Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації

# Бакалаврська кваліфікаційна робота на тему:

«Захищений мобільний застосунок для зберігання конфіденційних даних»

Виконав: студент 4 курсу групи 1БС-21б
спеціальності 125 Кібербезпека
Віктор МУДРИЦЬКИЙ

Керівник: к. т. н., доцент каф. ЗІ
Володимир ГАРНАГА
«13» червня 2025 р.

Рецензент: к. т. н., доцент каф. ПЗ
Володимир МАЙДАНЮК
«13» червня 2025 р.

**Допущено до захисту**
Завідувач кафедри ЗІ
д. т. н., проф.
Володимир ЛУЖЕЦЬКИЙ
«13» червня 2025 р.

Вінниця ВНТУ – 2025 року

Міністерство освіти і науки України
Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра захисту інформації
Рівень вищої освіти І (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 125 Кібербезпека
Освітньо-професійна програма – Безпека інформаційних і комунікаційних систем

**ЗАТВЕРДЖУЮ**
Завідувач кафедри ЗІ, д. т. н., проф.
Володимир ЛУЖЕЦЬКИЙ
«21» березня 2025 року

**ЗАВДАННЯ**
**НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ**
Віктору МУДРИЦЬКОМУ

1. Тема роботи: «Захищений мобільний застосунок для зберігання конфіденційних даних»,
   керівник роботи: Володимир ГАРНАГА, к. т. н., доцент кафедри ЗІ,
   затверджені наказом ректора ВНТУ від 20 березня 2025 року за №97.
2. Строк подання студентом роботи 13 червня 2025 р.
3. Вихідні дані до роботи:
   - галузь застосування: захист персональних та конфіденційних даних у мобільних застосунках;
   - метод захисту: двофакторна автентифікація з використанням одноразових паролів на основі часу, шифрування даних;
   - середовище для реалізації підсистеми: мобільні операційні системи Android та IOS.
4. Зміст текстової частини: Вступ. Аналіз методів та засобів захисту конфіденційних даних у мобільних застосунках. Проєктування захищеного мобільного застосунку. Розробка та тестування захищеного мобільного застосунку. Висновки. Список використаних джерел. Додатки.
5. Перелік ілюстративного матеріалу: Схема безпечного передавання паролю на сервер, схема реалізації HOTP, схема реалізації TOTP, схема реалізації OTP, схема передавання паролів у системі LastPass, схема реєстрації користувача, схема входу користувача, схема генерації ключа в архітектурі додатку, схема передачі конфіденційних даних, схема роботи застосунку, результати тестування застосунку.

6. Консультанти розділів роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| 1 | Володимир ГАРНАГА, к.т.н., доц. каф.ЗІ | 24.03.25 | 10.06.25 |
| 2 | Володимир ГАРНАГА, к.т.н., доц. каф.ЗІ | 28.04.25 | 10.06.25 |
| 3 | Володимир ГАРНАГА, к.т.н., доц. каф.ЗІ | 30.05.25 | 10.06.25 |

7. Дата видачі завдання 21 березня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів бакалаврської дипломної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз завдання. Вступ | 24.03.25 – 30.03.25 | |
| 2 | Аналіз інформаційних джерел за напрямком бакалаврської кваліфікаційної роботи | 31.03.25 – 01.05.25 | |
| 3 | Розробка рішень, моделей, алгоритмів | 02.05.25 – 08.05.25 | |
| 4 | Практична реалізація, моделювання, експериментування, результати | 09.05.25 – 16.05.25 | |
| 5 | Аналіз виконання БКР, висновки | 17.05.25 – 18.05.25 | |
| 6 | Оформлення пояснювальної записки | 19.05.25 – 29.05.25 | |
| 7 | Попередній захист БКР | 30.05.25 – 02.06.25 | |
| 8 | Виправлення зауважень, перевірка на наявність текстових запозичень, підготовка ілюстративного матеріалу | 03.06.25 – 16.06.25 | |
| 9 | Представлення БКР до захисту, рецензування | 17.06.25 – 19.06.25 | |

Студент _____ Віктор МУДРИЦЬКИЙ

Керівник роботи _____ Володимир ГАРНАГА

# АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 66 сторінок формату А4, на яких розміщено 22 рисунки, 5 таблиць. Список використаних джерел містить 25 найменувань.

Робота присвячена підвищенню рівня безпеки персональних даних у мобільних застосунках. Проведено аналіз сучасних підходів до захисту інформації та обґрунтовано вибір архітектурних рішень для забезпечення конфіденційності користувацьких даних. У межах дослідження розроблено мобільний застосунок, що реалізує основні механізми захисту, які відповідають актуальним вимогам кібербезпеки. Проведено перевірку працездатності запропонованого рішення та оцінено його ефективність.

Ключові слова: одноразові паролі, мобільний застосунок, захист даних, автентифікація, конфіденційність.

# ABSTRACT

The bachelor's qualification thesis consists of 66 A4-sized pages, including 22 figures and 5 tables. The list of references contains 25 sources.

The thesis is devoted to enhancing the security of personal data in mobile applications. The study analyzes modern approaches to information protection and justifies the selection of architectural solutions aimed at ensuring the confidentiality of user data. A mobile application was developed within the research, implementing essential protection mechanisms in accordance with current cybersecurity requirements. The functionality of the proposed solution was tested and its effectiveness was evaluated.



Keywords: one-time passwords, mobile application, data protection, authentication, confidentiality.

# ЗМІСТ

# ВСТУП

У сучасному цифровому середовищі мобільні застосунки набули широкого розповсюдження як інструмент для зберігання та обробки персональних і конфіденційних даних. Разом із цим зростає й кількість кіберзагроз, спрямованих на отримання несанкціонованого доступу до цієї інформації. Забезпечення належного рівня захисту стає критично важливим завданням як для розробників, так і для користувачів програмного забезпечення.

Актуальність теми дослідження зумовлена необхідністю впровадження ефективних заходів безпеки в мобільних застосунках, які відповідають сучасним вимогам до конфіденційності, цілісності та доступності даних. Особливого значення набувають методи автентифікації, шифрування та обмеження доступу, які повинні забезпечувати надійний захист інформації навіть у разі фізичної втрати пристрою чи компрометації облікових даних.

Метою роботи є підвищення рівня захисту персональних даних у мобільному застосунку шляхом реалізації засобів, що відповідають сучасним принципам інформаційної безпеки.

Для досягнення поставленої мети необхідно виконати такі основні завдання:

- провести аналіз існуючих підходів до захисту даних у мобільному середовищі;
- обґрунтувати вибір архітектурних і програмних рішень;
- реалізувати мобільний застосунок з урахуванням вимог до безпеки;
- оцінити ефективність розробленого рішення.

Об’єктом дослідження є процес захисту даних у мобільному застосунку.

Предметом дослідження є програмні та архітектурні рішення, що забезпечують захист конфіденційної інформації під час використання мобільного застосунку.

# 1 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ЗАХИСТУ КОНФІДЕНЦІЙНИХ ДАНИХ У МОБІЛЬНИХ ЗАСТОСУНКАХ

Із швидким розвитком мобільних технологій питання захисту конфіденційної інформації набуває особливої актуальності. Мобільні застосунки дедалі частіше використовуються для зберігання персональних даних, паролів, банківської інформації, медичних записів та іншої чутливої інформації, що робить їх ціллю атак.

Захист даних у мобільних застосунках має охоплювати всі етапи: автентифікацію користувача, захист інформації під час передавання, шифрування даних перед збереженням, а також обмеження доступу до цих даних. Особливої уваги потребує зберігання даних на віддалених серверах, потрібно гарантувати, що конфіденційні дані недоступні стороннім сторонам, включаючи власників системи. Для цього вони мають зберігатись лише в зашифрованому вигляді.

Цей розділ присвячено аналізу актуальних загроз, методів автентифікації, шифрування, а також порівнянню з існуючими рішеннями. Особливий акцент зроблено на шифруванні даних перед їх передаванням на сервер, використанні двофакторної автентифікації, а також методах генерації ключів із паролів для забезпечення криптостійкості.

## 1.1 Особливості обробки конфіденційної інформації у мобільному середовищі

При розробці захищених мобільних застосунків, одним з ключових архітектурних рішень є вибір між локальним і віддаленим зберіганням конфіденційних даних. Обидва підходи мають свої особливості та ризики, однак у системах, орієнтованих на роботу з декількома платформами та централізовану синхронізацію, віддалене зберігання переважає за гнучкістю.

Локальне зберігання передбачає запис даних у сховище самого мобільного пристрою. Вони можуть зберігатись у локальній базі даних, файловій системі або за допомогою спеціалізованих системних сховищ. Основною перевагою такого

підходу є відсутність потреби у мережевому з'єднанні, що робить дані доступними без інтернет під'єднання, а ризики перехоплення під час передавання відсутні. Ключовими недоліками локального зберігання даних є:

- залежність від фізичної безпеки пристрою;
- високі вимоги до правильного зберігання ключів;
- обмежена масштабованість, а саме проблема синхронізації даних на різних пристроях.

Віддалене зберігання, реалізоване через хмарні сервіси, забезпечує централізований доступ до даних, однак при цьому виникають такі ризики:

- незашифровані дані можуть бути прочитані власниками системи;
- компрометація сервера або бази даних;
- ненадійна автентифікація користувача, що може призвести до отримання доступу до акаунту зловмисником.

Обробка конфіденційної інформації у мобільному середовищі є складним завданням, яке потребує врахування особливостей платформи, поведінки користувача та сучасних кіберзагроз. Вибір між локальним і віддаленим зберіганням даних безпосередньо впливає на архітектуру системи безпеки. Хоча локальне зберігання дозволяє уникнути ризиків, пов'язаних із передачею даних і компрометацією серверів, воно є менш гнучким та уразливим у разі фізичного доступу до пристрою [1].

## 1.2 Методи автентифікації користувачів у мобільних застосунках

Автентифікація користувача є критично важливим етапом у забезпеченні доступу до конфіденційних даних у мобільному застосунку. Вона визначає, чи має особа право отримати доступ до інформації, яка зберігається у системі. У даному підрозділі розглянуто найпопулярніші методи автентифікації користувачів, які є доцільними для мобільних додатків (табл. 1.1) [2]. Їх вибір залежить від рівня захисту, який необхідно забезпечити, з урахуванням можливостей пристрою, досвіду користувача та потенційних ризиків. Поєднання кількох методів, як пароль та одноразовий код, значно підвищує безпеку доступу до чутливої інформації.

Таблиця 1.1 – Популярні методи автентифікації для мобільних додатків

| Метод автентифікації | Опис методу | Реалізація | Особливості |
|---|---|---|---|
| Пароль | Користувач вводить логін та пароль | Пошта та пароль | Слабкий захист, піддається фішингу |
| Біометрія | Ідентифікація на основі фізичних характеристик користувача | Відбиток пальця, ідентифікація обличчя | Прив’язка до пристрою, не підходить для віддаленого зберігання |
| Одноразовий часозалежний пароль TOTP | Генерація кодів кожні на основі часу та секретного ключа | Користувач використовує програму аутентифікатор | Сучасний метод, який використовується для багатофакторної автентифікації |
| Одноразовий пароль OTP | Код дійсний лише для одного сеансу | Код отриманий через пошту або повідомлення | Поширений, але менш безпечний через можливість перехоплення |
| Багатофакторна автентифікація MFA | Комбінація 2 і більше методів з різних категорій | Пароль та одноразовий пароль | Надійна модель, підвищує складність атак |

У таблиці 1.2 проведено оцінку однофакторної та двофакторної автентифікації. Дана оцінка є важливим фактором вибору для обрання найбільш доцільного методу. Рівні оцінки поділяються на:

- відсутній;
- низький;
- середній;
- високий;
- повний.

