна панель або бокове меню чатів, і містять спеціальні області — слоти <slot> для ін’єкції контенту.

Слот виступає контейнером, у який динамічно підставляється вміст конкретної сторінки залежно від активного маршруту. Самі сторінки, в свою чергу, компонуються з менших, ізольованих UI-компонентів, такі як кнопки, форми вводу, картки повідомлень, що забезпечує високий рівень перевикористання коду.

Шар бізнес-логіки та управління станом виступає посередником між інтерфейсом та даними, інкапсулюючи правила роботи додатку. Він відповідає за реактивність даних та їх синхронізацію між різними частинами інтерфейсу.

Механізм управління станом вирішує проблему зберігання даних авторизованого користувача і відповідних йому пари ключів. Для

централізованого управління станом додатку використовується бібліотека Pinia. Stores Pinia дозволяють уникнути проблеми складної передачі даних через пропси, так звані prop drilling, і надають єдине джерело правди для всіх компонентів. Сам store — це об’єкт, що містить стан і бізнес-логіку, які не прив’язані до дерева компонентів. Це компонент, який завжди присутній і який кожен може читати та записувати. Він має три поняття: стан, геттери та дії, і ці поняття є еквівалентами даних, обчислень та методів у компонентах.

Щоб визначити store, ми використовуємо функцію defineStore() і передаємо першим аргументом унікальне ім’я:

```
import { defineStore } from 'pinia'
export const useAuthStore = defineStore('auth', {
  // other options...
})

// Виклик стору і використання
import { useAuthStore } from '@/stores/auth.ts'
import { storeToRefs } from 'pinia'
const authStore = useAuthStore()
const { user } = storeToRefs(authStore)
const { restoreSession, register, login, logout } = authStore
```

Для інкапсуляції та перевикористання логіки застосовано підхід Composition API. Специфічні алгоритми та бізнес-сценарії виносяться з компонентів у окремі модулі composables. Це дозволяє, наприклад огорнути наш auth store в composable функцію і використовувати її повторно в різних частинах системи, залишаючи код компонентів чистим та зосередженим лише на відображенні.

Шар ядра шифрування є ізольованим модулем, що відповідає за всі криптографічні перетворення і містить імплементацію всіх криптографічних протоколів. Він спроектований таким чином, щоб бути незалежним від фреймворку інтерфейсу. Шар надає чіткий API для вищих рівнів як сервіс, який

приймає відкриті дані та ключі, а повертає зашифровані контейнери або навпаки. Цей шар абстрагує складність роботи з Web Cryptography API та алгоритмами HPKE/AES від решти додатку, що дозволяє змінювати криптографічну реалізацію без впливу на інтерфейс користувача і гарантує, що криптографічні операції виконуються коректно та безпечно, незалежно від того, яка частина інтерфейсу їх викликає.

Шар комунікації, як нижній рівень архітектури, що відповідає за фізичний обмін даними із зовнішнім світом. Він абстрагує деталі мережевих протоколів від решти додатку і слугує адаптером між внутрішніми структурами даних додатку та зовнішнім API сервера. Реалізація цього рівня базується на HTTP-клієнті для REST-запитів та WebSocket-клієнті для обміну подіями в реальному часі. Його функція полягає у формуванні запитів, додаванні необхідних заголовків безпеки та первинній обробці відповідей сервера перед передачею даних на рівень бізнес-логіки.

Архітектура серверної частини

Архітектура серверної частини базується на платформі Node.js, що забезпечує єдиний мовний контекст TypeScript з клієнтською частиною. Архітектура сервера побудована за принципом інверсії управління Inversion of Control з широким використанням патерну впровадження залежностей Dependency Injection. Це дозволяє створювати слабкозв’язані компоненти, які легко тестувати та масштабувати.

Структура серверу також розділена на чіткі логічні шари, кожен з яких виконує свою специфічну функцію в обробці запиту. Загальну організацію потоків даних та ієрархію компонентів наведено на рисунку 3.10. Схема ілюструє трирівневу архітектуру, де обробка запиту відбувається у вигляді лінійного, односпрямованого процесу. Потік управління передається від вхідної точки Controller Layer через шар бізнес-логіки Service Layer до рівня доступу до даних Repository Layer. Такий послідовний конвеєр забезпечує чіткий розподіл

відповідальності Separation of Concerns, де кожен шар ізольований і виконує лише специфічний набір операцій перед передачею даних далі.

Рисунок 3.10 — Архітектура серверної частини

Шар взаємодії з клієнтом виступає вхідною точкою Entry Point для всіх зовнішніх запитів. Контролери відповідають за маршрутизацію routing HTTP-запитів та подій WebSocket. Їхня основна функція — диригувати процесом обробки запиту: прийняти дані, провести їх первинну валідацію відповідності типам та схемам, передати їх на обробку в шар бізнес-логіки та сформувати стандартизовану відповідь для клієнта. Контролери не містять складної логіки обробки даних, виступаючи виключно як інтерфейс взаємодії.

Шар бізнес-логіки виступає серцевиною серверного додатку, де реалізовані основні алгоритми та правила системи. Сервіси реалізують конкретні функціональні вимоги, наприклад AuthService інкапсулює логіку авторизації користувача. Вони можуть взаємодіяти з іншими сервісами або звертатися до шару даних. Завдяки DI, сервіси отримують необхідні залежності через конструктор, що дозволяє легко підміняти реалізації, наприклад для тестування.

Шар доступу до даних відповідає за абстракцію роботи з базою даних. Репозиторії надають методи для виконання CRUD-операцій (Create, Read, Update, Delete), приховуючи деталі реалізації запитів. Для взаємодії з реляційною базою даних використовується ORM. Такий підхід дозволяє працювати з даними як з типізованими об’єктами, уникаючи написання SQL-коду в бізнес-логіці та забезпечуючи захист від SQL-ін’єкцій на рівні інфраструктури.

Організація взаємодії між бізнес-логікою та шаром даних реалізована за принципом суворої інкапсуляції в межах функціональних контекстів, де сервіси мають прямий доступ виключно до репозиторіїв своєї зони відповідальності.

Потреба в отриманні даних із суміжних модулів задовольняється через горизонтальну комунікацію між відповідними сервісами, тоді як перехресне звернення до репозиторіїв інших компонентів прямо заборонено архітектурою для уникнення сильної зв’язності та забезпечення централізованої обробки бізнес-правил.

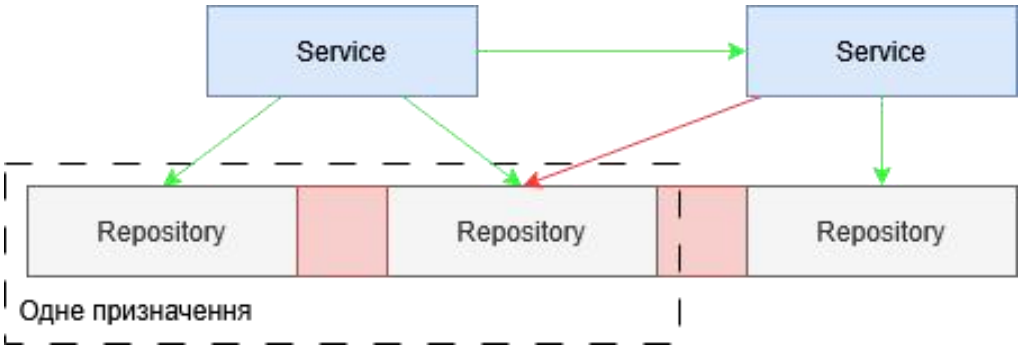


Рисунок 3.11 — Взаємодія сервісів і репозиторіїв

Для забезпечення структурованості коду застосовано модульний підхід, де кожен логічний блок об’єднує пов’язані контролери, сервіси та репозиторії. Модулі відповідають за налаштування контексту виконання та ініціалізацію своїх компонентів, надаючи зовнішньому світу лише необхідний публічний інтерфейс.

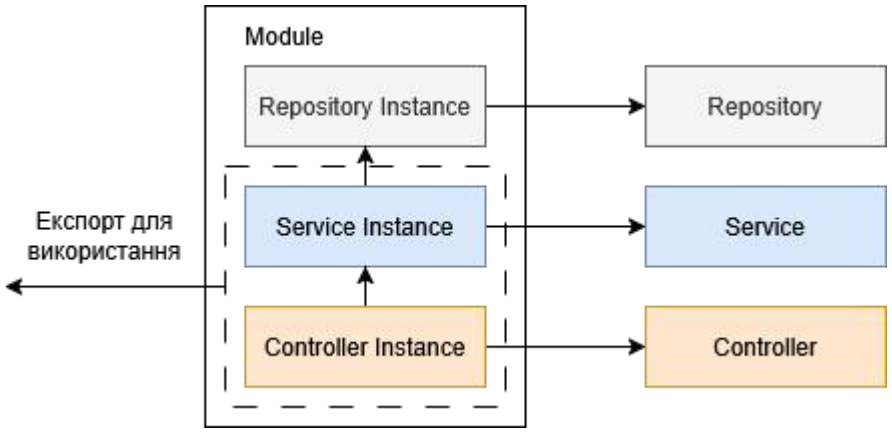


Рисунок 3.12 — Модульна організація

Важливо зазначити, що у спроектованій архітектурі сервер виконує роль розпорядника інформації, а не учасника комунікації. Серверна логіка відповідає за

автентифікацію користувачів, збереження та синхронізацію даних, маршрутизацію повідомлень, але не бере участі в криптографічних операціях з контентом. Сервер оперує виключно зашифрованими контейнерами і не володіє ключами для їх розшифрування. Така модель мінімізує ризики витоку конфіденційної інформації навіть у випадку повної компрометації серверної інфраструктури, адже сервер технічно не має можливості виступати в ролі клієнта, що здатен розшифрувати повідомлення.

Механізми авторизації та безпеки сесій

Ключовим елементом архітектури є система ідентифікації та управління сесіями користувачів. Оскільки додаток оперує конфіденційними даними, навіть у зашифрованому вигляді, до механізмів авторизації висуваються підвищені вимоги безпеки.

Для реалізації stateless авторизації обрано стандарт JSON Web Tokens. Схема автентифікації базується на використанні пари токенів, що дозволяє збалансувати безпеку та зручність користування:

- Короткостроковий Access Token, час життя якого від 15 до 30 хвилин, що містить ідентифікатор користувача. Він додається до заголовків кожного HTTP-запиту та використовується сервером для перевірки прав доступу до ресурсів API. Короткий час життя мінімізує ризики у випадку викрадення токена.
- Довгостроковий Refresh Token, час життя якого від 7 до 30 днів. Він використовується виключно для отримання нової пари токенів без необхідності повторного введення пароля користувачем.

Критичним архітектурним рішенням є відмова від зберігання токенів у localStorage або sessionStorage браузера, оскільки ці сховища вразливі до атак типу XSS. Будь-який шкідливий JavaScript-код, впроваджений на сторінку, може отримати доступ до локального сховища і викрасти сесію.

Натомість, у розробленій архітектурі системи використовується механізм транспортування токенів через HttpOnly Cookies. Цей прапор забороняє доступ до cookies з боку клієнтських скриптів, запобігаючи їх викраденню зловмисниками навіть у випадку наявності вразливостей у коді.

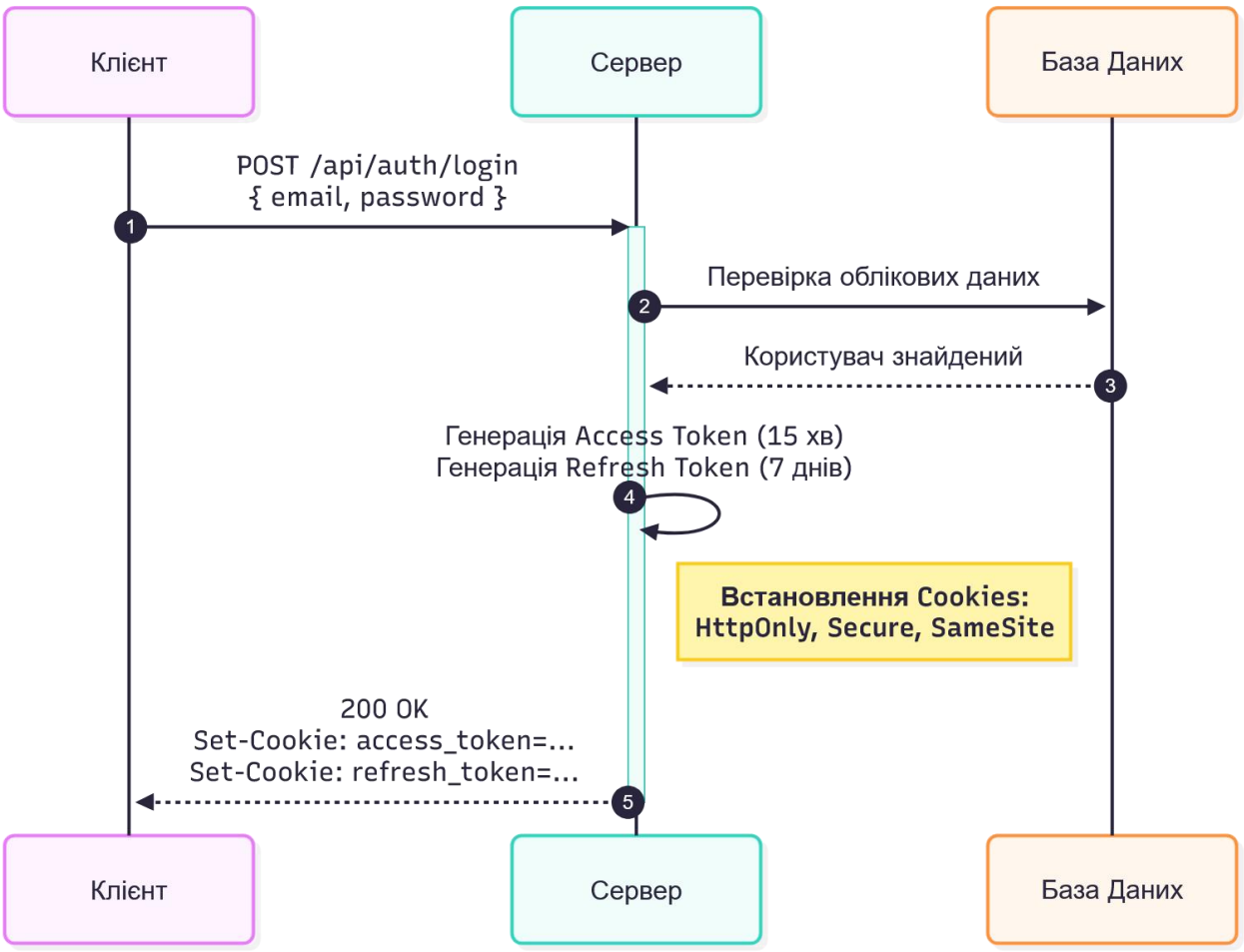


Рисунок 3.13 — Автентифікація з JWT і HttpOnly Cookies

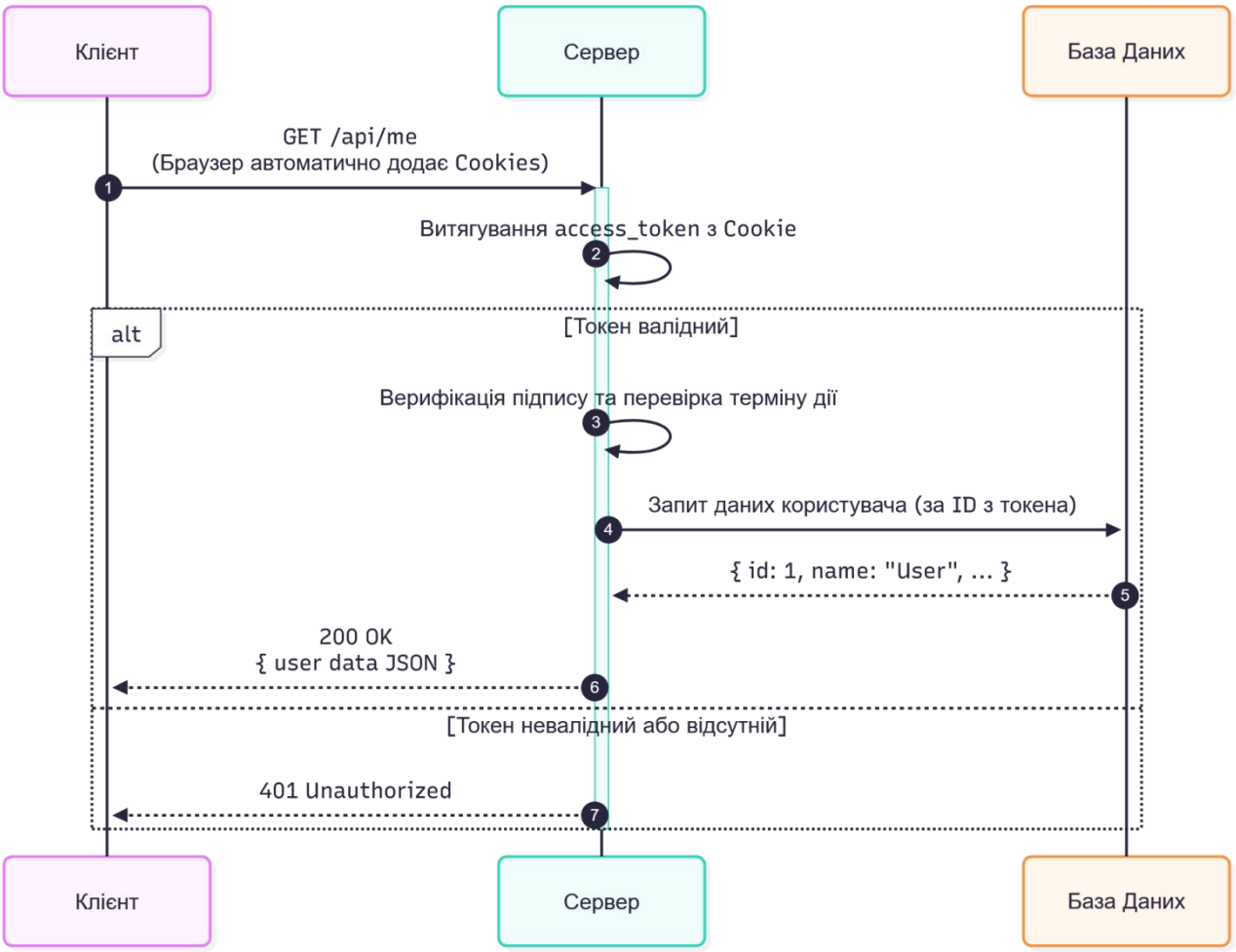


Рисунок 3.14 — Запит захищених даних з JWT і HttpOnly Cookies

Обмін даним та стандартизація взаємодії

Ефективна робота системи неможлива без чітко визначених контрактів обміну даними. У розробленій архітектурі, незважаючи на недовіру до сервера в питаннях конфіденційності контенту, клієнт розглядає сервер як “єдине джерело правди” у питаннях стану системи. Стабільність роботи сервера є критичною умовою функціонування всього комплексу, оскільки його відмова блокує можливість обміну ключами та маршрутизацію трафіку.

Основним форматом обміну даними між клієнтом та сервером обрано JSON. Цей вибір обумовлений його нативною підтримкою в середовищі JavaScript/TypeScript, як на Node.js, так і в браузері.

Однак використання JSON створює технічний виклик для криптографічної системи: результати роботи алгоритмів шифрування є бінарними даними, масиви байтів Uint8Array або ArrayBuffer, які не підтримуються стандартом JSON напряму. Для вирішення цієї проблеми впроваджено шар адаптації даних: перед відправкою на сервер усі бінарні криптографічні артефакти кодуються у рядковий формат Base64. При отриманні даних клієнт виконує зворотне декодування. Такий підхід гарантує цілісність байт-коду при проходженні через протоколи HTTP/WebSocket.

Серверна частина реалізує сувору політику “нульової довіри” до вхідних даних. Усі вхідні запити проходять через шар валідації, де перевіряється:

- Структурна запиту і його відповідність.
- Типізація даних, чи є поле keys масивом, а message рядком.
- Логічна коректність, наприклад чи правильно довжина ключа.

Тільки після успішної валідації дані передаються на обробку. Відповіді сервера також проходять процедуру мапингу в об’єкти DTO. Це гарантує, що клієнт отримує дані у суворо визначеному форматі, що спрощує логіку та усуває необхідність додаткових перевірок існування полів на клієнті.

Обов’язково треба врахувати, що механізм обробки відрізняється залежно від протоколу взаємодії, HTTP або WebSocket, проте слідує єдиному алгоритму Валідація → Обробка → Відповідь.

Обробка запитів REST API (HTTP) побудована на базі патерну Middleware Pipeline. Вхідний запит проходить через шар валідації Middleware/Pipe перед тим, як потрапити до бізнес-логіки в Controller. Цей шар перевіряє наявність обов’язкових полів, відповідність типів даних та коректність форматів.

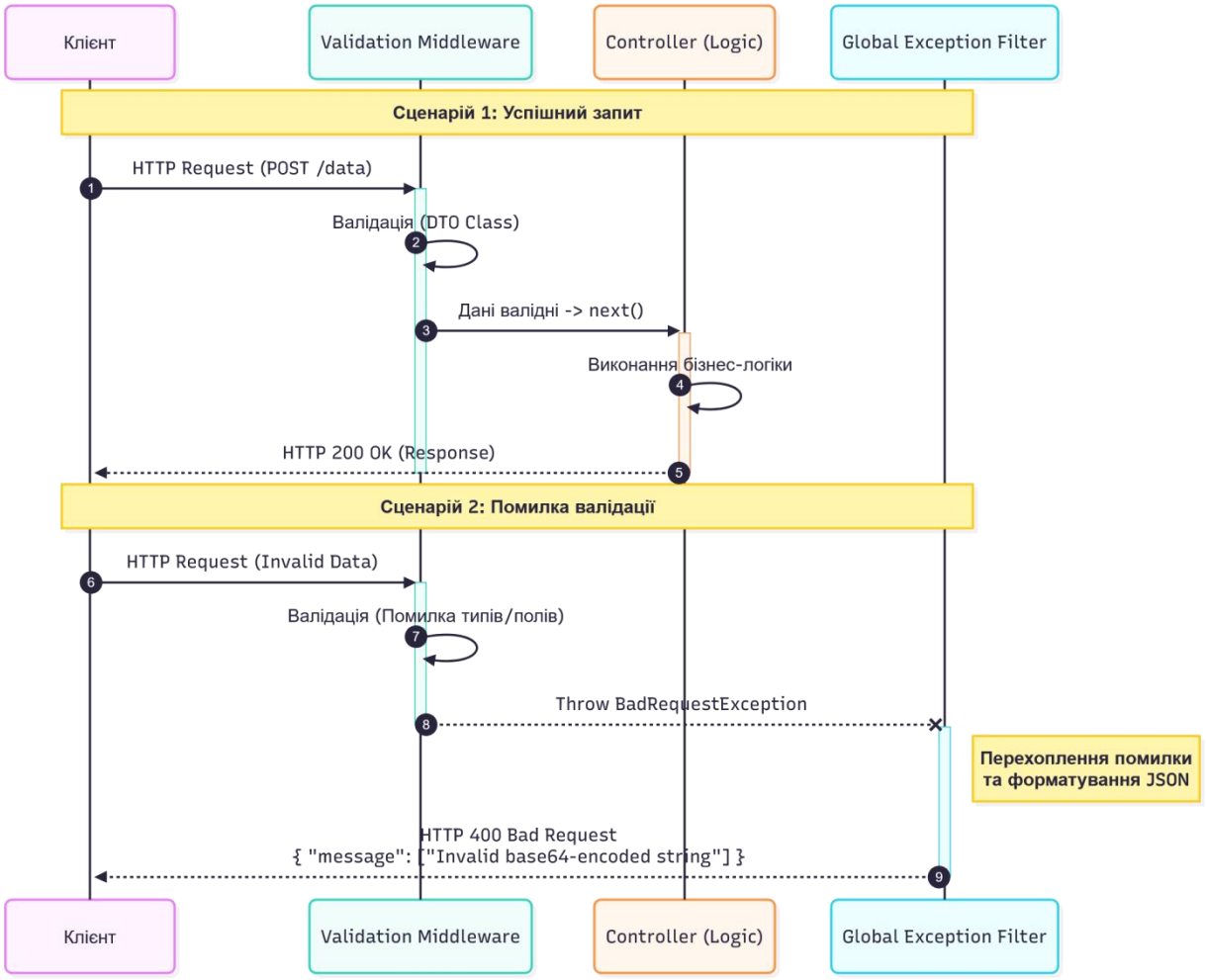


Рисунок 3.15 — Обробка запитів в REST API

При успішної валідації отриманого запиту HTTP управління передається контролеру, який виконує визначену бізнес-логіку і повертає успішний статус виконання.

У разі помилки валідації отриманого запиту процес переривається, генерується виключення Exception, яке перехоплюється глобальним фільтром помилок Global Exception Filter. Цей фільтр формує уніфіковану HTTP-відповідь, наприклад статус 400 Bad Request, з деталями помилки для клієнта.

Для WebSocket принцип обробки запитів відрізняється. Оскільки WebSockets працюють на основі подій Event-driven, а не класичного циклу запит-відповідь, використання стандартних HTTP middleware є неможливим. Для цього застосовано патерн Wrapper. Кожен обробник події обгортається у функцію вищого порядку, яка виступає гарантом валідності даних. Функція-обгортка

перехоплює вхідні дані події payload. Якщо дані невалідні, обгортка самостійно ловить помилку і, замість розриву з'єднання, генерує спеціальну подію exception для конкретного клієнта.