Такий підхід дозволяє порівняти методи автентифікації за ключовими критеріями безпеки, зручності та сумісності з мобільними платформами.

Таблиця 1.2 – Оцінка однофакторної та двофакторної автентифікації

| Критерій | Однофакторна автентифікація | Двофакторна автентифікація |
|---|---|---|
| Рівень безпеки | Низький | Високий |
| Захист від атак методом перебору | Низький | Наявний |
| Стійкість до фішингових атак | Низька | Висока |
| Залежність від пароля | Повна | Середня |
| Залежність від додаткового пристрою | Відсутня | Присутня |
| Складність реалізації | Низька | Середня |
| Зручність для користувача | Висока | Середня |
| Доцільність використання в системах, що обробляють конфіденційні дані | Низька | Висока |

Отже, аналіз існуючих методів автентифікації показує, що традиційна однофакторна автентифікація за паролем є вразливою до атак соціальної інженерії, фішингу та перебору. У зв’язку з цим сучасні системи все частіше впроваджують багатофакторні моделі, які об’єднують щонайменше два незалежні фактори перевірки.

Найбільш збалансованим рішенням для мобільного середовища є використання TOTP як другого фактору автентифікації у поєднанні з паролем. Такий підхід забезпечує високий рівень захисту, дозволяє користувачеві зберігати контроль над пристроєм автентифікації та значно підвищує стійкість системи до фішингових атак.

## 1.3 Теоретичні основи двофакторної автентифікації в мобільних застосунках

В умовах зростаючої кількості кібератак, спрямованих на мобільні платформи, впровадження двофакторної автентифікації стає стандартною вимогою для застосунків, що обробляють персональні або фінансові дані користувачів.

Водночас ефективність цієї технології залежить від правильного вибору типу автентифікації, її реалізації на рівні застосунку та зручності для кінцевого користувача.

1.3.1 Пароль та електронна пошта як перший фактор автентифікації

Зберігання паролів у безпечному вигляді є критично важливим аспектом побудови будь-якої системи, що обробляє автентифікацію користувачів. Найкращі практики зберігання паролів у базі даних:

- використовування криптографічної хеш-функції для перетворення пароля у рядок фіксованого розміру. Цей рядок є унікальним для кожного пароля і не може бути обернений для відновлення оригінального пароля. Таким чином при компрометації серверу чи бази даних паролі залишаються невідомими;
- використання спеціалізованих алгоритмів для хешування паролів, що робить їх стійкими до атак методом повного перебору;
- використання доцільного фактору складності, значення, яке визначає обчислювальні витрати на хешування пароля. Збільшуючи цей фактор, можливість здійснити успішну атаку методом перебору зменшується;
- використання солі, випадкового значення, яке додається до пароля перед його хешуванням. Це робить кожен хеш унікальним, навіть для однакових паролів, і запобігає використанню попередньо обчислених таблиць хешів для зламу [3].

Схема хешування та відправки паролю на сервер представлена на рисунку 1.1.


Рисунок 1.1 – Безпечне передавання паролю на сервер

Автентифікація на основі електронної пошти та пароля є базовим механізмом підтвердження особи, який широко застосовується в мобільних застосунках. Проте ефективність цього підходу значною мірою залежить від дотримання вимог до

складності паролів, а також від правильного та захищеного зберігання облікових даних. У сучасних системах, особливо в тих, що обробляють конфіденційну інформацію, парольна автентифікація повинна розглядатись лише як перший рівень безпеки, який має доповнюватися другим незалежним фактором.

### 1.3.2 Одноразові паролі як механізм реалізації другого фактору автентифікації

Одноразові паролі є ефективним засобом автентифікації, який забезпечує суттєво вищий рівень захисту в порівнянні з класичними паролями. Вони призначені для використання лише один раз, а саме, у межах однієї сесії або короткого проміжку часу. Це дозволяє ефективно протидіяти таким загрозам, як фішинг, повторне використання паролів, атаки перебором тощо.

У межах стандартів IETF виділяють дві основні реалізації одноразових паролів: HOTP та TOTP. Обидва стандарти базуються на криптографічному алгоритмі HMAC, однак відрізняються логікою генерації паролів [4].

Алгоритм HOTP визначено стандартом RFC 4226. Він генерує OTP коди на основі лічильника, який збільшується при кожному запиті. І клієнт, і сервер мають підтримувати синхронізовані значення лічильника для успішної перевірки (рис. 1.2). На базовому рівні одноразовий пароль за алгоритмом HOTP діє довше, ніж TOTP, що частково компенсує проблеми з доставкою або інші логістичні труднощі, які можуть зробити використання TOTP незручним або неможливим. Це вважається зручністю для кінцевих користувачів, оскільки вони мають більше часу на вхід до зміни OTP. Проте така зручність має свій недолік: HOTP має вбудовану вразливість, а саме триваліший час дії створює більше можливостей для атак перебором або подібних атак. Це робить HOTP менш придатним для сценаріїв, де потрібна висока стійкість до атак та гарантія обмеженого часового для використання одноразового коду.

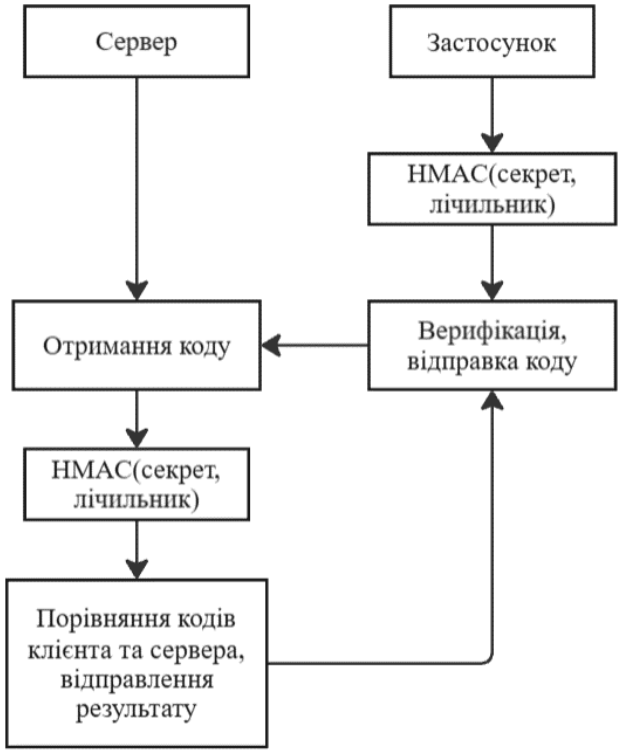



Рисунок 1.2 – Схема реалізації HOTP

TOTP – це стандарт генерації одноразових паролів, який використовує поточний час як змінну замість лічильника, на відміну від HOTP. Під час автентифікації клієнт і сервер незалежно обчислюють OTP код, використовуючи однаковий секретний ключ та синхронізоване значення часу, отримане шляхом ділення Unix часу на фіксований часовий інтервал. На стороні клієнта TOTP код генерується і відображається у застосунку аутентифікатору. Користувач вводить код, а сервер у момент перевірки повторно обчислює очікуване значення для поточного інтервалу часу (рис. 1.3). Якщо значення збігаються, автентифікація вважається успішною. Перевагою TOTP є відсутність потреби у синхронізації лічильників і обмежений строк дії коду, що зменшує ризик повторного використання пароля при його перехопленні [5]. Завдяки цьому TOTP є більш стійким до атак повторного відтворення, порівняно з алгоритмом HOTP. Крім того, його використання широко підтримується у сучасних системах автентифікації, зокрема в мобільних аутентифікаторах та багатофакторних застосунках.

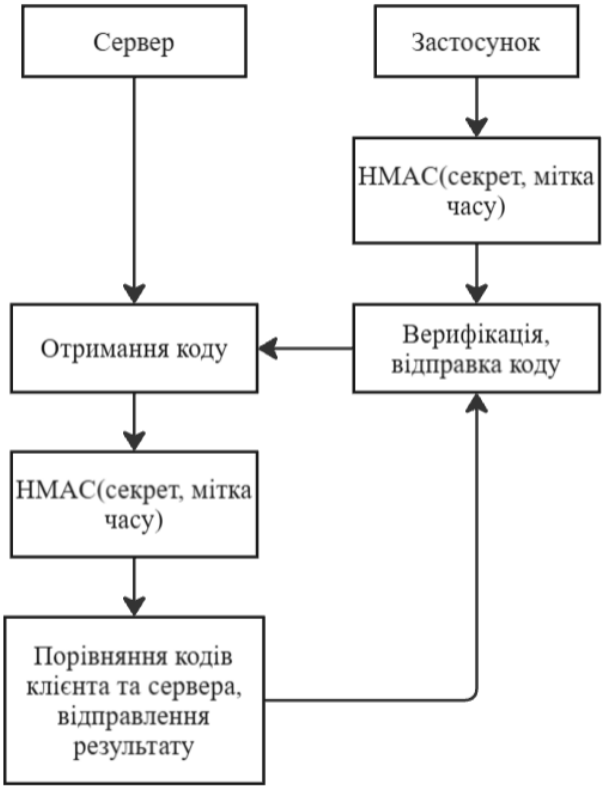



Рисунок 1.3 – Схема реалізації TOTP

OTP одноразовий пароль – це згенерований сервером код, який діє лише для одного входу і використовується для підтвердження особи користувача. Клієнти отримують цей токен електронною поштою або через SMS і вводять його у форму входу, щоб отримати доступ до свого облікового запису. Одноразові паролі, що мають обмежений термін дії і використовуються лише один раз, замінюють статичні паролі, забезпечуючи вищий рівень захисту від шахрайства та витоків даних [6]. Схему верифікації через OTP код, використовуючи пошту або телефонні повідомлення представлено на рисунку 1.4. Цей метод є особливо корисним у випадках, коли інші способи автентифікації є недоступними або ненадійними. Проте його безпека залежить від захищеності каналів доставки, таких як телефонні повідомлення або електронна пошта. Зловмисники можуть використати методи соціальної інженерії або перехоплення повідомлень, щоб отримати доступ до OTP і здійснити несанкціонований вхід.

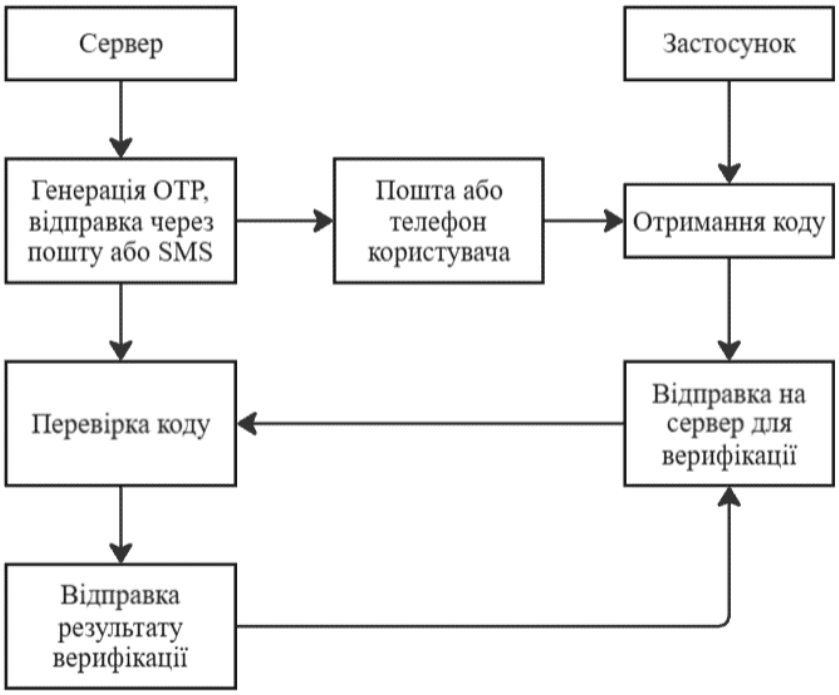



Рисунок 1.4 – Схема реалізації OTP

Щоб визначитись із підходящим методом, доцільно створити таблицю та оцінити фактори, які найкраще підходять для мобільного застосунку (табл. 1.3).