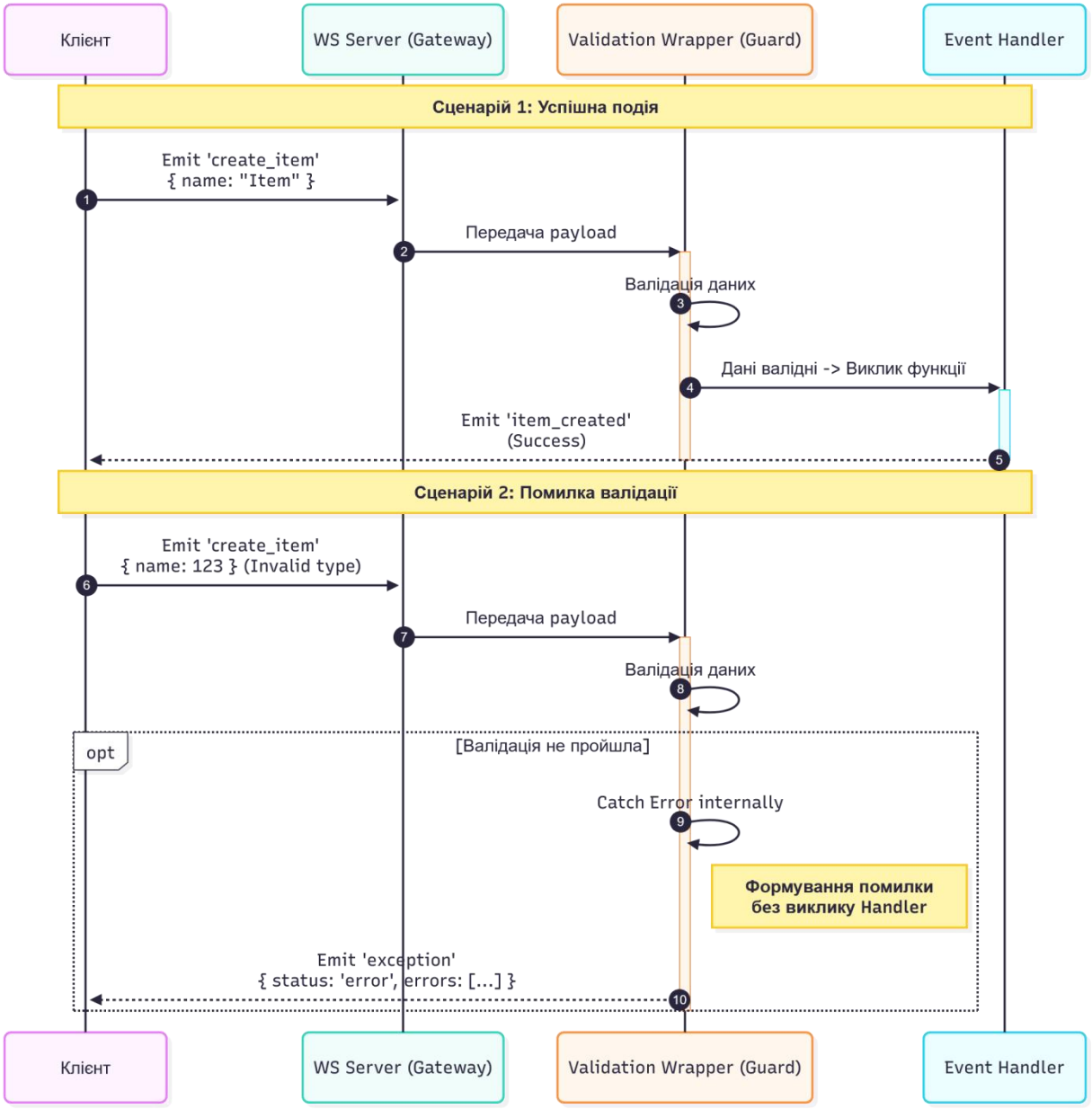


Рисунок 3.16 — Обробка запитів в WebSocket

При виникненні помилок, важливо не тільки обробити їх, а й стандартизувати. Для забезпечення передбачуваної поведінки системи розроблено уніфікований механізм обробки виключних ситуацій. Сервер не просто повертає

HTTP-статуси або зміст помилки, він надає типізовану структуру помилки, що містить:

- Код помилки як унікальний рядковий ідентифікатор, наприклад VALIDATION_ERROR або BAD_REQUEST.
- Текст помилки, який містить короткий опис для відлагодження.
- Деталі помилки (за наявності), може містити детальний зміст помилки.

Клієнтський додаток вміє працювати з такими помилками знаючи їх код. Отримавши код помилки, клієнт автоматично розуміє в чому проблема і, що необхідно робити, замість того, щоб просто показувати користувачеві повідомлення “Щось пішло не так”. Такий підхід дозволяє автоматизувати відновлення роботи системи після збоїв.

3.2.2 Проектування бази даних та схеми зберігання криптографічних артефактів

Центральним елементом серверної інфраструктури є база даних, яка забезпечує персистентне зберігання стану системи. У контексті розробки захищеної системи обміну повідомленнями до бази даних висуваються специфічні вимоги. Вона повинна виступати надійним сховищем не лише для реляційних зв’язків, але й для криптографічних артефактів: ключів, шифротекстів, векторів ініціалізації.

Ключовою є вимога до формату зберігання даних. Криптографічні алгоритми HPKE і AES-GCM оперують бінарними даними — масивами байтів. Хоча сучасні СУБД підтримують бінарні типи даних, як BYTEA у PostgreSQL, для забезпечення сумісності з текстовим форматом обміну JSON, описаним у попередньому підрозділі, було прийнято рішення зберігати криптографічні дані у текстовому форматі Base64. Це рішення дозволяє:

- Уніфікувати представлення даних на клієнті, сервері та в базі даних.
- Спростити налагодження та логування, адже текстові рядки легше аналізувати, ніж бінарні дампи.

- Зменшити ризики спотворення даних при транскодуванні між різними шарами додатку.

В якості системи управління базами даних обрано об’єктно-реляційну СУБД PostgreSQL. Її вибір обумовлений високою надійністю, підтримкою ACID-транзакцій та суворою типізацією.

На низькому рівні робота з PostgreSQL передбачає використання мови запитів SQL. Створення бази даних та таблиць вимагає написання DDL-інструкцій (CREATE TABLE), а маніпуляція даними — DML-інструкцій (INSERT, SELECT).

Наприклад, для отримання ключів користувача необхідно було б сформувати запит вигляду: SELECT public_key, encrypted_private_key FROM users WHERE id = $1; Хоча цей підхід надає повний контроль, ручне написання SQL-запитів у великому проекті на TypeScript підвищує ризик помилок та ускладнює підтримку коду при зміні схеми даних.

Для вирішення проблем ручного управління SQL-запитами та забезпечення type safety на рівні доступу до даних, у проект інтегровано Prisma ORM. Це інструмент нового покоління, який дозволяє описувати схему бази даних у декларативному файлі schema.prisma та автоматично генерує на його основі строго типізований клієнт для TypeScript. Використання Prisma дозволяє:

- Автоматизувати процес міграцій змін у схемі БД.
- Виконувати запити, використовуючи методи об’єктів, що виключає синтаксичні помилки SQL, наприклад prisma.user.findUnique.
- Автоматично керувати зв’язками між таблицями, relations.

Аналіз спроектованої схеми даних

Для забезпечення функціонування системи захищеного обміну повідомленнями було розроблено реляційну схему даних, яка включає моделі для користувачів, чатів, повідомлень та криптографічних ключів. Нижче наведено

детальний опис кожної моделі та її полів відповідно до розробленої схеми schema.prisma

Модель користувача User є фундаментом системи. Вона об’єднує дані для автентифікації та криптографічні параметри для реалізації наскрізного шифрування.

Таблиця 3.1 — Опис моделі User

Назва поля	Тип	Зміст
id	uuid	Унікальний ідентифікатор Primary Key, використовується тип uuid для забезпечення глобальної унікальності та безпеки (неможливість передбачити id іншого користувача перебором)
username	text	Унікальні поля для ідентифікації користувача в системі
email		
hashedPassword	text	Геш пароля, отриманий за допомогою алгоритму bcrypt; оригінальний пароль ніколи не зберігається
avatarName	text	Посилання на файл аватара, якщо є
Криптографічні поля		
publicKey	text	Публічний ключ користувача, стандарт HPKE, крива P-256, закодований у base64; цей ключ є відкритим для всіх інших учасників і використовується ними для шифрування ключів повідомлень на адресу цього користувача
encryptedPrivateKey	text	Контейнер, що містить приватний ключ користувача, зашифрований симетричним ключем, похідним від пароля і зберігається у форматі base64
kdfSalt	text	Випадкова сіль, яка використовувалася в алгоритмі PBKDF2 при генерації ключа шифрування для encryptedPrivateKey, необхідна для відтворення процесу деривації на клієнті
encryptionIV	text	Вектор ініціалізації, який використовувався алгоритмом AES-GCM при шифруванні приватного ключа, необхідний для коректного розшифрування контейнера

Модель чату Chat представляє собою сутність, що групує повідомлення та учасників.

Таблиця 3.2 — Опис моделі Chat

Назва поля	Тип	Зміст
id	uuid	Унікальний ідентифікатор чату
type	ChatType	Тип чату, визначений через enum ChatType, може набувати значень DIRECT або GROUP
name	text	Назва чату, опціонально, актуально переважно для спільних чатів

Модель учасника чату ChatParticipant є проміжною таблицею для реалізації зв’язку “багато-до-багатьох” між користувачами та чатами. Це дозволяє гнучко додавати учасників та масштабувати систему.

Таблиця 3.3 — Опис моделі ChatParticipant

Назва поля	Тип	Зміст
chatId	uuid	Зовнішній ключ, що вказує на чат
userId	uuid	Зовнішній ключ, що вказує на користувача
@@unique([userId, chatId])	text	Композитний ключ який гарантує, що користувач не може бути доданий в один і той самий чат двічі

Модель повідомлення Message зберігає зашифрований контент та метадані повідомлення. Важливо, що сервер не має доступу до відкритого тексту.

Таблиця 3.4 — Опис моделі Message

Назва поля	Тип	Зміст
id	uuid	Унікальний ідентифікатор повідомлення
chatId	uuid	Посилання на чат, до якого належить повідомлення
senderId	uuid	Посилання на автора повідомлення

Продовження таблиці 4

type	MessageType	Тип контенту, визначений через enum MessageType, може набувати значень TEXT або FILE, визначає, як клієнт має інтерпретувати розшифровані дані
Криптографічні поля		
encryptedContent	text	Основне тіло повідомлення, шифротекст отриманий в результаті роботи алгоритму AES-GCM над повідомлення, зберігається як рядок base64
encryptionIV	text	Унікальний вектор ініціалізації, згенерований випадковим чином саме для цього повідомлення, необхідний для розшифрування encryptedContent

Модель ключів повідомлення MessageKey — це найважливіший елемент для реалізації схеми Digital Envelope. Оскільки саме повідомлення шифрується один раз симетричним ключем, цей ключ потрібно передати кожному учаснику. Таблиця MessageKey містить персональні конверти для кожного отримувача, включаючи відправника.

Таблиця 3.5 — Опис моделі MessageKey

Назва поля	**Тип**	**Зміст**
messageId	uuid	Посилання на повідомлення
recipientId	uuid	Посилання на користувача, для якого призначений цей конкретний запис ключа
Криптографічні поля		
encryptedKey	text	Симетричний ключ, яким зашифровано encryptedContent у таблиці Message, інкапсульований (зашифрований) публічним ключем отримувача recipientId з використанням стандарту HPKE
encapsulationData	text	Ефемерний публічний ключ, згенерований відправником під час процесу інкапсуляції HPKE; цей параметр є обов’язковим для того, щоб отримувач міг узгодити спільний секрет і розшифрувати encryptedKey

Міграція та візуалізація схеми

На основі описаної схеми в schema.prisma за допомогою команди prisma migrate генеруються SQL-міграції, які створюють відповідні таблиці, індекси та зовнішні ключі у PostgreSQL. Це гарантує повну відповідність структури бази даних коду додатку.

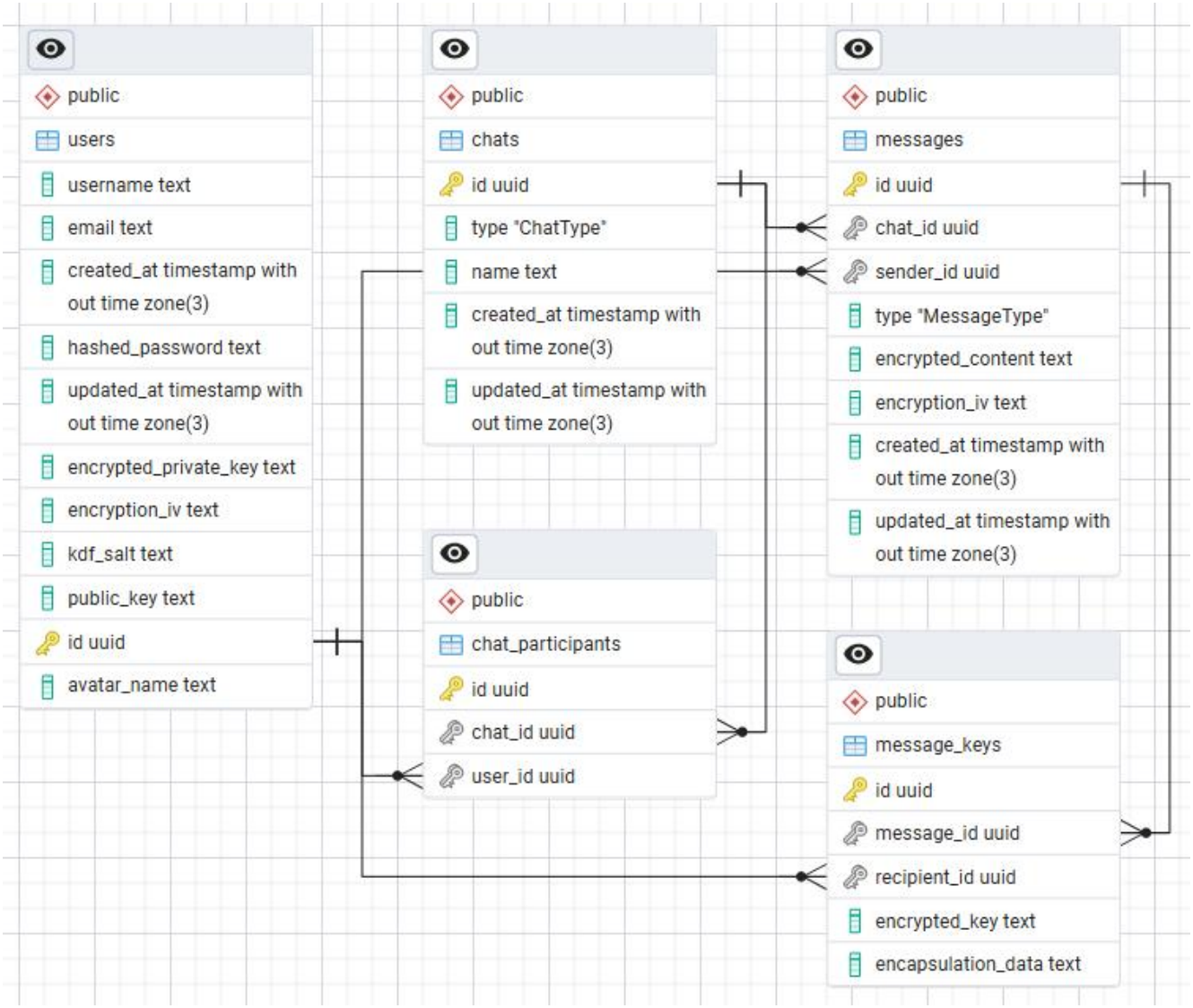


Рисунок 3.17 — ER-діаграма згенерована на основі схеми

Запропонована архітектура є гнучкою та легко адаптується для підтримки обміну файлами. Оскільки зберігання великих файлів безпосередньо в базі даних є неефективним, для передачі файлів використовується гібридний підхід:

- Файл шифрується на клієнті та завантажується у файлове сховище, локальну папку на сервері.
- У моделі Message зберігається не вміст файлу, а зашифроване посилання на нього та метадані (назва, розмір, тип), необхідні для відображення в інтерфейсі.
- Механізм розподілу ключів через таблицю MessageKey залишається незмінним, забезпечуючи захист доступу до файлу аналогічно текстовим повідомленням.

3.3 Реалізація управління сесіями та життєвим циклом ключів

3.3.1 Організація локального сховища криптографічних ключів

Критичною умовою зручності використання E2EE-додатку є збереження криптографічної сесії між перезавантаженнями сторінки. Користувач не повинен проходити процедуру деривації ключів при кожному оновленні вкладки браузера. Для вирішення цієї задачі було спроектовано шар локальної персистентності.

Стандартний механізм localStorage, який часто використовується у веб-розробці, виявився непридатним для зберігання криптографічних ключів, бо працює виключно з рядками. Збереження бінарних об’єктів вимагає їх серіалізації у base64, що створює додаткові вектори атаки та накладні витрати. Також, операції читання/запису є синхронними, що може блокувати основний потік виконання при роботі з великими даними.

Натомість було обрано IndexedDB — низькорівневий API для клієнтського зберігання значних обсягів структурованих даних. Його ключовою перевагою для даного проекту є підтримка алгоритму структурованого клонування, що дозволяє зберігати об’єкти CryptoKey у їх нативному вигляді без необхідності експорту в текстовий формат. Це підвищує безпеку, оскільки ключі залишаються внутрішніми об’єктами браузера.

Нативний API IndexedDB є подійно-орієнтованим та досить складним для використання у сучасному коді на TypeScript. Для інтеграції сховища в асинхронну архітектуру додатку було використано бібліотеку-обгортку idb. Вона надає зручний інтерфейс на основі Promises, що дозволяє використовувати синтаксис async/await для операцій з базою даних, забезпечуючи чистоту та читабельність коду.

На основі аналізу вимог до зберігання ключів було розроблено структуру локальної бази даних SecureDB. Схема бази даних визначена через TypeScript-інтерфейс MyAppDB, що забезпечує типізацію на етапі компіляції. База даних містить одне сховище об’єктів з назвою keys_store. Структура запису в цьому сховищі має наступний вигляд:

- Key — унікальний рядковий ідентифікатор запису.
- Value — об’єкт, що містить publicKey та privateKey типу CryptoKey.

Важливо зазначити, що при генерації ключів використовується параметр extractable: true. Це архітектурне рішення є необхідним, адже система повинна мати можливість експортувати приватний ключ, щоб зашифрувати його паролем користувача перед відправкою на сервер.

Програмна реалізація сервісу зберігання

Реалізація шару зберігання виконана з дотриманням принципу єдиної відповідальності та розділена на модуль ініціалізації і сервіс управління ключами.

Модуль ініціалізації експортує функцію getDb(), яка відповідає за відкриття з’єднання з базою даних та керування її версійністю. Під час ініціалізації перевіряється наявність сховища keys_store і, за його відсутності, відбувається його створення. Використання патерну Singleton для з’єднання з БД дозволяє уникнути відкриття зайвих з’єднань при багаторазових зверненнях.

Сервіс управління ключами KeyStoreService інкапсулює логіку операцій створення, зчитування та видалення над ключами. Він надає вищим рівням додатку абстрагований інтерфейс, приховуючи деталі роботи з idb.

Таблиця 3.6 — Опис KeyStoreService

Назва методу	Зміст
saveKeys	Приймає пару CryptoKey і зберігає їх під фіксованим ключем user_keys. Це спрощує логіку, оскільки в один момент часу в браузері активна лише одна сесія користувача
loadKeys	Асинхронно отримує збережену пару ключів, повертає null, якщо ключі відсутні, що слугує сигналом для додатку про необхідність ініціювати процедуру входу або відновлення сесії
clear	Виконує безпечне видалення ключів зі сховища, що є обов'язковим кроком при виході користувача з системи для запобігання несанкціонованому доступу до сесії з того ж пристрою

Екземпляр класу keyStoreService експортується як синглтон, що забезпечує єдину точку доступу до локального сховища ключів у всьому клієнтському додатку.

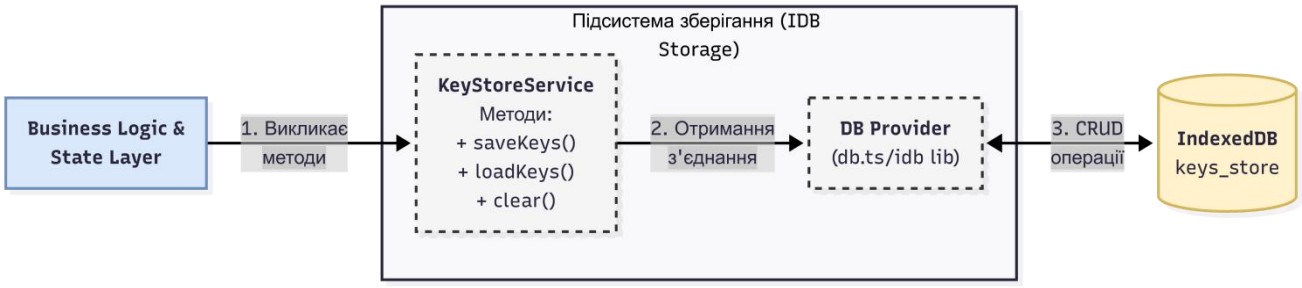


Рисунок 3.18 — Схема використання сховища ключів

3.3.2 Управління життєвим циклом криптографічної ідентичності

Ефективне управління криптографічними ключами є критично важливим аспектом системи наскрізного шифрування, оскільки втрата приватного ключа означає безповоротну втрату доступу до історії повідомлень, а його компрометація — порушення конфіденційності. Життєвий цикл ідентичності в розробленій системі побудований на взаємодії трьох компонентів: оперативної

пам’яті клієнта, де ключі використовуються, локального сховища IndexedDB, де ключі зберігаються між перезавантаженнями, та сервера, де зберігається зашифрована копія. Нижче наведено детальний опис алгоритмів для трьох основних сценаріїв використання: реєстрації, входу в систему та відновлення сесії.

Процес реєстрації нового користувача виходить за межі простого створення запису в базі даних. Він включає генерацію криптографічного матеріалу та створення механізму його відновлення. Алгоритм дій:

1) Генерація пари ключів, використовуючи метод crypto.subtle.generateKey, додаток створює нову пару ключів ECDH кривої P-256. Ключі генеруються з параметром extractable: true, що дозволяє експортувати їх для подальшого резервного копіювання.
2) Експорт пари ключів, де публічний ключ у форматі SPKI, а приватний у форматі PKCS8.
3) Генерується криптографічно випадкова salt.
4) Виведення ключа шифрування, де пароль, введений користувачем, разом із згенерованою сіллю подається на вхід функції PBKDF2. Результатом є симетричний ключ для огортання WrappingKey.
5) Приватний ключ експортований, як бінарний масив, шифрується алгоритмом AES-GCM з використанням WrappingKey. Результатом є захищений контейнер приватного ключа.
6) На сервер, для збереження, відправляються експортований публічний ключ publicKey, захищений контейнер приватного ключа encryptedPrivateKey, сіль kdfSalt та вектор ініціалізації encryptionIV.
7) В локальне сховище IndexedDB зберігається оригінальна, незашифрована, пара ключів CryptoKey для негайного використання без необхідності повторного введення пароля.

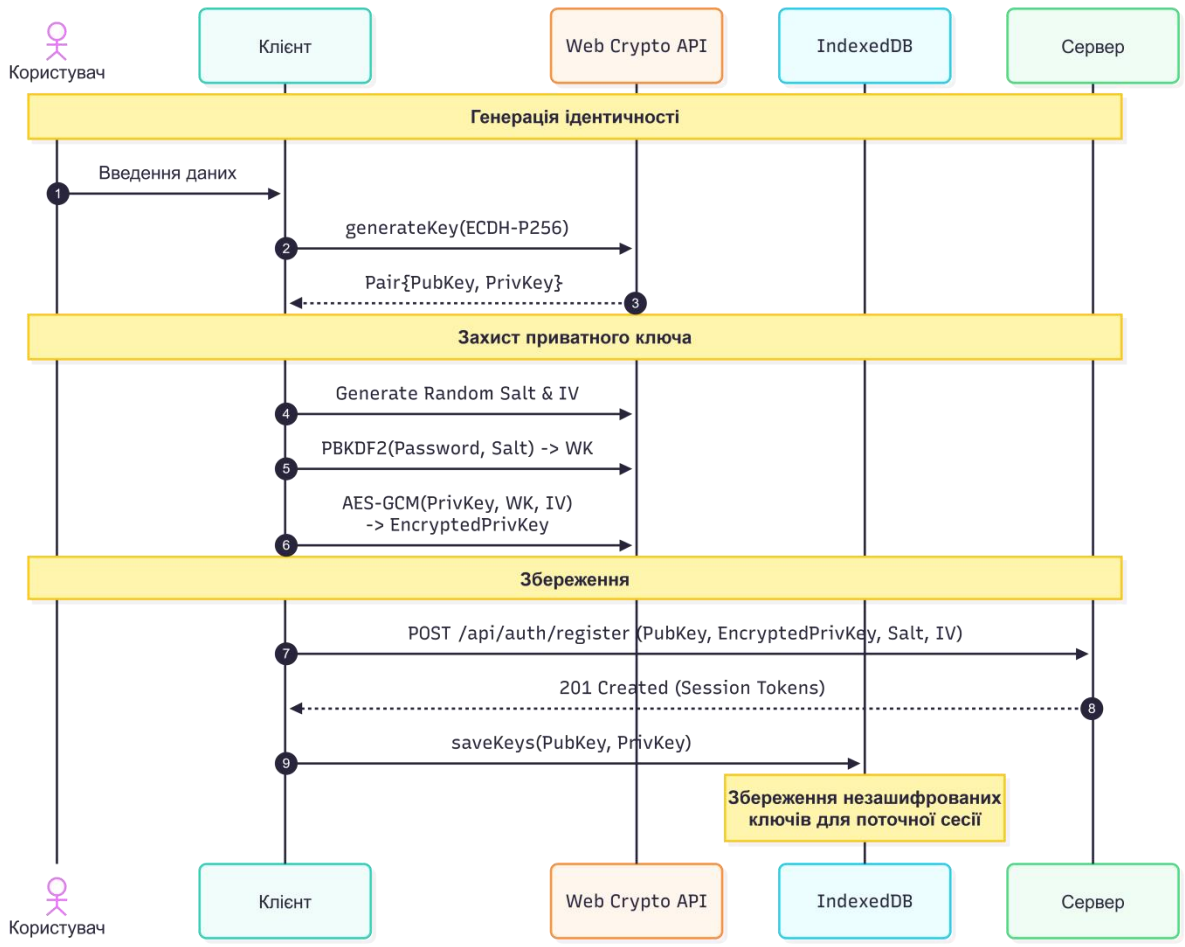


Рисунок 3.19 — Послідовність процесу реєстрації та первинної генерації ключів

Відновлення ключів з сервера активується, коли користувач входить у систему з нового пристрою або після очищення даних браузера. Головна задача — безпечно завантажити та розшифрувати приватний ключ. Алгоритм дій:

1) Клієнт відправляє логін/пароль на сервер, отримує JWT-токен та завантажує профіль користувача, який містить publicKey, encryptedPrivateKey, kdfSalt та encryptionIV.
2) Клієнт запитує у користувача пароль. Цей пароль та завантажена з сервера сіль використовуються для повторного виведення того самого симетричного ключа WrappingKey через PBKDF2.
3) За допомогою відновленого WrappingKey та encryptionIV виконується розшифрування контейнера.