Таблиця 1.3 – Оцінка варіацій OTP кодів

| Характеристика | HOTP | TOTP | OTP |
|---|---|---|---|
| Стандарт | RFC 4226 | RFC 6238 | Відсутній |
| Базовий параметр | Лічильник | Поточний час | Серверна генерація |
| Потреба у синхронізації | Так | Ні | Ні |
| Термін дії | Необмежений | 30–60 секунд | 3–5 хвилин |
| Генерація на пристрої | Так | Так | Ні |
| Залежність від мережі | Ні | Ні | Так |
| Захист від фішингу | Помірний | Високий | Низький |
| Поширеність у мобільних додатках | Обмежена | Дуже висока | Середня |
| Застосування | Апаратні токени | Застосунки аутентифікатори | Пошта, телефонні повідомлення |

Одноразові паролі є ефективним інструментом підвищення рівня безпеки автентифікації, особливо в умовах зростаючих загроз фішингу та компрометації облікових даних. Із двох основних стандартів HOTP та TOTP, останній є найбільш придатним для використання в мобільних застосунках. Завдяки прив'язці до поточного часу, він не потребує синхронізації лічильників, забезпечує короткий строк дії коду та зберігає криптографічну стійкість завдяки використанню HMAC функцій.

На практиці TOTP дозволяє створити надійний другий фактор автентифікації без необхідності доступу до мережі, що робить його незалежним від сторонніх каналів зв'язку. Завдяки широкій підтримці з боку відкритих бібліотек, простоті реалізації та високій сумісності з мобільними платформами, TOTP залишається одним із найкращих рішень для впровадження багатофакторної автентифікації в сучасних інформаційних системах, орієнтованих на захист конфіденційних даних.

## 1.4 Методи шифрування даних у мобільних застосунках

Захист конфіденційної інформації у мобільних застосунках неможливий без застосування надійних криптографічних методів. Шифрування даних дозволяє зробити інформацію недоступною для сторонніх осіб навіть у разі її перехоплення або компрометації середовища зберігання. У мобільних системах найбільш поширеними є два підходи: симетричне та асиметричне шифрування. Кожен із них має свої особливості, переваги та сфери застосування, що зумовлює вибір відповідної криптографічної моделі залежно від вимог безпеки.

### 1.4.1 Симетричне шифрування та виведення ключа з пароля

Симетричне шифрування є одним із найбільш ефективних способів захисту даних, при якому один і той самий секретний ключ використовується як для шифрування, так і для дешифрування інформації. Основною перевагою симетричного підходу є його висока швидкість та оптимізація, що робить його хорошим варфантом для використання у ресурсозалежних середовищах, таких як мобільні пристрої.

У мобільних застосунках симетричне шифрування найчастіше використовується для локального шифрування перед відправкою на віддалений сервер. Таким чином, навіть якщо дані будуть перехоплені з бази даних, їх зміст залишиться недоступним без відповідного ключа шифрування [7].

Переваги та недоліки симетричного шифрування показано у таблиці 1.4.

Таблиця 1.4 – Недоліки та переваги симетричного шифрування

| Критерій | Переваги | Недоліки |
|---|---|---|
| Продуктивність | Висока швидкість, мінімальне навантаження | Потрібно зберігати ключ шифрування секретно |
| Реалізація на мобільних платформах | Простота реалізації, багато існуючих рішень | Потребує захищеного середовища для зберігання ключа |
| Контроль користувача над даними | Дані шифруються на стороні застосунку, сервер отримує зашифрований вміст | Вся відповідальність за безпеку ключа лежить на стороні застосунку |
| Захист при витоку даних | Дані на сервері зберігаються у зашифрованому вигляді | У разі витоку ключа всі дані компрометуються |

Щоб не примушувати користувача пам’ятати ключ шифрування, використовуються методи виведення ключа із паролю за допомогою криптографічної функції. Найбільш поширеними є функції є PBKDF2 та HKDF, які реалізують багаторазове хешування пароля в поєднанні з випадковою сіллю та великою кількістю ітерацій. Це суттєво ускладнює перебір можливих варіантів ключа, навіть якщо зловмисник отримав доступ до значення солі та параметрів генерації. У результаті генерується криптографічно стійкий ключ фіксованої довжини, який може бути використаний, наприклад, для симетричного шифрування. Така модель дозволяє поєднати зручність для користувача з високим рівнем захисту, зводячи до мінімуму ризик витоку ключа через зберігання або передачу. Крім того, використання функцій виведення ключа забезпечує незалежність криптографічного захисту від конкретного пристрою або операційної системи [8].

### 1.4.2 Асиметричне шифрування у мобільному середовищі

Асиметричне шифрування є фундаментальною технологією сучасної криптографії, що базується на використанні пари ключів: відкритого та закритого. Відкритий ключ може бути вільно поширений і застосовується для шифрування або перевірки підпису, тоді як закритий ключ зберігається у захищеному середовищі та використовується для дешифрування або створення цифрового підпису. Такий підхід дозволяє побудувати механізми захищеного обміну інформацією без необхідності попереднього погодження спільного секрету.

Типові алгоритми включають в себе: RSA та Elliptic Curve Cryptography. RSA забезпечує високу криптостійкість при великих розмірах ключів, а ECC – при менших розмірах, що є перевагою в мобільному середовищі.

У мобільних системах асиметричне шифрування не використовується для шифрування великих масивів даних через низьку швидкість обробки. Однак воно незамінне для задач автентифікації, цифрового підпису, обміну ключами та встановлення захищених каналів [9]. У таблиці 1.5 представлені переваги та недоліки асиметричного шифрування у мобільному середовищі.

Таблиця 1.5 – Недоліки та переваги асиметричного шифрування

| Критерій | Переваги | Недоліки |
|---|---|---|
| Модель ключів | Відкритий ключ може передаватись безпечно | Закритий ключ потребує надійного зберігання |
| Безпека обміну даними | Дозволяє шифрувати без попереднього обміну секретом | Шифрування великих обсягів даних є непрактичним |
| Використання в автентифікації | Підтримує цифровий підпис, автентифікацію без пароля | Ускладнена реалізація |
| Швидкість | Використовується для шифрування маленького об’єму даних | Повільне при багаторазовому застосуванні |
| Використання в мобільних додатках | Підходить для встановлення сеансів, перевірки даних | Непридатне для безпосереднього шифрування файлів чи великих повідомлень |

Отже, симетричне шифрування забезпечує швидкий та ефективний захист даних у мобільних застосунках, особливо при локальній реалізації. Генерація ключа з пароля за допомогою криптографічних спеціалізованих алгоритмів дозволяє уникнути його зберігання у відкритому вигляді та підвищує стійкість до атак. Такий підхід є оптимальним для шифрування перед передачею даних на сервер.

### 1.5 Порівняльний аналіз існуючих рішень

Для порівняння обрано додаток LastPass. З офіційного сайту отримано інформацію про те, як відбувається шифрування та менеджмент паролів. LastPass – це один із найпопулярніших менеджерів паролів, що базується на моделі нульового розголошення [10]. Це означає, що жоден сервер LastPass не має доступу до незашифрованих даних користувача, включно з паролями. Усі критично важливі операції з шифрування та розшифрування відбуваються локально на пристрої користувача, наприклад у браузері або мобільному застосунку. Схему зберігання та передавання паролів у системах LastPass представлено на рисунку 1.5.

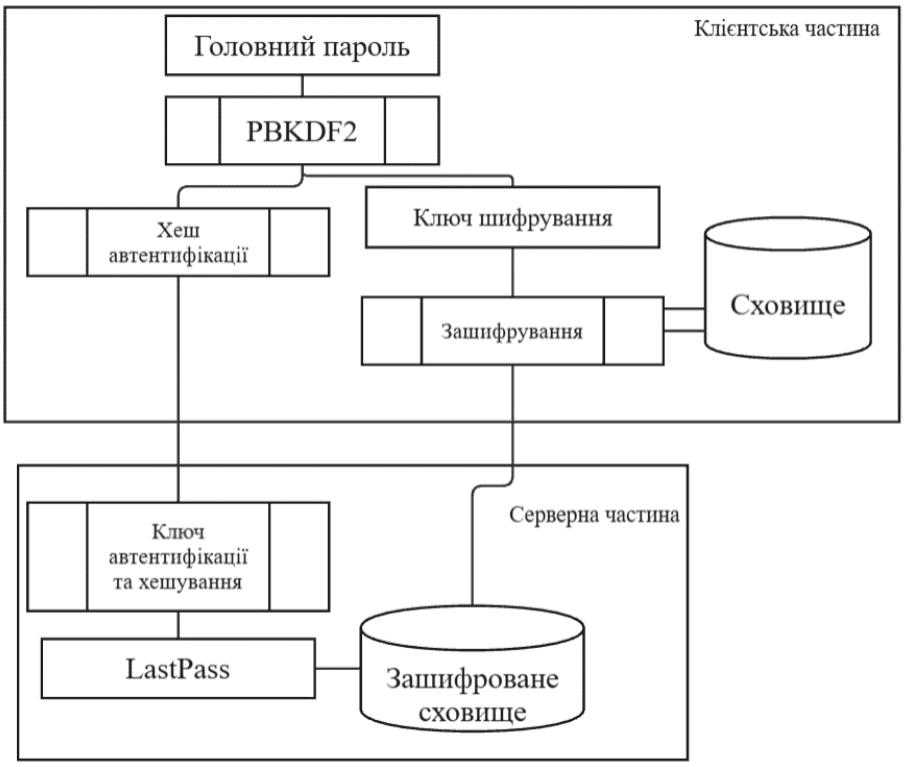



Рисунок 1.5 – Схема передавання паролів у системі LastPass

Ключові операції, які використовуються у системах LastPass:

- локальне шифрування на пристрої користувача;
- паролі та інші збережені дані шифруються на пристрої до надсилання в хмару. Усі операції з шифрування та розшифрування виконуються локально;
- усі дані шифруються за допомогою алгоритму AES-256, одного з найстійкіших стандартів;
- головний пароль не зберігається і не передається, з нього на пристрої генерується майстер-ключ за допомогою PBKDF2 з SHA-256, із понад 100000 ітерацій. Це суттєво ускладнює атаки перебором;
- у хмарі зберігаються лише зашифровані дані. Без майстер-ключа, який відомий лише користувачу, ці дані неможливо розшифрувати навіть адміністраторам сервісу;
- LastPass підтримує TOTP, біометрію та інші методи додаткової перевірки особи, що забезпечує захист навіть у разі витоку пароля;
- на мобільних пристроях можна використовувати ідентифікацію обличчя або відбиток пальця для розблокування сховища, не передаючи при цьому головний пароль.

Отже, LastPass забезпечує високий рівень захисту паролів шляхом локального шифрування, криптографічно стійкої генерації ключів, двофакторної автентифікації та збереження лише зашифрованих даних у хмарі. Модель нульового розголошення гарантує, що ніхто, крім самого користувача, не може отримати доступ до його конфіденційної інформації.