4) Отримані байти, публічного і приватного ключів, імпортуються назад у формат CryptoKey браузера.
5) Відновлені ключі зберігаються в IndexedDB через сервіс KeyStoreService, щоб уникнути необхідності вводити пароль при наступному відкритті сторінки.

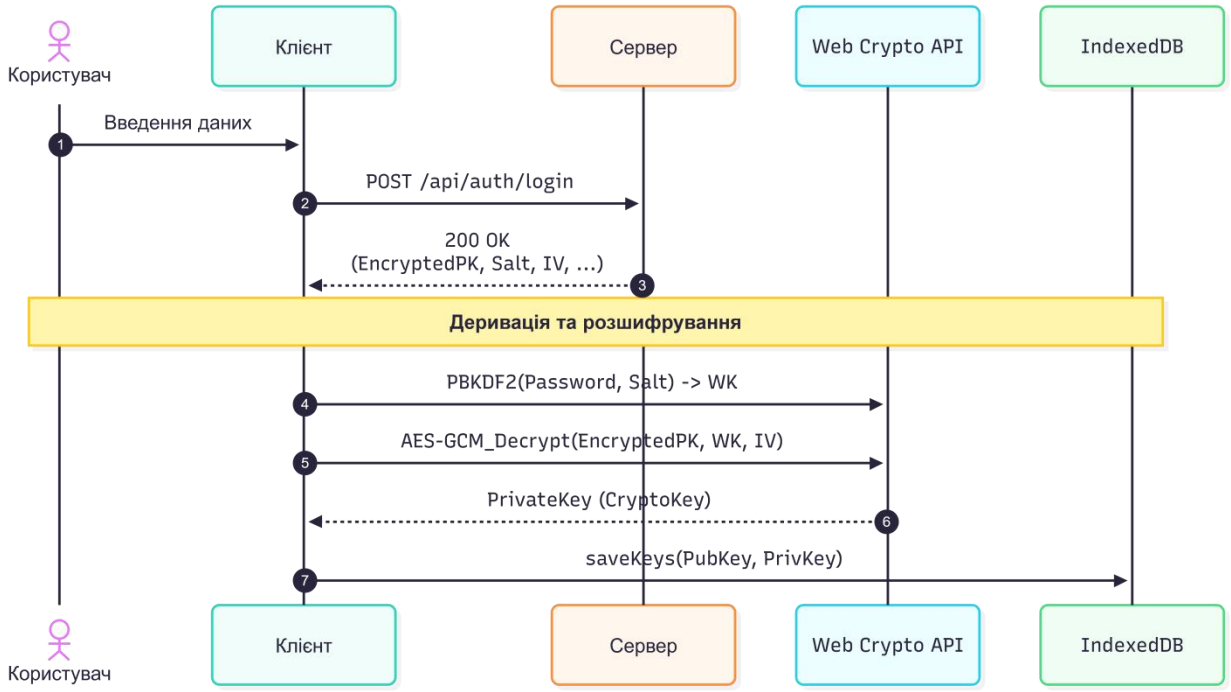


Рисунок 3.20 — Послідовність процесу входу та відновлення ключів з сервера

Відновлення сесії відбувається при кожному оновленні сторінки або повторному відкритті вкладки. Система повинна відновити готовність до шифрування/розшифрування максимально швидко і непомітно для користувача, але важливо переконатися у валідності поточної сесії на сервері. Алгоритм дій:

- Перевірка локального сховища при ініціалізації додатку, в хуках життєвого циклу головного компоненту, викликом методу keyStoreService.loadKeys().
- Якщо ключі успішно отримані з IndexedDB, клієнт виконує фоновий запит до ендпоінту /api/auth/me, що дозволяє перевірити валідність Access/Refresh токенів.

- Якщо сервер повертає статус 200 OK, ключі завантажуються в глобальний Pinia Store, встановлюється об’єкт поточного користувача, і додаток переходить у стан готовності до роботи.
- Якщо сервер повертає 401 Unauthorized або якщо ключі в IndexedDB відсутні (метод повертає null), система вважає сесію недійсною і виконує примусовий вихід, очищає сховище та перенаправляє користувача на сторінку входу. Це гарантує, що користувач не опиниться в ситуації, коли він авторизований, але не може читати повідомлення.

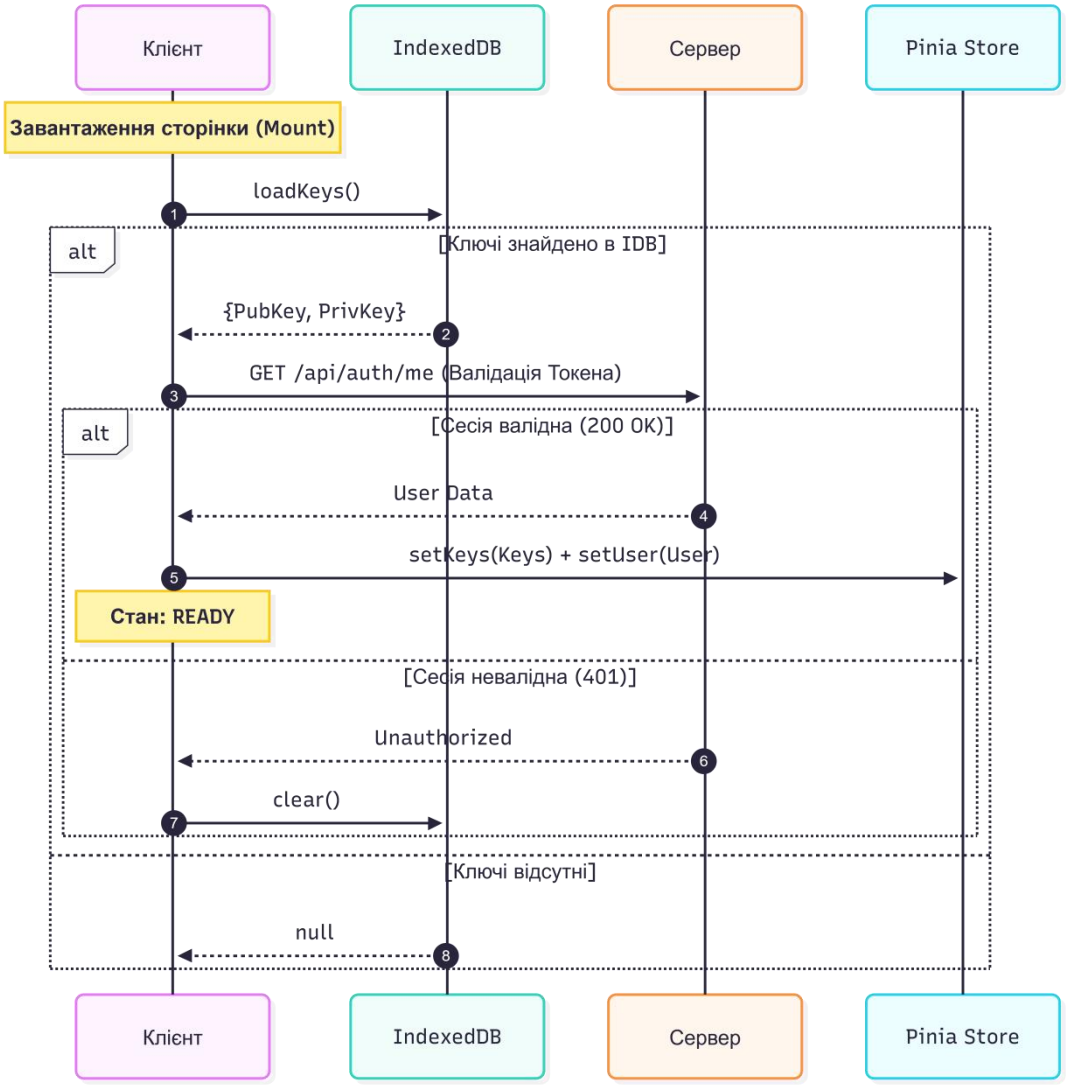


Рисунок 3.21 — Послідовність процесу відновлення сесії

Така трирівнева модель управління ключами забезпечує баланс між безпекою, адже ключі ніколи не передаються у відкритому вигляді, та зручністю, тому, що користувач вводить пароль лише один раз на пристрій.

3.4 Реалізація ядра шифрування

Реалізація криптографічного модуля є найбільш критичною частиною системи, оскільки будь-яка помилка в логіці або неправильне використання примітивів може призвести до компрометації даних. Тому при розробці цього модуля головний акцент було зроблено на суворій типізації, ізоляції відповідальності та передбачуваності поведінки коду.

Для написання модуля обрано мову TypeScript. Використання статичної типізації дозволяє на етапі компіляції виключити цілий клас помилок, пов’язаних з некоректною обробкою типів даних, наприклад передачу рядка base64 у функцію, що очікує Uint8Array. Усі вхідні та вихідні параметри криптографічних функцій описані через інтерфейси, що гарантує дотримання контрактів взаємодії.

Архітектурні патерни та оркестрація сервісів

Архітектура ядра базується на об’єктно-орієнтованому підході з використанням принципів SOLID. Для організації коду та управління залежностями застосовано наступні патерни проектування: впровадження залежностей, одинак, фасад.

Відповідно Dependency Injection, класи сервісів не створюють екземпляри своїх залежностей самостійно, а отримують їх через конструктор. Це забезпечує слабку зв’язність і дозволяє легко замінювати реалізації низькорівневих сервісів без зміни коду високорівневих.

Оскільки криптографічні сервіси не зберігають внутрішнього стану і оперують лише даними, що передаються в аргументах методів, вони передаються в єдиному екземплярі на весь час життя додатку.

Для простоти використання логіки гібридного шифрування, створено фасад який надає спрощений інтерфейс для складних операцій, приховуючи взаємодію декількох сервісів.

Ініціалізація та зв’язування компонентів відбувається у точці входу модуля services/encryption/index.ts. Цей файл виконує роль Composition Root, де створюються екземпляри класів і формується граф залежностей.

Логічна структура ядра складається з п’яти спеціалізованих сервісів, які можна розділити на три рівні абстракції:

- Рівень генерації та захисту ключів, який включає HpkeKeyService і KeyProtectionService.
- Рівень криптографічних примітивів, який включає SymmetricCryptoService і HpkeService.
- Рівень оркестрації, на якому HybridCryptoService об’єднує роботу попередніх компонентів.

Така декомпозиція дозволяє локалізувати логіку кожного криптографічного кроку в окремому класі, що спрощує аудит коду та його тестування.

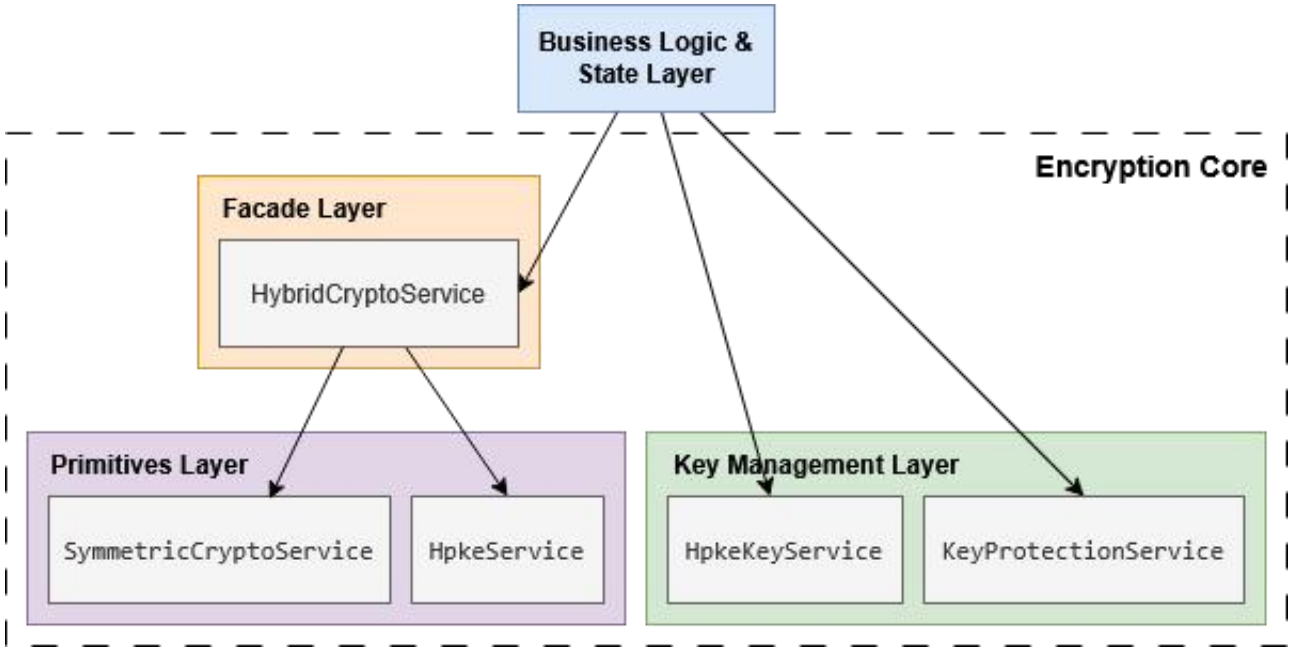


Рисунок 3.22 — Схема ядра шифрування

Сервіс управління асиметричними ключами HpkeKeyService

Першим елементом у ланцюжку криптографічних перетворень є сервіс HpkeKeyService. Цей клас відповідає за генерацію, експорт та імпорт пар ключів, що використовуються в протоколі HPKE. Він імплементує інтерфейс IHpkeKeyService, що забезпечує відповідність контракту взаємодії з іншими компонентами системи.

У класі жорстко зафіксовано конфігурацію алгоритму:

```
private readonly ALG = { name: 'ECDH', namedCurve: 'P-256' }
```

Використання алгоритму ECDH на кривій P-256 (secp256r1) обумовлено нативною підтримкою у всіх сучасних браузерах та відповідністю рекомендаціям NIST. Це забезпечує баланс між швидкістю генерації ключів та їх криптографічною стійкістю.

Таблиця 3.7 — Функціональні можливості HpkeKeyService

Функція/Метод	Опис
Генерація ключів (generateKeyPair)	Метод використовує crypto.subtle.generateKey для створення нової пари. Критично важливим параметром є extractable: true. Це дозволяє додатку надалі експортувати закритий ключ для його шифрування та резервного копіювання на сервері. Властивість deriveBits вказує на призначення ключа — виведення спільних секретів shared secrets.
Експорт ключів (exportPublicKey, exportPrivateKey)	Для збереження ключів у базі даних їх необхідно перетворити з об’єкта CryptoKey у бінарний формат ArrayBuffer. Сервіс реалізує експорт у стандартних форматах сумісності, SPKI (Subject Public Key Info) для публічних ключів, і PKCS8 (Private-Key Information Syntax Standard) для приватних ключів.
Імпорт ключів (importPublicKey, importPrivateKey)	Методи виконують зворотну операцію — відновлюють об’єкт CryptoKey з бінарного буфера. Це необхідно при завантаженні ключів з локального сховища або після розшифрування резервної копії з сервера.

Сервіс захисту приватного ключа KeyProtectionService

Сервіс KeyProtectionService є критично важливим компонентом безпеки, що реалізує механізм перетворення чутливого приватного ключа у безпечний для транспортування та зберігання контейнер. Він імплементує інтерфейс IKeyProtectionService та забезпечує захист даних методом шифрування на основі пароля.

У класі визначено жорстку конфігурацію алгоритмів, що забезпечує відповідність сучасним стандартам безпеки:

```
private readonly ALG = {
   pbkdf2: { name: 'PBKDF2', hash: 'SHA-256', iterations: 1000 },
   aes: { name: 'AES-GCM', length: 256 },
}
```

Специфіка роботи з Web Cryptography API вимагає двоетапного процесу деривації, який у коді інкапсульовано у приватні методи. Спочатку метод getPasswordKey імпортує рядкове представлення пароля у форматі raw як базовий ключовий матеріал. Після цього метод deriveKey виконує обчислювально складну операцію PBKDF2, використовуючи унікальну сіль, для генерації фінального ключа огортання wrapping key, придатного для алгоритму AES-GCM.

Таблиця 3.8 — Методи захисту ключів в KeyProtectionService

Метод	**Опис**
protect	Захисту ключа розпочинається з генерації ентропії: за допомогою криптографічно стійкого генератора створюються випадкова сіль та вектор ініціалізації. Отримана сіль використовується для деривації ключа огортання з пароля користувача. На фінальному етапі “сирі” байти приватного ключа шифруються отриманим симетричним ключем. Результатом роботи методу є структурований об’єкт, що містить шифротекст, сіль та вектор ініціалізації.
restore	Метод виконує зворотну послідовність дій. Критично важливою умовою є використання тієї ж солі, що була згенерована під час захисту. Алгоритм витягує параметри з переданого об’єкта та виконує повторну деривацію ключа з введеного користувачем пароля, для подальшо використання при розшифруванні даних.

Сервіс симетричного шифрування даних SymmetricCryptoService

Сервіс SymmetricCryptoService відповідає за безпосереднє шифрування змісту повідомлень, реалізуючи рівень даних Payload Layer у загальній архітектурі гібридного шифрування. Клас імплементує інтерфейс ISymmetricCryptoService та розроблений для забезпечення високої продуктивності операцій шифрування, оскільки саме на цей компонент припадає основне обчислювальне навантаження при передачі великих обсягів інформації.

Конфігурація криптографічного примітиву, зафіксована у властивостях класу:

```
private readonly ALG = { name: 'AES-GCM', length: 256 }
```

Життєвий цикл симетричного ключа в цьому сервісі починається з методу generateKey, який створює ефемерний ключ для сесії або конкретного повідомлення. Оскільки архітектура Digital Envelope вимагає передачі цього ключа іншим учасникам чату, сервіс реалізує методи серіалізації.

Метод exportKeyRaw експортує ключ у формат (aw bytes) що дозволяє подальшу його інкапсуляцію асиметричними алгоритмами. Відповідно, метод importKeyRaw дозволяє відновити об’єкт CryptoKey з байтового масиву після того, як його було розшифровано на стороні отримувача.

Таблиця 3.9 — Методи захисту даних в SymmetricCryptoService

Метод	Опис
encrypt	Метод починається з генерації нового випадкового IV довжиною за допомогою криптографічно стійкого генератора. Результатом виконання операції є об’єкт типу SymmetricPacket, який інкапсулює пару значень: отриманий шифротекст та вектор ініціалізації, що був використаний для його створення.
decrypt	Приймає відновлений симетричний ключ та пакет даних, витягує вектор ініціалізації з пакету та передає його разом із шифротекстом у функцію crypto.subtle.decrypt.

Сервіс інкапсуляції ключів HpkeService

Сервіс HpkeService є ключовим елементом архітектури гібридного шифрування, що відповідає за безпечний транспорт симетричних ключів між учасниками обміну. Клас імплементує інтерфейс IHpkeService та реалізує логіку стандарту HPKE, адаптовану для виконання в браузерному середовищі за допомогою базових примітивів Web Cryptography API.

На відміну від використання готових високорівневих бібліотек, дана реалізація вручну оркеструє процеси узгодження ключів та розгортання контексту. Це дозволяє мати повний контроль над параметрами алгоритму та мінімізувати розмір кінцевого бандлу додатку. Криптографічний набір, зафіксований у конфігурації класу, відповідає сучасним рекомендаціям безпеки та складається з трьох компонентів:

```
private readonly ALG = {
  kem: { name: 'ECDH', namedCurve: 'P-256' }, // Механізм інкапсуляції
  kdf: { name: 'HKDF', hash: 'SHA-256' },      // Функція деривації
  aead: { name: 'AES-GCM', length: 256 },      // Автентифіковане шифрування
}
```

Центральним елементом логіки сервісу є приватний метод deriveEncryptionContext. Він відповідає за те, щоб дві сторони, маючи лише пари ключів, свою приватну та чужу публічну, могли незалежно один від одного отримати ідентичні параметри шифрування. Процес деривації реалізовано наступним чином:

- Обчислення спільного секрету, де за допомогою методу deriveBits виконується операція Діффі-Хеллмана на еліптичній кривій. Результатом є “сирий” бітовий масив, який ще не можна використовувати як ключ.
- Отриманий секрет імпортується як матеріал для алгоритму HKDF. Далі виконується повторний виклик deriveBits, який генерує потік псевдовипадкових байтів необхідної довжини (у даному випадку 44 байти: 32 для ключа + 12 для IV).

- Згенерований потік розділяється на дві частини, де перші 32 байти імпортуються як ключ AES-GCM, а наступні 12 байт стають вектором ініціалізації.

Публічні методи сервісу забезпечують високорівневу взаємодію з механізмами інкапсуляції та декапсуляції.

Таблиця 3.10 — Публічні методи в HpkeService

Метод	**Опис**
seal	Метод (інкапсуляція) виконується на стороні відправника. Він генерує одноразову ефемерну пару ключів ephemeralKey спеціально для цієї транзакції. Використовуючи ефемерний приватний ключ та публічний ключ отримувача, метод формує контекст шифрування та запечатує вхідні дані, зазвичай це симетричний ключ повідомлення. Результатом є об’єкт HpkePacket, що містить шифротекст та enc — експортований ефемерний публічний ключ.
open	Метод (декапсуляція) виконується на стороні отримувача. Він приймає пакет даних та довгостроковий приватний ключ користувача. Першим кроком є імпорт enc, після чого відбувається дзеркальна процедура деривації контексту. Отримавши ідентичний AES-ключ та IV, метод розшифровує вміст пакету, повертаючи вихідний буфер даних.

Фасадний сервіс гібридного шифрування HybridCryptoService

Завершує архітектуру криптографічного модуля сервіс HybridCryptoService, який виконує роль оркестратора або фасаду. Цей клас імплементує інтерфейс IHybridCryptoService і є єдиною точкою входу для прикладної логіки додатку при роботі з повідомленнями.

На відміну від попередніх сервісів, цей клас не реалізує математичних алгоритмів самостійно. Натомість, він використовує патерн Dependency Injection, приймаючи екземпляри ISymmetricCryptoService та IHpkeService у свій

конструктор. Такий підхід забезпечує модульність системи, де логіка склеювання алгоритмів відокремлена від їхньої реалізації.

За реалізацію патерну Digital Envelope, в сервісі, відповідає метод encrypt, головною задачею якого є створення захищеного контейнера для групи отримувачів. Алгоритм побудований таким чином, щоб ресурсомістка операція шифрування даних виконувалася лише один раз, незалежно від кількості адресатів. Процес складається з наступних етапів:

1) За допомогою симетричного сервісу створюється унікальний ефемерний ключ contentKey.
2) Вхідні дані шифруються цим ключем. Отриманий шифротекст і вектор ініціалізації стають спільною частиною повідомлення для всіх.
3) Ефемерний ключ експортується у бінарний формат raw bytes, щоб його можна було обробити як звичайні дані.
4) Сервіс ітерує список отримувачів, включаючи самого відправника і для кожного користувача виконується операція hpkeService.seal, яка шифрує байти contentKey публічним ключем цього конкретного користувача.

Результатом роботи є об’єкт типу DigitalEnvelope, що містить один великий блок зашифрованих даних та масив легких recipientBoxes, кожна з яких призначена конкретному учаснику чату.

Логіка розшифрування та відновлення доступу припадає на метод decrypt, який реалізує зворотний процес, приймаючи цифровий конверт та приватний ключ поточного користувача. Логіка методу включає механізми пошуку та відновлення:

1) Алгоритм сканує список recipients у конверті, шукаючи запис, що відповідає ID поточного користувача myId. Якщо запис не знайдено, це означає, що користувач не має доступу до цього повідомлення, і викидається помилка.
2) Знайдений зашифрований ключ encryptedKey та ефемерні дані encapsulationData передаються у hpkeService.open. Результатом є байти симетричного ключа.

3) Отримані байти імпортуються назад у повноцінний об’єкт CryptoKey за допомогою симетричного сервісу, після чого виконується фінальне розшифрування основного контенту повідомлення.

Така архітектура гарантує, що навіть при перехопленні DigitalEnvelope, зловмисник не зможе отримати доступ до даних, оскільки не володіє приватним ключем жодного з учасників для розгортання конверта з ключем контенту.

Провівши детальний огляд всіх сервісів, можна доповнити схему ядра шифрування конкретними методами та полями і представити у вигляді діаграми класів рисунок 3.23.