### 1.6 Висновки до розділу

У межах першого розділу було здійснено аналіз сучасних методів та засобів захисту персональних даних у мобільних застосунках, що дозволило сформувати теоретичну базу для розробки власного захищеного застосунку. Основний акцент було зроблено на дослідженні механізмів автентифікації користувача та криптографічному захисту інформації, які є базовими складовими ефективної системи інформаційної безпеки в мобільному середовищі.

Особливу увагу приділено реалізації двофакторної автентифікації, що базується на поєднанні фактору знання, наприклад пароль, і фактору володіння, наприклад додаток, який генерує TOTP. Такий підхід забезпечує суттєве підвищення рівня стійкості системи до несанкціонованого доступу та водночас є зручним для впровадження в межах мобільних платформ без створення надмірного навантаження на користувача або інфраструктуру.

У рамках аналізу методів шифрування обґрунтовано доцільність використання симетричних алгоритмів, як оптимальних для застосування на стороні застосунку. Такий вибір базується на високій швидкості та ефективності симетричних алгоритмів, що є критично важливим у мобільному середовищі. При цьому ключовою перевагою обраного підходу є використання процедури виведення криптографічного ключа з пароля користувача за допомогою спеціалізованих криптографічних функцій. Це дозволяє уникнути необхідності зберігання ключів на сервері, істотно знижуючи ризики їх компрометації у разі зламу або втрати контролю над інфраструктурою. Окремо розглянуто потенціал використання асиметричного шифрування як додаткового інструменту.

Крім того, в межах розділу було проаналізовано типові загрози, що виникають при обробці персональних даних у мобільних застосунках. Виявлено, що основними векторами атаки залишаються як мережеві ризики, так і локальні загрози, пов’язані з втратою або крадіжкою пристрою. Результати аналізу стали основою для формування цілісного підходу до побудови архітектури мобільного застосунку, у якій кожен компонент взаємодіє з іншими через безпечні канали та використовує принцип найменших привілеїв.

Узагальнюючи результати проведеного аналізу, можна стверджувати, що сформована концептуальна модель захисту даних у мобільному застосунку базується на поєднанні перевірених практик автентифікації користувача та незалежного локального шифрування даних. Такий підхід забезпечує високий рівень конфіденційності інформації навіть у випадку компрометації серверної частини системи та створює надійну основу для подальшої реалізації захищеного мобільного застосунку.

# 2 ПРОЄКТУВАННЯ ЗАХИЩЕНОГО МОБІЛЬНОГО ЗАСТОСУНКУ

У цьому розділі розглянуто технічні аспекти побудови захищеного мобільного застосунку для зберігання конфіденційних даних. Проєктування такої системи вимагає ретельного врахування особливостей мобільного середовища, специфіки обробки персональної інформації, а також стійкості до сучасних кіберзагроз.

На основі аналізу методів автентифікації та шифрування, проведеного у першому розділі, сформульовано ключові вимоги до архітектури застосунку. А саме, впровадження двофакторної автентифікації, а також використання криптографічно стійких алгоритмів на стороні застосунку для забезпечення конфіденційності збережених даних.

Розділ включає опис архітектурних рішень, обґрунтування вибору криптографічних механізмів, особливості зберігання даних у зашифрованому вигляді та реалізацію перевірки автентичності користувача. Також виконано оцінку ризиків, пов’язаних із потенційними векторами атак, з урахуванням моделі загроз, характерної для мобільних застосунків.

## 2.1 Вимоги до архітектури захищеного мобільного застосунку

Захищений мобільний застосунок, який обробляє конфіденційні персональні дані, має задовольняти комплекс вимог щодо безпеки, обґрунтованих сучасними кіберзагрозами. У попередньому розділі було висвітлено ризики фішингу, атак «людина посередині», перехоплення даних при передачі, атак на локальні сховища, а також компрометації облікових записів. Щоб запобігти таким загрозам, архітектура реалізованого застосунку відповідає наступним принципам:

- багатофакторна автентифікація. Реалізовано автентифікацію через пошту та пароль, а також обов’язкову перевірку TOTP-коду при кожному запуску [11]. Цей підхід відповідає найкращим практикам, визначеним NIST;
- постійна автентифікація при кожному запуску. Повна повторна перевірка при запуску мобільного застосунку забезпечує цілісність сесійного контексту

й виключає сценарії зловмисного відновлення активного сеансу після блокування пристрою;

- локальне шифрування даних з використанням ChaCha20-Poly1305. Цей алгоритм рекомендований є IETF стандартом RFC 8439, що забезпечує як конфіденційність, так і автентичність даних. Його продуктивність на ARM-пристроях дозволяє уникнути затримок при шифруванні великих обсягів у мобільному застосунку [12];
- HKDF для деривації ключа з пароля. Застосування даного алгоритму гарантує стійку генерацію ключа локально без потреби зберігати чи передавати сам ключ [13];
- використання Firebase як інфраструктурної платформи. Firebase забезпечує готові засоби автентифікації та керування користувачами з підтримкою MFA. Така модель відповідає вимогам до побудови безпечної інфраструктури, в якій сервер не має доступу до зашифрованих даних користувача [11].

Дані рішення забезпечують комплексний захист застосунку на критичних етапах автентифікації, зберігання та обробки даних. Вони відповідають вимогам, щодо підвищення стійкості до атак на автентифікацію, перехоплення інформації та забезпечують, що дані.

## 2.2 Проєктування автентифікації з двофакторною перевіркою

Для забезпечення захищеного доступу до мобільного застосунку обрано двофакторну модель автентифікації, яка базується на поєднанні Firebase Authentication через пошту та пароль і одноразових паролів за стандартом TOTP. Такий підхід забезпечує стійкість до більшості типових атак на автентифікацію, зокрема фішинг, перебір паролів та повторне використання сесій.

Перший етап автентифікації реалізовано за допомогою сервісу Firebase Authentication, який підтримує автентифікацію з поштою і паролем. Після успішного входу Firebase створює ідентифікаційний токен у форматі JWT, який використовується застосунком для взаємодії з іншими сервісами Firebase. Цей токен містить закодовану інформацію про користувача: його ідентифікатор, пошту,

час видачі, термін дії, а також додаткові дані. Токен підписано цифровим підписом самим сервером автентифікації, що гарантує його автентичність [14].

Як другий фактор реалізовано перевірку коду TOTP, що генерується на основі спільного секретного ключа. Для цього використано інтеграцію з платформою Twilio, яка дозволяє створити унікальний секрет для кожного користувача та генерувати QR код для зручного додавання у застосунки аутентифікатори [15]. Перевірка TOTP коду виконується на сервері через хмарну функцію, що звіряє введений код з очікуваним значенням на основі системного часу. Схему реєстрації користувача зображено на рисунку 2.1.

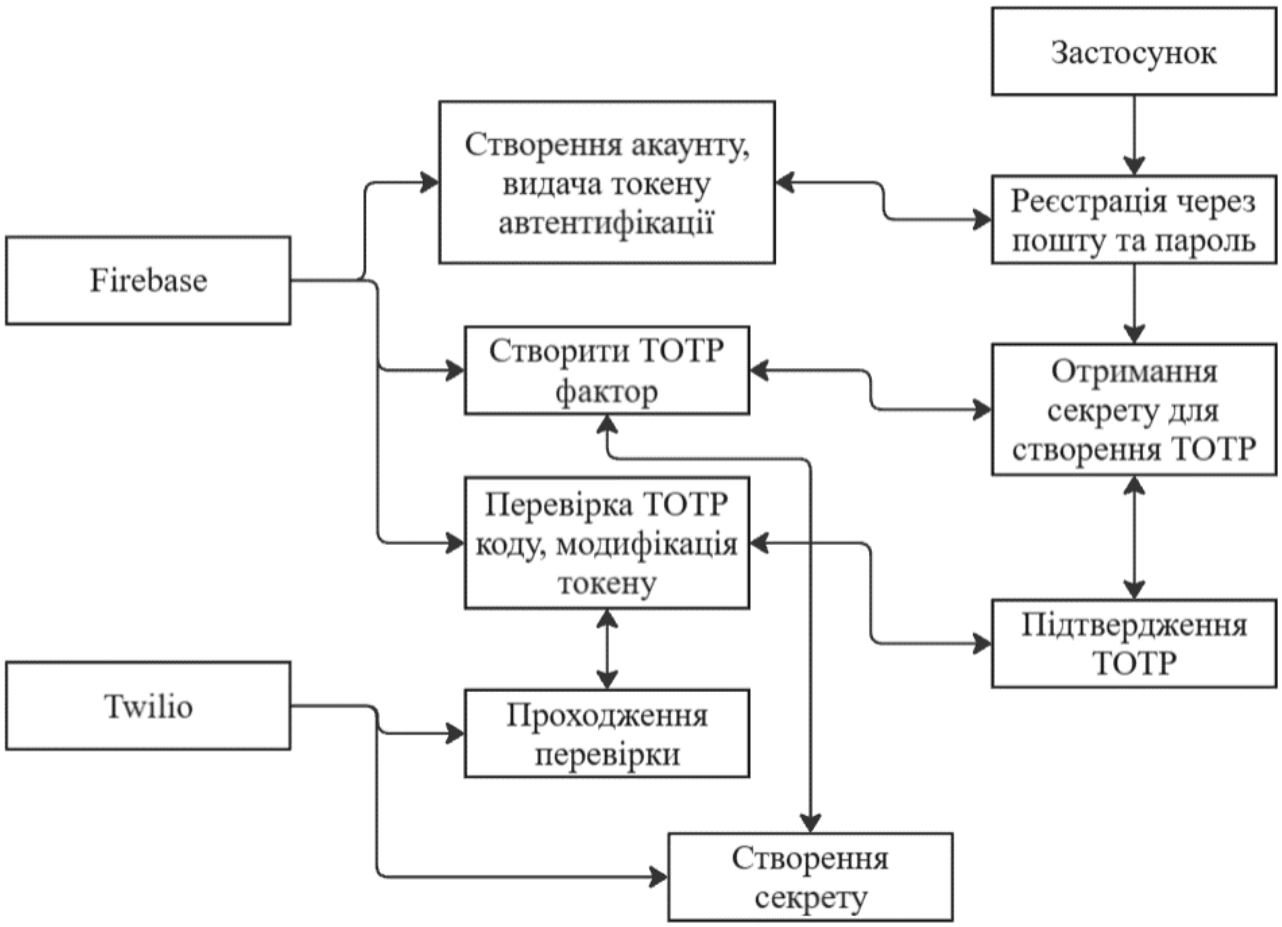



Рисунок 2.1 – Схема реєстрації користувача

Основною відмінністю входу від реєстрації є відсутність створення TOTP фактору, замість цього користувач одразу підтверджує код. Схему входу для існуючого користувача зображено на рисунку 2.2.


Рисунок 2.2 – Схема входу користувача

Обрана схема не передбачає збереження стану автентифікації або токенів у незашифрованому вигляді. Це забезпечує жорсткий контроль доступу до зашифрованих даних.

Таким чином, комбінація Firebase Authentication та Twilio TOTP дає змогу швидко впровадити надійний механізм автентифікації з мінімальними ризиками реалізаційних помилок і високим ступенем ізоляції даних користувача.

### 2.3 Обґрунтування вибору засобів шифрування даних

Однією з ключових вимог до безпечного мобільного застосунку є захист конфіденційних даних, що зберігаються на пристрої. У процесі проєктування розглядалися два сучасних AEAD алгоритми: ChaCha20-Poly1305 та AES-256-GCM. Обидва забезпечують одночасне шифрування і перевірку цілісності даних.

AES – це блочний шифр з фіксованим розміром блоку 128 біт, а GCM – це режим роботи, який забезпечує автентифікацію даних. AES-256-GCM є надійним і

широко використовуваним стандартом, підтримується апаратно на більшості пристроях [16].

ChaCha20 – це потоковий шифр, розроблений Деніелом Бернштейном як безпечна та швидка альтернатива RC4. Він працює з 256-бітним ключем, 96-бітовим одноразовим номером та 32-бітовим лічильником блоку. У кожному раунді виконується перестановка і додавання ключових слів у 512-бітному внутрішньому стані, після чого генерується псевдовипадковий потік байтів, до якого застосовується XOR операція з відкритим текстом. Завдяки простоті та відсутності залежності від апаратної реалізації, ChaCha20 демонструє стабільну швидкість навіть на слабких процесорах. У комбінації з Poly1305 для автентифікації, ChaCha20 утворює повноцінну AEAD-схему, яка використовується в TLS 1.3, SSH, WireGuard, та підтримується більшістю сучасних бібліотек [17].

Після аналізу було обрано саме ChaCha20-Poly1305, з огляду на його переваги в мобільному середовищі, а саме швидкості, зручності використання та меншого навантаження на процесор ніж AES.

ChaCha20-Poly1305 працює за наступною схемою:

- перша сесія ChaCha20 використовується для шифрування заголовка з використанням перших 32 байтів ключа та одноразовим номером, який є номером пакета. Значення номеру блоку при цьому встановлюється в 0;
- друга сесія ChaCha20 застосовується для генерації ключа для Poly1305. Для цього використовується остання частина ключа, а саме 32 байти, вхідні дані, які спочатку ініціалізуються нулями, а для подальшого використання зберігаються перші 32 байти вихідного потоку;
- друга сесія ChaCha20 використовується з номером блоку зі значенням 1 і тим самим одноразовим номером для шифрування основного вмісту повідомлення. ChaCha20 самостійно збільшує значення лічильника блоків для кожного наступного блоку, але управління одноразовим номером виконується зовнішньо, він має збільшуватись при кожному новому пакеті даних;

- після шифрування заголовка та даних виконується обчислення MAC за допомогою Poly1305. Вхідними даними для MAC є шифрований заголовок і зашифровані дані, а згенерований тег автентичності додається в кінець пакета;
- для розшифрування потрібно спочатку розшифрувати заголовок, щоб визначити довжину пакета, потім перевірити MAC. І лише у випадку успішної перевірки MAC дозволяється розшифрування основної частини пакета.

Таким чином, ChaCha20-Poly1305 забезпечує необхідний баланс між безпекою, ефективністю та простотою реалізації, що робить його оптимальним вибором для шифрування даних у мобільному застосунку. Його архітектура дозволяє повноцінно захищати як вміст повідомлення, так і його структуру, без необхідності покладатися на складні зовнішні схеми управління шифруванням [17].

## 2.4 Обґрунтування вибору методу генерації та зберігання ключа шифрування

У застосунку, що працює виключно на клієнтському пристрої, ключ для шифрування виводиться з пароля користувача. Це вимагає використання криптографічного алгоритму, який забезпечує обґрунтований рівень стійкості, не вимагає великих ресурсів та придатний для мобільного середовища.

На етапі проєктування було проаналізовано декілька сучасних алгоритмів: PBKDF2, bcrypt, Argon2 та HKDF-SHA256.

PBKDF2 є стандартом, рекомендованим у багатьох документах, проте його ефективність напряму залежить від кількості ітерацій, які доводиться вручну підбирати відповідно до обчислювальних можливостей пристрою. Недостатня кількість ітерацій робить ключ вразливим до перебору, а надмірна – значно уповільнює виконання на слабких смартфонах.

Bcrypt, який використовує блочний шифр Blowfish і був розроблений для захисту паролів у базах даних, має вузьку оптимізацію під серверні сценарії і

показує низьку продуктивність на ARM-процесорах, що характерні для більшості мобільних платформ.

Argon2, як найсучасніший алгоритм, демонструє найкращу стійкість до атак із використанням GPU і побудований з урахуванням захисту від атак через споживання пам’яті. Водночас його ефективність на мобільних пристроях обмежена високими вимогами до оперативної пам’яті та обчислювальних ресурсів [18].

Серед даних алгоритмів обрано HKDF з використанням SHA-256. Він не потребує ітерацій, складного налаштування параметрів, значних ресурсів чи апаратної підтримки. Його конструкція заснована на HMAC, що забезпечує передбачуваний рівень криптографічної стійкості, а сама процедура поділяється на два етапи: спочатку відбувається виділення ентропії з пароля і значення солі, після чого отриманий ключ розширюється до потрібної довжини.

Основний криптографічний ризик використання HKDF полягає в тому, що він не захищає від атаки перебору зі слабкими або легко-передбачуваними паролями. Алгоритм HKDF не має параметрів, які б штучно ускладнювали перебір на відміну від PBKDF2 та Argon2, де це забезпечується ітераціями або споживанням пам’яті. Отже, якщо вхідне значення має низьку ентропію, зловмисник може ефективно перебирати можливі варіанти, отримуючи той самий вихідний ключ [19].

Важливою умовою безпечного використання HKDF є поєднання його з перевіреними політиками створення складних паролів, які мають достатню довжину та випадковість. Крім того, застосування унікальної солі для кожного користувача або сесії значно ускладнює масові атаки, оскільки зловмисник змушений перебирати ключі для кожного випадку окремо. У контексті мобільного застосунку такий підхід дозволяє зберегти високу продуктивність навіть на пристроях із обмеженими ресурсами, одночасно підтримуючи належний рівень криптографічного захисту.

В архітектурі додатку включено генерування випадкової криптографічно стійкої солі на етапі виведення ключа, яка поєднується з паролем користувача. Схему генерації ключа показано на рисунку 2.3.

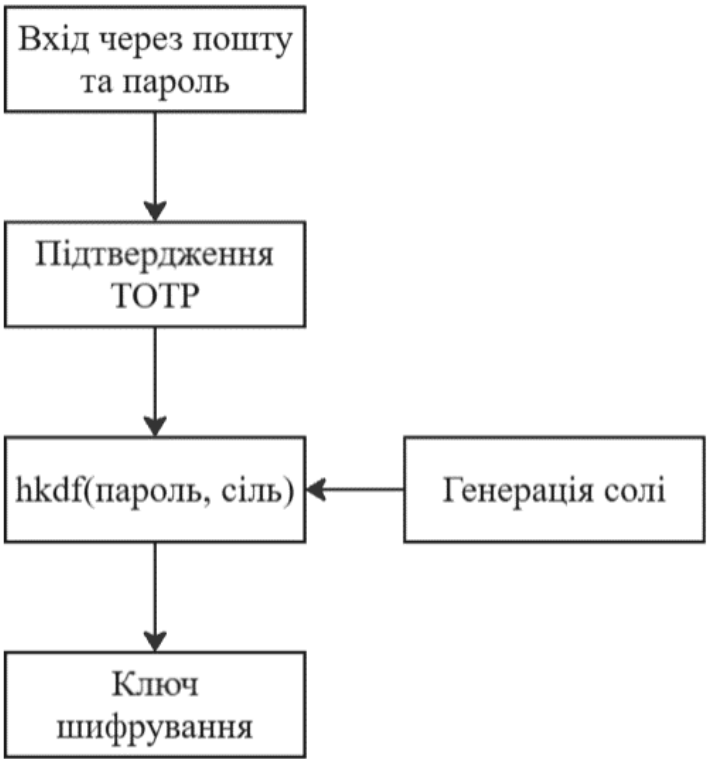



Рисунок 2.3 – Схема генерації ключа в архітектурі додатку

Архітектура застосунку передбачає використання захищених системних сховищ, доступних у мобільних операційних системах. Для платформи iOS таким сховищем є Keychain, що реалізує захищене зберігання ключів, токенів і паролів із використанням апаратного шифрування. Кожен запис у Keychain ізольований на рівні додатку, а доступ до нього можливий лише після успішної автентифікації або розблокування пристрою [20].

На платформі Android застосунок використовує Android Keystore для безпечного зберігання криптографічного ключа. При першому запуску генерується або імпортується AES-ключ, який зберігається у Keystore, що ізолює ключовий матеріал від решти системи. Усі дані конфігурації, які потребують захисту, шифруються з використанням AES-256 у режимі GCM, що гарантує як конфіденційність, так і цілісність інформації. Таким чином, навіть у разі отримання

фізичного доступу до файлової системи пристрою, розшифрування даних буде неможливим без доступу до внутрішнього Keystore [21].

IOS Keychain та Android Keystore – є рекомендованими способами зберігання чутливих даних в офіційних документаціях Apple та Google, і повністю відповідають вимогам до мобільної криптографічної безпеки, що застосовуються у фінансових і медичних застосунках. У контексті проєкту їх використання дозволяє надійно зберігати ключі без ризику компрометації навіть у разі втрати пристрою або його зламу.

### 2.5 Особливості зберігання зашифрованих даних у Firebase Firestore

У реалізованій архітектурі мобільного застосунку для зберігання даних використовується Firebase Firestore – хмарна нереляційна база даних, що входить до складу платформи Firebase. Вона забезпечує зберігання структурованих даних у реальному часі, з підтримкою автоматичної синхронізації між пристроями. Архітектурно Firestore реалізована як ієрархічна база, де дані організовано у вигляді колекцій та документів. Кожен документ має набір даних, що може містити вкладені підколекції, створюючи дерево даних довільної глибини [22]. Оскільки за замовчуванням Firestore не забезпечує шифрування даних, всі важливі дані, зокрема текстові записи користувача в цьому проєкті, проходять шифрування локально за допомогою ChaCha20-Poly1305 до відправлення на сервер. У зашифрованому вигляді дані передаються через Firebase SDK до бази даних, де зберігаються як текст закодований у форматі Base64.

Процес побудований таким чином, що сервер не володіє жодною інформацією про вміст повідомлення, а дані розшифровуються лише після введення пароля користувачем та генерації ключа локально в додатку. Така передача унеможливлює доступ до змісту навіть при повному контролі над акаунтом Firebase, оскільки криптографічний ключ для дешифрування ніколи не покидає пристрій і не зберігається в мережі. Цей принцип відповідає концепції локального шифрування. Він гарантує, що лише користувач має технічну

можливість отримати доступ до власних даних, незалежно від рівня компрометації інфраструктури або втрати контролю над хмарними сервісами.

Схему передачі конфіденційних даних зображено на рисунку 2.4.

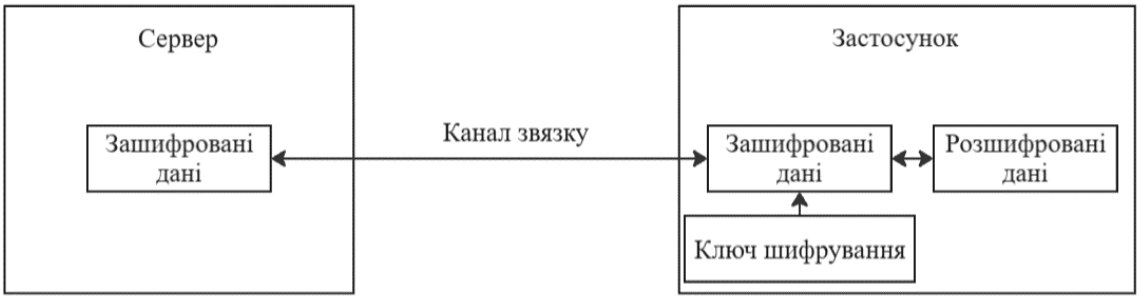



Рисунок 2.4 - Схема передачі конфіденційних даних

З боку серверної інфраструктури для Firestore реалізовано розширену політику доступу, яка контролює кожен запит до документів і підколекцій. В основі лежить принцип багатофакторної перевірки: користувач може лише читати головний документ профілю за наявності сесії автентифікації, а записи та доступ до вкладених колекцій дозволяються виключно після проходження TOTP верифікації. Це досягається за допомогою призначених токенів, які містять поле mfa_passed, що виставляється після підтвердження TOTP і використовується в Firebase Security Rules – правилах, які обмежують доступ на рівні Firebse серверу.