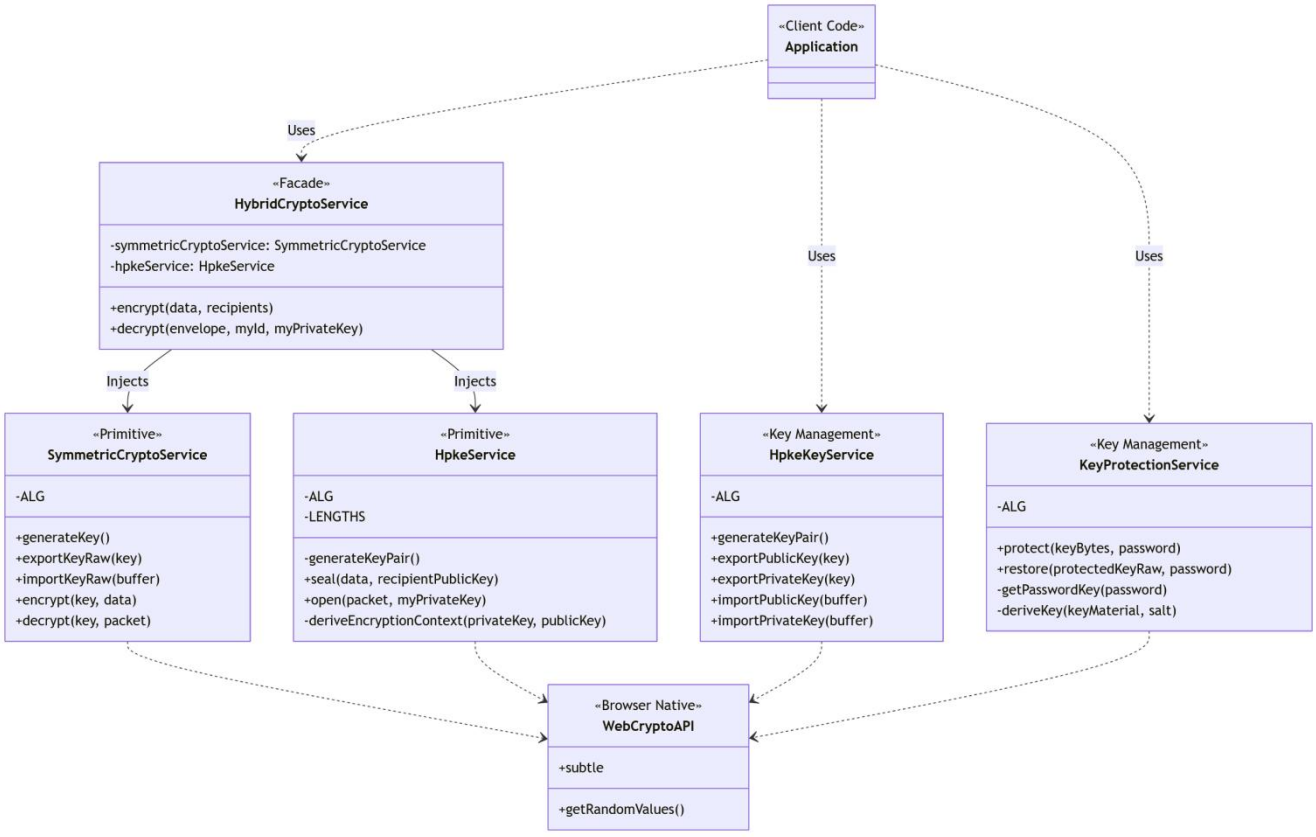


Рисунок 3.23 — Діаграма класів ядра шифрування

Розроблені сервіси забезпечують повний цикл захисту інформації, від генерації та персистентності ключів до реалізації ефективної гібридної схеми шифрування, яка стійка до вразливостей.

ВИСНОВКИ

У магістерській кваліфікаційній роботі вирішено науково-технічну задачу забезпечення конфіденційності інформаційного обміну у веб-середовищі. За результатами проведеного аналізу загроз та обмежень існуючих протоколів було обгрунтовано та розроблено гібридну криптографічну модель. Такий підхід дозволив поєднати високу продуктивність симетричного шифрування AES-GCM з надійністю асиметричного стандарту HPKE на еліптичній кривій P-256, що вирішило проблему масштабування.

Спроектовано захищену клієнт-серверну архітектуру за принципом Zero-Knowledge Architecture, де сервер виступає виключно як недовірений ретранслятор зашифрованих контейнерів. Ключовим досягненням стала реалізація безпечної стратегії управління ідентичністю, де генерація ключів відбувається виключно на клієнті засобами Web Cryptography API, а їх мобільність забезпечується через зберігання на сервері криптографічного контейнера, захищеного ключем, виведеного з пароля. Це дозволило досягти балансу між безпекою та зручністю доступу з різних пристроїв.

У рамках практичної частини розроблено програмний прототип ядра шифрування на мові TypeScript. Реалізація включає вирішення специфічних інженерних проблем веб-середовища: обмеження оперативної пам’яті браузера подолано завдяки впровадженню алгоритму потокового шифрування для об’ємних даних, а типовий недолік E2EE-систем усунуто через механізм синхронізації історії для відправника. Також забезпечено надійне управління життєвим циклом сесій із використанням локального сховища IndexedDB.

Практична значимість отриманих результатів полягає у створенні відкритої, аудитоздатної архітектури, яка може бути використана як фундамент для побудови захищених корпоративних комунікаційних систем, незалежних від сторонніх сервісів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 24 Канал, Найбільший витік в історії, https://24tv.ua/tech/naybilshiy-vitik-paroliv-istoriyi-ohopiv-16-milyardiv-danih-apple_n2850443
2. Інститут масової інформації, У 2024 році кількість кібератак на Україну зросла на 70%, https://imi.org.ua/news/u-2024-rotsi-kilkist-kiberatak-na-ukrayinu-zrosla-na-70-i65931
3. ZMINA, Від початку 2025 року в Україні фіксують близько 15 кібератак щодня, https://zmina.info/news/medijnyky-z-bbc-news-ap-reuters-ta-afp-goloduyut-u-sektori-gazy/
4. GeraBot, Популярність месенджерів в Україні у 2022–2025 роках: Telegram, Viber та WhatsApp, https://gerabot.com/article/populyarnist_mesendzheriv_v_ukraini_u_20222025_rokah_telegram_viber_ta_whatsapp
5. Signal,. The Signal Protocol, https://signal.org/docs/
6. IETF, RFC 9420: The Messaging Layer Security (MLS) Protocol, https://www.rfc-editor.org/rfc/rfc9420.html
7. Cloudflare Blog, HPKE: Standardizing public-key encryption (finally!), https://blog.cloudflare.com/hybrid-public-key-encryption/
8. Interaction Design Foundation, Native, Web or Hybrid App: Which One is Better?, https://www.interaction-design.org/literature/article/native-vs-hybrid-vs-responsive-what-app-flavour-is-best-for-you
9. Amazon Web Services, What's the Difference Between Web Apps, Native Apps, and Hybrid Apps?, https://aws.amazon.com/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/
10. Electronic Frontier Foundation, A Deep Dive on End-to-End Encryption: How Do Public Key Encryption Systems Work?, https://ssd.eff.org/module/deep-dive-end-end-encryption-how-do-public-key-encryption-systems-work
11. Emily M. Stark, E2EE on the web: isolating plaintext, https://emilymstark.com/2023/09/09/e2ee-on-the-web-isolating-plaintext.html

12. Stack Overflow, Is end-to-end encryption possible in JavaScript? https://stackoverflow.com/questions/28526464/is-end-to-end-encryption-possible-in-javascript
13. Thales Group, What is Web Cryptography API?, https://cds.thalesgroup.com/en/hot-topics/what-web-cryptography-api
14. Signal, Technology preview: Signal Private Group System, https://signal.org/blog/signal-private-group-system/
15. Cryptography Engineering, Attack of the Week: Group Messaging in WhatsApp and Signal, https://blog.cryptographyengineering.com/2018/01/10/attack-of-the-week-group-messaging-in-whatsapp-and-signal/
16. Wikipedia, Messaging Layer Security, https://en.wikipedia.org/wiki/Messaging_Layer_Security
17. IETF, RFC 9180: Hybrid Public Key Encryption, https://www.rfc-editor.org/info/rfc9180
18. Національний інститут стандартів і технологій США (NIST), NIST SP 1800-25, https://www.nccoe.nist.gov/publication/1800-25/VolA/index.html
19. IBM, What Is a Man-in-the-Middle (MITM) Attack?, https://www.ibm.com/think/topics/man-in-the-middle
20. Vaadata, What is a Man in the Middle (MitM) Attacks? Types and Security Best Practices, https://www.vaadata.com/blog/what-is-a-man-in-the-middle-mitm-attack-types-and-security-best-practices/
21. Galaxkey, What's the difference between transport-layer encryption and end-to-end encryption?, https://www.galaxkey.com/transport-layer-encryption-vs-end-to-end-encryption/
22. Electronic Frontier Foundation, What Should I Know About Encryption?, https://ssd.eff.org/module/what-should-i-know-about-encryption
23. VMware, Perfect Forward Secrecy Definition, https://www.vmware.com/topics/perfect-forward-secrecy
24. Signal, Signal Protocol and Post-Quantum Ratchets, https://signal.org/blog/spqr/

25. Signal, Simplifying OTR deniability, https://signal.org/blog/simplifying-otr-deniability/
26. Wikipedia, Pretty Good Privacy, https://en.wikipedia.org/wiki/Pretty_Good_Privacy
27. Virtru, What is PGP Encryption? https://www.virtru.com/blog/email-encryption/pgp
28. Wikipedia, Off-the-Record Messaging, https://en.wikipedia.org/wiki/Off-the-record_messaging
29. Web Encrypt, Difference between PGP and OTR, https://webencrypt.org/otr/
30. Signal, Simplifying OTR deniability, https://signal.org/blog/simplifying-otr-deniability/
31. ResearchGate, How Practical is OTR?, https://www.researchgate.net/publication/324437690_How_Practical_is_OTR
32. Netzpolitik.org, Silent Circle Instant Messaging Protocol, Protocol Specification, https://netzpolitik.org/wp-upload/SCIMP-paper.pdf
33. Signal, Advanced cryptographic ratcheting, https://signal.org/blog/advanced-ratcheting/
34. Signal, The X3DH Key Agreement Protocol, https://signal.org/docs/specifications/x3dh/
35. Signal, The Double Ratchet Algorithm, https://signal.org/docs/specifications/doubleratchet/
36. Bachelor Thesis Computing Science, The X3DH Protocol: A Proof of Security, https://www.cs.ru.nl/bachelors-theses/2021/Ferran_van_der_Have___4104145___The_X3DH_Protocol_-_A_Proof_of_Security.pdf
37. Signal, Private Group Messaging, https://signal.org/blog/private-groups/
38. Messaging Layer Security Rocks, The Messaging Layer Security (MLS) Architecture, https://messaginglayersecurity.rocks/mls-architecture/issue280_restrictive/draft-ietf-mls-architecture.html
39. Wikipedia, Messaging Layer Security, https://en.wikipedia.org/wiki/Messaging_Layer_Security

40. Quarkslab, Secure Messaging Apps and Group Protocols, Part 2, https://blog.quarkslab.com/secure-messaging-apps-and-group-protocols-part-2.html
41. GitHub, openmls/openmls: Rust implementation of the Messaging Layer Security (MLS) protocol, https://github.com/openmls/openmls
42. Franziskuskiefer, TL;DR - Hybrid Public Key Encryption, https://www.franziskuskiefer.de/p/tldr-hybrid-public-key-encryption/#hpke
43. Benjamin Lipp, Hybrid Public Key Encryption: My Involvement in Development and Analysis of a Cryptographic Standard, https://www.benjaminlipp.de/p/hpke-cryptographic-standard/
44. MDN Web Docs, Web Crypto API, https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API
45. MDN Web Docs, CryptoKey: extractable property, https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey/extractable
46. WhatsApp, WhatsApp Encryption Overview, https://scontent.xx.fbcdn.net/v/t39.8562-6/455962147_1148247109601582_1673264986279156121_n.pdf?_nc_cat=101&ccb=1-7&_nc_sid=e280be&_nc_ohc=lioPTbj7loQQ7kNvwFA4PZ3&_nc_oc=Adk9rmL2RswpZ6BxaMi_1k68dkwO-Mfi_L1hXeczzmjKP8WmAuPhOTOliCgIoJV3I90&_nc_zt=14&_nc_ht=scontent.xx&_nc_gid=QTH3kL3EnTmVg6KXVDl8QQ&oh=00_AfeTGs1XHxpGY005Cf8VlDFB3oUNH0Bh5Nprc7ad1fxRkw&oe=68FA4519
47. Telegram, MTProto Mobile Protocol, https://core.telegram.org/mtproto
48. Telegram, End-to-End Encryption, Secret Chats, https://core.telegram.org/api/end-to-end
49. Apple Support, Apple Platform Security, https://support.apple.com/uk-ua/guide/security/welcome/web
50. Viber, Private and secure messaging, https://www.viber.com/en/security/
51. Rocket.Chat Docs, End-to-End Encryption Specifications, https://docs.rocket.chat/docs/end-to-end-encryption-specifications

52. JWT Debugger, Introduction to JSON Web Tokens, https://www.jwt.io/introduction#what-is-json-web-token
53. Javascript.Info, LocalStorage, sessionStorage, https://uk.javascript.info/localstorage
54. Medium, Understanding Access Tokens and Refresh Tokens, https://medium.com/@lakshyakumarsingh.2003.va/understanding-access-tokens-and-refresh-tokens-2ec4bc4f9a4f
55. MDN, Using HTTP cookies, https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies#:~:text=A%20cookie%20with%20the%20HttpOnly,make%20them%20available%20to%20JavaScript.
56. Вікіпедія, PBKDF2, https://uk.wikipedia.org/wiki/PBKDF2
57. FoxmindEd, Що таке брутфорс атака і які є методи захисту?, https://foxminded.ua/brute-force/
58. TypeScript Lang, TypeScript is JavaScript with syntax for types, https://www.typescriptlang.org/
59. Vue.js, The Progressive JavaScript Framework, https://vuejs.org/
60. TailwindCSS, Rapidly build modern websites without ever leaving your HTML, https://tailwindcss.com/
61. Pinia, Pinia The intuitive store for Vue.js, https://pinia.vuejs.org/
62. Dajiaji/hpke-js, A Hybrid Public Key Encryption (HPKE) module built on top of Web Cryptography API, https://github.com/dajiaji/hpke-js
63. Privacyresearchgroup/libsignal-protocol-typescript, Signal Protocol library for TypeScript, https://github.com/privacyresearchgroup/libsignal-protocol-typescript
64. Node.js, Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts, https://nodejs.org/en
65. PostgreSQL, PostgreSQL: The World's Most Advanced Open Source Relational Database, https://www.postgresql.org/
66. Prisma, From idea to scale. Simplified., https://www.prisma.io/
67. ExpressJS, Fast, unopinionated, minimalist web framework for Node.js, https://expressjs.com/

68. Go It, REST API: що це, як працює, переваги і приклади, https://goit.global/ua/articles/rest-api-shcho-tse-iak-pratsiuie-perevahy-i-pryklady/
69. Axios Http, Що таке Axios?, https://axios-http.com/uk/docs/intro
70. MDN, The WebSocket API (WebSockets), https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
71. Socket.io, Bidirectional and low-latency communication for every platform, https://socket.io/
72. Zod, TypeScript-first schema validation with static type inference, https://zod.dev/
73. Wikipedia, GCM Galois/Counter Mode — лічильник з автентифікацією Галуа, https://uk.wikipedia.org/wiki/Galois/Counter_Mode

ДОДАТКИ

Лістинг storage/db.ts

```
import { openDB, type IDBPDatabase, type DBSchema } from 'idb'

const DB_NAME = 'SecureDB'
const DB_VERSION = 1
const KEY_STORE_NAME = 'keys_store'

interface MyAppDB extends DBSchema {
  [KEY_STORE_NAME]: {
    key: string
    value: {
      publicKey: CryptoKey
      privateKey: CryptoKey
    }
  }
}

export async function getDb(): Promise<IDBPDatabase<MyAppDB>> {
  return openDB<MyAppDB>(DB_NAME, DB_VERSION, {
    upgrade(db) {
      if (!db.objectStoreNames.contains(KEY_STORE_NAME)) {
        db.createObjectStore(KEY_STORE_NAME)
      }
    },
  })
}
```

Лістинг storage/key-store.service.ts

```
import { getDb } from '@/services/storage/db.ts'

class KeyStoreService {
  private readonly KEY_STORE_NAME = 'keys_store'
  private readonly KEY_ID = 'user_keys'

  async saveKeys(publicKey: CryptoKey, privateKey: CryptoKey): Promise<void> {
    const db = await getDb()

    await db.put(this.KEY_STORE_NAME, { publicKey, privateKey }, this.KEY_ID)
  }

  async loadKeys(): Promise<{ publicKey: CryptoKey; privateKey: CryptoKey } | null> {
    const db = await getDb()

    const result = await db.get(this.KEY_STORE_NAME, this.KEY_ID)

    return result || null
  }
```

```
  async clear(): Promise<void> {
    const db = await getDb()

    await db.delete(this.KEY_STORE_NAME, this.KEY_ID)
  }
}

export const keyStoreService = new KeyStoreService()
```

Лістинг encryption/hpke-key.service.ts

```
import type { IHpkeKeyService } from '@/interfaces/encryption.interface.ts'

export class HpkeKeyService implements IHpkeKeyService {
  private readonly ALG = { name: 'ECDH', namedCurve: 'P-256' }

  async generateKeyPair(): Promise<CryptoKeyPair> {
    return crypto.subtle.generateKey(this.ALG, true, ['deriveBits'])
  }

  async exportPublicKey(key: CryptoKey): Promise<ArrayBuffer> {
    return crypto.subtle.exportKey('spki', key)
  }

  async exportPrivateKey(key: CryptoKey): Promise<ArrayBuffer> {
    return crypto.subtle.exportKey('pkcs8', key)
  }

  async importPublicKey(buffer: BufferSource): Promise<CryptoKey> {
    return crypto.subtle.importKey('spki', buffer, this.ALG, true, [])
  }

  async importPrivateKey(buffer: BufferSource): Promise<CryptoKey> {
    return crypto.subtle.importKey('pkcs8', buffer, this.ALG, true, ['deriveBits'])
  }
}
```

Лістинг encryption/key-protection.service.ts

```
import type { ProtectedKeyRaw } from '@/types/encryption.type'
import type { IKeyProtectionService } from '@/interfaces/encryption.interface.ts'

export class KeyProtectionService implements IKeyProtectionService {
  private readonly ALG = {
    pbkdf2: { name: 'PBKDF2', hash: 'SHA-256', iterations: 1000 },
    aes: { name: 'AES-GCM', length: 256 },
  }

  private encoder = new TextEncoder()
```

```
  async protect(keyBytes: BufferSource, password: string): Promise<ProtectedKeyRaw> {
    const salt = crypto.getRandomValues(new Uint8Array(16))
    const iv = crypto.getRandomValues(new Uint8Array(12))

    const keyMaterial = await this.getPasswordKey(password)
    const wrappingKey = await this.deriveKey(keyMaterial, salt)

    const ciphertext = await crypto.subtle.encrypt(
      { name: this.ALG.aes.name, iv },
      wrappingKey,
      keyBytes,
    )

    return {
      ciphertext,
      salt,
      iv,
    }
  }

  async restore(protectedKeyRaw: ProtectedKeyRaw, password: string): Promise<ArrayBuffer> {
    const { ciphertext, salt, iv } = protectedKeyRaw

    const keyMaterial = await this.getPasswordKey(password)
    const wrappingKey = await this.deriveKey(keyMaterial, salt)

    return crypto.subtle.decrypt({ name: this.ALG.aes.name, iv }, wrappingKey, ciphertext)
  }

  private async getPasswordKey(password: string): Promise<CryptoKey> {
    return crypto.subtle.importKey(
      'raw',
      this.encoder.encode(password),
      this.ALG.pbkdf2.name,
      false,
      ['deriveKey'],
    )
  }

  private async deriveKey(keyMaterial: CryptoKey, salt: BufferSource): Promise<CryptoKey> {
    return crypto.subtle.deriveKey({ ...this.ALG.pbkdf2, salt }, keyMaterial, this.ALG.aes, false, [
      'encrypt',
      'decrypt',
    ])
  }
}
```

Лістинг encryption/symmetric-crypto.service.ts

```
import type { SymmetricPacket } from '@/types/encryption.type.ts'
import type { ISymmetricCryptoService } from '@/interfaces/encryption.interface.ts'
```

```
export class SymmetricCryptoService implements ISymmetricCryptoService {
  private readonly ALG = { name: 'AES-GCM', length: 256 }

  async generateKey(): Promise<CryptoKey> {
    return crypto.subtle.generateKey(this.ALG, true, ['encrypt', 'decrypt'])
  }

  async exportKeyRaw(key: CryptoKey): Promise<ArrayBuffer> {
    return crypto.subtle.exportKey('raw', key)
  }

  async importKeyRaw(buffer: BufferSource): Promise<CryptoKey> {
    return crypto.subtle.importKey('raw', buffer, this.ALG, true, ['encrypt', 'decrypt'])
  }

  async encrypt(key: CryptoKey, data: BufferSource): Promise<SymmetricPacket> {
    const iv = crypto.getRandomValues(new Uint8Array(12))

    const ciphertext = await crypto.subtle.encrypt({ name: this.ALG.name, iv }, key, data)

    return { ciphertext, iv }
  }

  async decrypt(key: CryptoKey, packet: SymmetricPacket): Promise<ArrayBuffer> {
    const { ciphertext, iv } = packet

    return crypto.subtle.decrypt({ name: 'AES-GCM', iv }, key, ciphertext)
  }
}
```

Лістинг encryption/hpke.service.ts

```
import type { HpkePacket } from '@/types/encryption.type.ts'
import type { IHpkeService } from '@/interfaces/encryption.interface.ts'

export class HpkeService implements IHpkeService {
  private readonly ALG = {
    kem: { name: 'ECDH', namedCurve: 'P-256' },
    kdf: { name: 'HKDF', hash: 'SHA-256' },
    aead: { name: 'AES-GCM', length: 256 },
  }

  private readonly LENGTHS = {
    key: 32, // 32 bytes for AES-256
    iv: 12, // 12 bytes standard IV
  }

  private async generateKeyPair(): Promise<CryptoKeyPair> {
    return crypto.subtle.generateKey(this.ALG.kem, true, ['deriveBits'])
  }

  async seal(data: BufferSource, recipientPublicKey: CryptoKey): Promise<HpkePacket> {
```

```
  const ephemeralKey = await this.generateKeyPair()

  const { aesKey, iv } = await this.deriveEncryptionContext(
    ephemeralKey.privateKey,
    recipientPublicKey,
  )

  const ciphertext = await crypto.subtle.encrypt({ name: this.ALG.aead.name, iv }, aesKey, data)

  const enc = await crypto.subtle.exportKey('spki', ephemeralKey.publicKey)

  return { ciphertext, enc }
}

async open(packet: HpkePacket, myPrivateKey: CryptoKey): Promise<ArrayBuffer> {
  const senderPublicKey = await crypto.subtle.importKey(
    'spki',
    packet.enc,
    this.ALG.kem,
    false,
    [],
  )

  const { aesKey, iv } = await this.deriveEncryptionContext(myPrivateKey, senderPublicKey)

  return crypto.subtle.decrypt({ name: this.ALG.aead.name, iv }, aesKey, packet.ciphertext)
}

private async deriveEncryptionContext(privateKey: CryptoKey, publicKey: CryptoKey) {
  const sharedSecretBits = await crypto.subtle.deriveBits(
    { name: this.ALG.kem.name, public: publicKey },
    privateKey,
    256,
  )

  const hkdfMaterial = await crypto.subtle.importKey(
    'raw',
    sharedSecretBits,
    this.ALG.kdf.name,
    false,
    ['deriveBits'],
  )

  const totalBitsNeeded = (this.LENGTHS.key + this.LENGTHS.iv) * 8

  const derivedBits = await crypto.subtle.deriveBits(
    {
      name: this.ALG.kdf.name,
      hash: this.ALG.kdf.hash,
      salt: new Uint8Array(), // Порожній salt
      info: new Uint8Array(), // Порожній info (контекст)
    },
```

```
      hkdfMaterial,
      totalBitsNeeded,
    )

    const keyBytes = derivedBits.slice(0, this.LENGTHS.key)
    const ivBytes = derivedBits.slice(this.LENGTHS.key, this.LENGTHS.key + this.LENGTHS.iv)

    const aesKey = await crypto.subtle.importKey('raw', keyBytes, this.ALG.aead.name, false, [
      'encrypt',
      'decrypt',
    ])

    return { aesKey, iv: new Uint8Array(ivBytes) }
  }
}
```

Лістинг encryption/hybrid-crypto.service.ts

```
import type { DigitalEnvelope } from '@/types/encryption.type.ts'
import type {
  IHpkeService,
  IHybridCryptoService,
  ISymmetricCryptoService,
} from '@/interfaces/encryption.interface.ts'

export class HybridCryptoService implements IHybridCryptoService {
  constructor(
    private symmetricCryptoService: ISymmetricCryptoService,
    private hpkeService: IHpkeService,
  ) {}

  async encrypt(
    data: BufferSource,
    recipients: { id: string; publicKey: CryptoKey }[],
  ): Promise<DigitalEnvelope> {
    const contentKey = await this.symmetricCryptoService.generateKey()
    const { ciphertext, iv } = await this.symmetricCryptoService.encrypt(contentKey, data)
    const contentKeyBytes = await this.symmetricCryptoService.exportKeyRaw(contentKey)
    const recipientBoxes = []

    for (const user of recipients) {
      const hpkePacket = await this.hpkeService.seal(contentKeyBytes, user.publicKey)
      recipientBoxes.push({
        recipientId: user.id,
        encryptedKey: hpkePacket.ciphertext,
        encapsulationData: hpkePacket.enc,
      })
    }

    return {
      ciphertext,
      iv,
```

```
      recipients: recipientBoxes,
    }
  }

  async decrypt(
    envelope: DigitalEnvelope,
    myId: string,
    myPrivateKey: CryptoKey,
  ): Promise<ArrayBuffer> {
    const myBox = envelope.recipients.find((r) => r.recipientId === myId)
    if (!myBox) throw new Error('Recipient box not found for the given ID')

    const contentKeyBytes = await this.hpkeService.open(
      {
        ciphertext: myBox.encryptedKey,
        enc: myBox.encapsulationData,
      },
      myPrivateKey,
    )

    const contentKey = await this.symmetricCryptoService.importKeyRaw(contentKeyBytes)

    return this.symmetricCryptoService.decrypt(contentKey, {
      ciphertext: envelope.ciphertext,
      iv: envelope.iv,
    })
  }
}
```

Лістинг encryption/index.ts

```
import { HpkeKeyService } from '@/services/encryption/hpke-key.service.ts'
import { KeyProtectionService } from '@/services/encryption/key-protection.service.ts'
import { SymmetricCryptoService } from '@/services/encryption/symmetric-crypto.service.ts'
import { HpkeService } from '@/services/encryption/hpke.service.ts'
import { HybridCryptoService } from '@/services/encryption/hybrid-crypto.service.ts'

const hpkeKeyServiceInstance = new HpkeKeyService()
const keyProtectionServiceInstance = new KeyProtectionService()
const symmetricCryptoServiceInstance = new SymmetricCryptoService()
const hpkeServiceInstance = new HpkeService()
const hybridCryptoServiceInstance = new HybridCryptoService(
  symmetricCryptoServiceInstance,
  hpkeServiceInstance,
)

export const hpkeKeyService = hpkeKeyServiceInstance
export const keyProtectionService = keyProtectionServiceInstance
export const symmetricCryptoService = symmetricCryptoServiceInstance
export const hpkeService = hpkeServiceInstance
export const hybridCryptoService = hybridCryptoServiceInstance
```