У проєкті визначено такий набір правил:

– тільки авторизований користувач може читати виключно свої дані;
– тільки авторизований користувач із параметром mfa_passed може змінювати інформацію в своєму документі;
– тільки авторизований користувач із параметром mfa_passed може читати та редагувати дані у підколекції свого документу.

Опис даних правил у сервісі Firebase:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read: if request.auth != null &&
                  request.auth.uid == userId;
      allow write: if request.auth != null &&
                  request.auth.token.mfa_passed == true &&
                  request.auth.uid == userId;
```

```
      match /{subcollection=**} {
        allow read, write: if request.auth != null &&
                           request.auth.token.mfa_passed == true &&
                           request.auth.uid == userId;
      }
    }
  }
}
```

Усі вкладені підколекції захищено даними правилами, читання або запис дозволено лише після двофакторної перевірки. Ця модель мінімізує ризик несанкціонованого доступу навіть у разі отримання зловмисником пошти та пароля, оскільки без активного параметру mfa_passed жодна модифікація або читання вмісту не буде дозволено. Таким чином, TOTP виступає не тільки механізмом локальної перевірки, але й серверним фільтром доступу до Firestore.

Отже, така модель об'єднує локальне шифрування, MFA контроль доступу та Firestore як віддалене сховище. Навіть при повному доступі до бази даних Firebase стороння особа не зможе прочитати вміст документів або підколекцій без знання пароля та проходження MFA. Така схема дозволяє дотримуватись високого рівня безпеки за відсутності власного сервера, спираючись лише на можливості Firebase та криптографічний захист у застосунку.

### 2.6 Висновки до розділу

У цьому розділі було сформовано архітектуру захищеного мобільного застосунку, що ґрунтується на принципах локального шифрування, двофакторної автентифікації та локальної генерації ключів. Основу захисту становить симетричне шифрування за алгоритмом ChaCha20-Poly1305, яке обрано з огляду на його високу продуктивність у мобільному середовищі та стійкість до атак побічних каналів.

Ключ шифрування динамічно виводиться на основі пароля користувача за допомогою HKDF-SHA256. Для зберігання ключа шифрування використано рекомендовані сховища: Android Keystore і IOS Keychain.

З боку хмарного зберігання реалізовано модель зберігання зашифрованих даних у Firebase Firestore. Вся конфіденційна інформація шифрується на стороні

застосунку до моменту надсилання, що виключає доступ до вмісту зі сторони сервера. Додатковий контроль доступу забезпечується через систему правил Firestore, які перевіряють ідентифікатор користувача та обов’язкову наявність даних про проходження MFA у токені автентифікації.

Особливу увагу в реалізації архітектури приділено зручності використання без шкоди для безпеки. Усі критичні процеси, зокрема введення пароля, підтвердження через TOTP та шифрування даних, відбуваються прозоро для користувача, без необхідності в складних налаштуваннях або додаткових технічних знаннях. Це досягається завдяки інтеграції з функціями мобільних операційних систем та інтуїтивному інтерфейсу, що дозволяє користувачам ефективно управляти конфіденційною інформацією. Такий баланс між функціональністю та простотою використання є ключовим чинником успішного впровадження захищених застосунків у повсякденну практику.

Додатково варто зазначити, що архітектура застосунку враховує принципи мінімізації прав доступу, що знижує потенційний вплив у разі компрометації одного з компонентів. Усі взаємодії між застосунком і сервером проходять через захищені канали з використанням HTTPS, а доступ до даних обмежується як на рівні коду застосунку, так і за допомогою політик доступу Firestore. Це дозволяє зменшити ризики несанкціонованого доступу з боку сторонніх осіб.

Запропоновані технічні рішення орієнтовані на повну ізоляцію користувацьких даних як у хмарі, так і на самому пристрої. Механізми автентифікації та зберігання реалізовані таким чином, щоб унеможливити доступ до конфіденційної інформації навіть у разі компрометації серверної інфраструктури або втрати фізичного контролю над пристроєм. Такий підхід дозволяє створити застосунок, який залишає контроль над даними виключно в руках користувача.

Загалом, система поєднує переваги автономного зберігання, сильного криптографічного захисту та інфраструктури Firebase без покладання на довіру до серверної частини, що відповідає сучасним вимогам до безпеки мобільних застосунків.

# 3 РОЗРОБКА ТА ТЕСТУВАННЯ ЗАХИЩЕНОГО МОБІЛЬНОГО ЗАСТОСУНКУ

Цей розділ присвячено практичній реалізації захищеного мобільного застосунку, призначеного для зберігання конфіденційної інформації користувача. У підрозділах описано впровадження двофакторної автентифікації, захист даних у Firestore за допомогою правил доступу та локального шифрування, а також створення механізмів перевірки безпеки функціонування системи.

У межах розділу здійснюється обґрунтований вибір інструментів і технологій, реалізуються ключові компоненти застосунку та серверної частини, а також проводиться тестування критичних функцій із безпеки.

## 3.1 Вибір архітектури та інструментів розробки

У процесі створення захищеного мобільного застосунку було обрано сучасний технологічний стек, який дозволяє реалізувати високий рівень безпеки і гнучкість розробки. Основною метою стало створення застосунку з підтримкою двофакторної автентифікації, шифруванням конфіденційних даних та централізованим зберіганням інформації у хмарному середовищі.

Для реалізації графічного інтерфейсу використано Flutter – кросплатформений SDK від компанії Google, яка дозволяє створювати інтерфейси для Android та iOS з єдиною кодовою базою. Розробка здійснювалася в середовищі Visual Studio Code, що забезпечило зручну інтеграцію з інструментами налагодження, автоматизації збірки та контролю версій. Основні переваги Visual Studio Code полягають у наступному:

- швидке та оптимізоване середовище розробки;
- підтримка Flutter через офіційне розширення;
- підтримка розширень для покращення роботи із Dart кодом та Flutter SDK;
- можливість інтеграції із Xcode для написання IOS застосунків;
- інтеграція з git, контроль версій.

Платформа Flutter забезпечує додаткові переваги:

- одна кодова база для Android та iOS;
- висока продуктивність завдяки спеціальній оптимізації Dart під Flutter SDK;
- кросплатформеність;
- можливості швидкого оновлення та перезапуску;
- обширна документація;
- каталоги віджетів з Material Design доступних у Flutter SDK;
- можливість написання плагінів;
- SDK регулярно оновлюється;
- Flutter має власну платформу для створення та публікації бібліотек;
- зручний інтерфейс та швидка розробка.

У якості серверної частини застосунку використано сервіс Firebase, який надає модульну хмарну інфраструктуру з підтримкою автентифікації, бази даних Firestore, хмарних функцій і механізмів управління доступом. Firebase Authentication використовується для базової автентифікації користувачів з поштою та паролем. Переваги Firebase включають:

- повна інтеграція з автентифікацією та хмарною базою даних;
- синхронізація даних у режимі реального часу;
- висока масштабованість та безкоштовні плани для невеликих додатків;
- високий захист гарантований Google;
- можливість створювати додатки без власного серверу;
- спрощення розробки для невеликих застосунків;
- швидкі та прості інтеграції з Google сервісами.

Для реалізації другого фактору автентифікації у застосунку було використано сервіс Twilio, який надає програмні інтерфейси для генерації та перевірки TOTP. Це дозволяє забезпечити додатковий рівень захисту без використання власного серверу та написання відповідного коду. Переваги використання Twilio:

- підтримка відкритого стандарту TOTP (RFC 6238);
- гнучка інтеграція з мобільними платформами та іншими сервісами;
- обширна документація;

– спрощення розробки для невеликих застосунків.

Для захисту даних, які зберігаються у Firestore, реалізовано локальне шифрування. Застосовано симетричний алгоритм ChaCha20-Poly1305, який поєднує автентифіковане шифрування з високою ефективністю. Основні переваги цього алгоритму:

– висока швидкість у мобільних процесорах завдяки потоковій природі;
– низьке енергоспоживання у порівнянні з AES при відсутності апаратної підтримки;
– вбудована перевірка цілісності даних через Poly1305.

Формування ключів для шифрування здійснюється за допомогою HKDF, що забезпечує:

– криптографічну стійкість виведеного ключа;
– можливість формувати незалежні ключі для різних цілей із одного секрету;
– зменшення ризику компрометації навіть при частковому витоку даних.

Окрім реалізації графічного інтерфейсу, розробка також включала написання серверної логіки для обробки автентифікації, генерації TOTP секретів та роботи з JWT токеном автентифікації. Для цього було використано Firebase Cloud Functions, які реалізовано на базі Node.js.

Node.js став оптимальним вибором з таких причин:

– SDK Firebase Cloud Functions розроблений саме для середовища Node.js і має повну сумісність із його модулями;
– неблокуюча модель вводу і виводу дозволяє обробляти велику кількість запитів із низькою затримкою;
– доступ до розширених сервісів Google Cloud, таких як Identity Platform, Cloud Scheduler, Pub/Sub, Firestore Admin SDK.

Серверна логіка написана на Node.js із використанням Firebase Admin SDK, що забезпечує безпечну взаємодію з Firebase Authentication, базою даних Firestore та сторонніми сервісами, в даному проєкті. Завдяки цьому вдалося реалізувати перевірку другого фактора автентифікації та оновлення токенів користувача.

Таким чином, обраний набір технологій дозволяє реалізувати безпечний, масштабований та ефективний мобільний застосунок, що відповідає сучасним вимогам до захисту персональних даних. Використання Flutter забезпечує гнучку розробку з високою продуктивністю на обох мобільних платформах. Firebase, у свою чергу, надає готову інфраструктуру для автентифікації, зберігання та синхронізації даних у хмарі з детальним контролем доступу. Впровадження двофакторної автентифікації через Twilio підвищує рівень стійкості до атак, пов’язаних із компрометацією облікових записів. А використання алгоритмів ChaCha20-Poly1305 та HKDF дозволяє реалізувати криптографічно надійний захист даних без надмірного навантаження на пристрій. Така архітектура дає змогу забезпечити конфіденційність, цілісність та контроль доступу до даних навіть у разі часткової компрометації окремих компонентів системи. У підсумку, обрана технологічна база створює передумови для надійного функціонування мобільного застосунку у реальних умовах.

### 3.2 Реалізація мобільного застосунку

Мобільний застосунок виконує роль основної точки взаємодії користувача з функціоналом системи. Особлива увага приділялася реалізації механізмів захисту конфіденційних даних на рівні пристрою, ще до моменту їх передавання до хмарного середовища Firebase.

Після запуску застосунок відображає форму авторизації з підтримкою email-пароля як першого фактору. У разі успішної автентифікації користувача система переходить до перевірки TOTP коду, згенерованого за стандартом RFC 6238.

Після успішного входу всі дані, що створюються або зчитуються із бази даних, проходять шифрування та розшифрування локально. Для цього реалізовано механізм шифрування з автентифікацією за допомогою ChaCha20-Poly1305, що забезпечує не тільки конфіденційність, але й захист від підробки даних. Перед шифруванням відбувається формування ключа за допомогою HKDF, у якості початкового секрету використовується ключ, виведений із пароля користувача, який не зберігається у відкритому вигляді в системі.

Схема роботи застосунку зображено на рисунку 3.1.

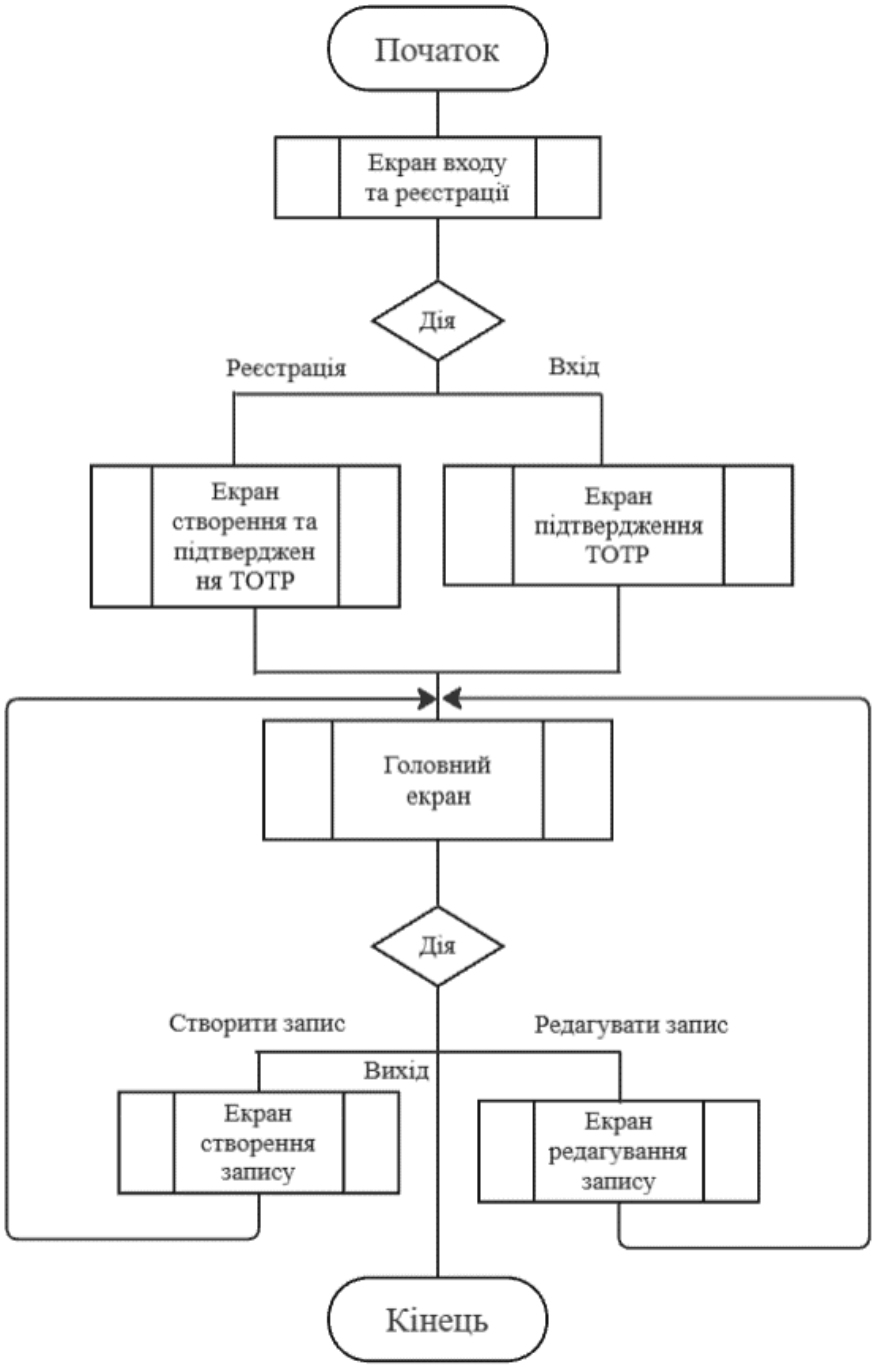



Рисунок 3.1 – Схема роботи застосунку

Дана схема відображає послідовність основних етапів взаємодії користувача з системою.

У застосунку реалізовано рівневу архітектуру, що передбачає чітке розділення відповідальностей між компонентами. Логіка інтерфейсу структурована за принципом BLoC. Цей принцип дозволяє відокремити основну логіку від відображення інтерфейсу. Основні рівні поділено наступним чином:

- рівень перегляду відповідає за інтерфейс користувача, побудований на Flutter віджетах, і реагує на зміни стану через потоки даних;
- рівень даних забезпечує зберігання та обробку даних, зокрема, шифрування, розшифрування, генерацію TOTP, роботу з Firebase;
- рівень блок слугує посередником між рівнями перегляду та даних, обробляє події, трансформує їх у стани і забезпечує реактивну взаємодію.

Дана модель дозволяє досягти гнучкості, розмежованості та масштабованості застосунку, а також спрощує подальший розвиток і впровадження нових механізмів захисту.

Першим етапом взаємодії користувача із застосунком є реєстрація. Для цього реалізовано метод registerUser, що викликається на рівні даних та ініціює створення облікового запису у Firebase Authentication:

Фрагмент методу registerUser на рівні data:

```
final userCredential = await _auth.createUserWithEmailAndPassword(
  email: email,
  password: password,
);
final uid = userCredential.user!.uid;
logSuccess('User registered: $email, $uid');
return uid;
```

Цей метод приймає електронну пошту та пароль, які передаються до функції createUserWithEmailAndPassword – стандартного API Firebase для створення нового користувача. У разі успішної реєстрації повертається унікальний ідентифікатор користувача, який надалі використовується для прив’язки до даних у Firestore.

Схожим чином зроблено метод login. Цей метод також знаходиться на рівні даних та відповідає за вхід користувача у ситему. Великою різницею є використання хмарної функції замість стандартного методу входу із email та пароль. Ця хмарна функція змінює дані jwt токену, а саме поля mfa_enrolled та mfa_passed.

Фрагмент методу login:

```
final result = await FirebaseFunctions.instance
        .httpsCallable('loginUserAndResetMFA')
        .call({'email': email, 'password': password});
    final customToken = result.data['customToken'];
    final userCredential = await
FirebaseAuth.instance.signInWithCustomToken(
```

```
      customToken,
    );
    await userCredential.user?.getIdToken(true);
    logSuccess('User logged in: $email');
    return userCredential.user?.uid;
```

Після успішного входу в систему ініціюється процедура генерації TOTP секрету, який надалі буде використовуватись для створення одноразових кодів доступу. Для цього викликається хмарна функція createTOTP. Фрагмент коду даного методу:

```
final result = await _functions.httpsCallable('createTOTP').call({
    'userIdentifier': uid,
    'factorName': currentUser!.email,
  });
  logSuccess('TOTP created (unverified): ${result.data}');
  return EnrollTotpResponse.fromJson(result.data);
```

Даний метод виконує асинхронний виклик до Firebase Cloud Functions, передаючи у запиті унікальний ідентифікатор користувача. У результаті функція повертає TOTP секрет, який містить інформацію для подальшого створення QR-коду або ручного додавання в додаток аутентифікатор.

Після генерації та відображення TOTP секрету користувачу наступним етапом є перевірка, для цього застосунок реалізує функцію verifyTotp, яка надсилає введений код до хмарної функції confirmTOTPAndSetClaim для підтвердження.

Фрагмент методу verifyTotp:

```
final result = await _functions
       .httpsCallable('confirmTOTPAndSetClaim')
       .call({
         'userIdentifier': uid,
         'verificationSid': verifSid,
         'code': code,
       });
  logSuccess('TOTP verified for $uid: ${result.data}');
```

Ця функція відправляє на сервер три параметри: унікальний ідентифікатор користувача, twilio ідентифікатор, який повертався в методі createTOTP, а також сам одноразовий код TOTP, згенерований користувачем на основі секрету. На сервері код перевіряється сервісом Twilio, після чого у разі успішної валідації до токена користувача додається поле mfa_passed зі значенням «true».

Ці дані автоматично вбудовуються в JWT токен користувача і можуть бути використані у Firebase Security Rules для умовного надання доступу до зашифрованих даних. Таким чином, перевірка TOTP завершує процес

двофакторної автентифікації, надаючи користувачу доступ до конфіденційної частини застосунку.

Після початкової реєстрації та підтвердження TOTP секрету, наступні сесії користувача передбачають періодичну перевірку одноразового коду доступу як частину процесу багатофакторної автентифікації. Для цього реалізовано метод createChallenge, який використовується для валідації TOTP під час входу.

Фрагмент методу createChallenge:

```
final result = await _functions.httpsCallable('createChallenge').call({
     'userIdentifier': uid,
     'factorSid': verifSid,
     'code': code,
   });
   final isSuccess = result.data['success'];
   logSuccess('TOTP challenge result: $isSuccess');
   return isSuccess;
```

Цей код ініціює виклик до хмарної функції createChallenge, яка перевіряє, чи відповідає наданий користувачем TOTP код очікуваному значенню, згенерованому на основі збереженого секрету. Параметри userIdentifier, factorSid та code забезпечують ідентифікацію користувача, вказівку на фактор автентифікації та безпосередньо сам код перевірки.

Для реалізації умовного доступу до функціоналу застосунку залежно від стану автентифікації було створено метод checkUserStatus, який аналізує токен користувача та повертає його актуальний статус входу з урахуванням багатофакторної автентифікації:

Фрагмент коду:

```
final claims = json.decode(payload);
    final hasEnrolledMfa = (claims['mfa_enrolled'] as bool?) ?? false;
    final hasPassedMfa = (claims['mfa_passed'] as bool?) ?? false;
    UserAuthStatus status;
    if (hasPassedMfa) {
      status = UserAuthStatus.passedMFA;
    } else if (hasEnrolledMfa) {
      status = UserAuthStatus.enrolledMFA;
    } else {
      status = UserAuthStatus.loggedIn;
    }
```

Метод спочатку оновлює токен, після чого декодує його дані, та зчитує значення mfa_enrolled і mfa_passed. На основі цих значень визначається поточний рівень автентифікації користувача:

– loggedOut – користувач не авторизований або токен недійсний;

– loggedIn – користувач авторизований, але без налаштованого MFA;
– enrolledMFA – користувач автентифікований, але код ще не підтверджено;
– passedMFA – автентифікація з TOTP успішно пройдена.

Ця логіка дає змогу контролювати доступ до критичних дій або даних відповідно до стану автентифікації, а також реалізувати сценарії автоматичного перенаправлення на правильні екрани автентифікації. Таким чином, токен виконує роль єдиного джерела істини для визначення рівня доступу.

Після успішної активації TOTP та генерації пов'язаних криптографічних параметрів, таких як сіль та ідентифікатор фактору, система формує об'єкт UserModel і зберігає його у Firestore.

Реалізація методу setInitialData:

```
final model = UserModel.initial(id: _authRepo.uid, salt: salt, sid: sid);
    await _firestore
        .collection(usersCol)
        .doc(_authRepo.uid)
        .set(model.toJson());
    logSuccess('User data is set in $usersCol/${_authRepo.uid}');
    return model;
```

Для роботи із криптографічними функціями створено клас-сервіс EncryptionService. Всі операції шифрування, розшифрування та проміжні кроки обробляються в цьому класі.

Одним із ключових аспектів безпечного виведення симетричного ключа шифрування є використання унікальної солі для кожного користувача. У застосунку реалізовано генерацію криптографічно стійкої солі за допомогою надійного генератора випадкових байтів. Фрагмент коду наведено нижче.

```
final random = Random.secure();
    final bytes = Uint8List.fromList(
      List.generate(length, (_) => random.nextInt(256)),
    );
    final saltString = _toBase64String(bytes);
    return saltString;
```

Для передачі даних та їхнього зберігання використовується Base64 кодування. Формат Base64 дозволяє перетворити довільну послідовність байтів у рядок, що складається виключно з друкованих ASCII символів. Це робить його особливо зручним для:

– зберігання в базах даних, які очікують рядкові значення;
– серіалізації у форматі JSON;

– передачі в URL та HTTP-запитах.

Алгоритм кодування працює наступним чином: кожні 3 байти вхідних даних розбиваються на 4 групи по 6 біт, які далі представляються символами з алфавіту Base64. У разі, якщо довжина вхідного масиву не кратна трьом, додаються символи “=” для заповнення [23].

Фрагмент коду, який відповідає за кодування та декодування:

```
String _toBase64String(Uint8List bytes) {
  return base64Encode(bytes);
}
Uint8List _fromBase64String(String base64String) {
  return base64Decode(base64String);
}
```

Для шифрування конфіденційних даних у мобільному застосунку використовується симетричний криптографічний ключ, який не зберігається безпосередньо в системі, а виводиться з пароля користувача за допомогою функції HKDF. Відповідну логіку реалізовано у методі deriveKeyFromPassword:

```
final secretKey = await _hkdfKeyDerivation.deriveKey(
      secretKey: SecretKey(utf8.encode(password)),
      nonce: _fromBase64String(salt),
    );
    final bytes = Uint8List.fromList(secretKey.bytes);
    secretKey.destroy();
    return _toBase64String(bytes);
```

У цьому фрагменті вхідними параметрами виступають пароль користувача і криптографічна сіль, яка раніше була згенерована випадковим чином і збережена у Firestore у форматі Base64. Метод використовує реалізацію HKDF, яка на основі HMAC функції виконує виведення ключа із зазначеного секрету.

Для забезпечення конфіденційності даних, які створюються та зберігаються користувачем, у мобільному застосунку реалізовано шифрування відкритого тексту на стороні клієнта. Це здійснюється через метод encryptPlainText, який використовує алгоритм симетричного шифрування ChaCha20-Poly1305.

```
final nonce = _textEncryptionAlg.newNonce();
  final secretKey = SecretKey(_fromBase64String(encryptionKey));
  final encrypted = await _textEncryptionAlg.encrypt(
    utf8.encode(text),
    secretKey: secretKey,
    nonce: nonce,
  );
  return EncryptionData(
    encryptedText:
_toBase64String(Uint8List.fromList(encrypted.cipherText)),
    nonce: _toBase64String(Uint8List.fromList(nonce)),
    mac: _toBase64String(Uint8List.fromList(encrypted.mac.bytes)),
```

```
);
```

Розшифрування також виконується повністю локально у застосунку. Для цього застосовується метод, який відновлює вихідний текст з об'єкта EncryptionData. Відповідний код подано нижче:

```
final clearBytes = await _textEncryptionAlg.decrypt(
    SecretBox(
      _fromBase64String(encrData.encryptedText),
      nonce: _fromBase64String(encrData.nonce),
      mac: Mac(_fromBase64String(encrData.mac)),
    ),
    secretKey: SecretKey(_fromBase64String(key)),
  );
  return utf8.decode(clearBytes);
```

Після завершення локального шифрування конфіденційного запису, зашифровані дані зберігаються у підколекції користувача в Firebase Firestore. Відповідний код для збереження реалізовано наступним чином:

```
await _firestore
    .collection(usersCol)
    .doc(_authRepo.uid)
    .collection(dataRecordsCol)
    .doc(record.id)
    .set(record.toJson());
logSuccess(
  'Data record set in
$usersCol/${_authRepo.uid}/$dataRecordsCol/${record.id}',
);
```

Важливо, що всі дані, які записуються до Firestore, вже пройшли етап шифрування. Отже, навіть при прямому доступі до цієї колекції зловмисник отримає лише нерозшифровувані байти. Ця реалізація дозволяє зберігати будь-яку кількість зашифрованих записів, кожен з яких є ізольованим, з унікальними параметрами шифрування.

Для повноти керування зашифрованими записами користувача додаток також реалізує можливість їх безпечного видалення. Це дозволяє користувачу мати повний контроль над життєвим циклом своїх даних. Відповідний метод взаємодії з Firestore наведено нижче:

```
await _firestore
     .collection(usersCol)
     .doc(_authRepo.uid)
     .collection(dataRecordsCol)
     .doc(id)
     .delete();
 logSuccess(
   'Data record deleted in
$usersCol/${_authRepo.uid}/$dataRecordsCol/$id',
 );
```

Окрім збереження та видалення зашифрованих даних, важливою функцією застосунку є можливість отримання списку записів користувача. Ця операція реалізована через звернення до відповідної вкладеної колекції Firestore:

```
final records =
     await _firestore
         .collection(usersCol)
         .doc(_authRepo.uid)
         .collection(dataRecordsCol)
         .orderBy('updatedAt', descending: true)
         .get();
 final models =
     records.docs.map((doc) {
       final data = doc.data();
       return DataRecordModel.fromJson(data);
     }).toList();
 return models;
```

Цей етап не включає розшифрування, оскільки воно виконується окремо, після проходження перевірки автентифікації та виведення ключа. Це дає змогу розділити відповідальність між етапом зчитування даних та етапом їх обробки, що відповідає принципам безпечного дизайну.

Після отримання зашифрованих записів із Firestore наступним етапом є їх розшифрування перед передачею у рівень перегляду. Для цього використовується спеціальна подія у класі рівня блок, що виконує поетапну обробку всіх об’єктів та формує відповідний стан:

```
final encrypted = await _dataRecordRepo.getDataRecords();
 final records = <DataRecordModel>[];
   for (final rec in encrypted) {
     final decrypted = await _encryptionService.decryptRecord(
     encrData: EncryptionData(
       encryptedText: rec.encryptedText,: rec.nonce,: rec.mac,),
       key: await _secureStorageService.getKey() ?? '',
     );
 records.add(rec.copyWith(encryptedText: decrypted));
```

Особливістю цієї реалізації є асинхронна обробка кожного запису, що дозволяє обробляти великі обсяги даних без блокування основного потоку. Крім того, ключ, необхідний для розшифрування, отримується з захищеного сховища.

У межах програмної реалізації було впроваджено повноцінну архітектуру захисту даних, яка базується на сучасних криптографічних принципах та практиках безпечної мобільної розробки. Основна функціональність побудована за моделлю BLoC з чітким поділом логіки на рівні: інтерфейс користувача, обробка подій та доступ до даних.

Важливим етапом реалізації є поєднання Firebase Authentication для базової ідентифікації з реалізацією другого фактора автентифікації через TOTP на основі сервісу Twilio. Це дозволило досягти високого рівня захищеності при мінімальній складності для користувача. Ключова відмінність застосунку полягає у тому, що всі конфіденційні дані проходять шифрування на клієнті з використанням алгоритму ChaCha20-Poly1305, а ключі виводяться з пароля користувача через HKDF з індивідуальною сіллю.

Отримані результати засвідчують, що застосунок не лише реалізує повноцінний захист, але й дозволяє масштабувати його під різні потреби користувача, зберігаючи високу гнучкість та контроль над інформацією.

### 3.3 Реалізація серверної частини та зберігання даних у Firestore

Серверна частина мобільного застосунку реалізована за допомогою хмарної платформи Firebase, яка виконує функції автентифікації, керування доступом до ресурсів та зберігання даних користувачів. Усі критично важливі дії відбуваються через Cloud Firestore, нереляційну базу даних, у реальному часі, яка дозволяє організувати зберігання даних у вигляді вкладених колекцій і документів. Завдяки цьому реалізується структура зберігання, яка адаптована до багаторівневого доступу.

Зберігання даних у Firestore побудовано навколо головної колекції users, де кожен документ відповідає унікальному ідентифікатору користувача в системі автентифікації. Внутрішня структура містить вкладену колекцію dataRecords, що містить зашифровані записи. Доступ до Firestore відбувається через стандартний API [24].

Вданому проекті використання стандартних функцій Firebase виявилось недостатньо для реалізації двофакторної автентифікації, тому було створено хмарні функції. Firebase Cloud Functions – це середовище, яке дозволяє запускати код автоматично у відповідь на події, виклики через HTTPS, звернення з Firebase Admin SDK або заплановані задачі через Cloud Scheduler. Весь код, написаний на Node.js, розміщується в інфраструктурі Google Cloud і виконується у повністю керованому

середовищі, що усуває потребу в налаштуванні, масштабуванні та обслуговуванні власних серверів. Такий підхід забезпечує безперервну доступність серверної логіки при мінімальних витратах на її підтримку [25].

У проєкті Cloud Functions використовуються не для обробки логіки користувача, а як проксі-механізм для реалізації критичних перевірок безпеки, зокрема багатофакторної автентифікації та контролю параметрів JWT токену автентифікації.

Однією з ключових хмарних функцій, реалізованих на платформі Node.js, є loginUserAndResetMFA. Вона відповідає за автентифікацію користувача на основі пошти та пароля з подальшим скиданням стану двофакторної автентифікації. Її основна мета це розділення базової автентифікації та підтвердження TOTP, реалізуючи двофакторну логіку через дані, які зберігаються у JWT токені.

Фрагмент методу:

```
      const userRecord = await admin.auth().getUserByEmail(email);
            const customToken = await
admin.auth().createCustomToken(userRecord.uid);
            const currentClaims =
              (await admin.auth().getUser(userRecord.uid)).customClaims || {};
            const updatedClaims = {
              mfa_passed: false,
              mfa_enrolled: currentClaims.mfa_enrolled ?? false,
            };
            await admin.auth().setCustomUserClaims(userRecord.uid, updatedClaims);
```

Ця функція дозволяє користувачеві пройти лише перший етап автентифікації, після чого необхідно підтвердити другий фактор. Саме такий поділ дозволяє керувати доступом, які перевіряються у Firebase Security Rules.

Однією з ключових серверних функцій є createTOTP, яка відповідає за генерацію TOTP секрету для прив’язки другого фактору автентифікації користувача. Реалізація функції виконана на основі Twilio Verify API і розміщена у Firebase Cloud Functions.

Фрагмент коду:

```
  const newFactor = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .newFactors.create({
          factorType: "totp",
          friendlyName: factorName,
        });
      return {
        success: true,
```

```
        message: "TOTP setup initiated. Scan QR in an Authenticator app.",
        verificationSid: newFactor.sid,
        secret: newFactor.binding["secret"],
      };
```

Функція виконує наступну логіку:

– перевіряє, чи користувач автентифікований. Якщо ні – запит блокується з помилкою;
– приймає два параметри: ідентифікатор користувача для Twilio, який по архітектурі збігається із ідентифікаторм користувача у Firebase Authentication, та ім’я фактору;
– створює новий фактор типу TOTP через Twilio Verify API. У результаті повертається об'єкт, який містить ідентифікатор фактору та секрет, з якого користувач генерує одноразові коди у додатках аутентифікаторах.

Функція повертає об’єкт, який включає секрет TOTP фактору, який на клієнті може бути перетворено у QR-код для зручності прив’язки. Сам Twilio зберігає секрет безпечно, а перевірка TOTP здійснюється виключно на сервері.

Завдяки цьому серверна частина гарантує, що генерація TOTP відбувається під контролем серверу, а застосунок не може самостійно модифікувати або створити неконтрольований фактор. Це критично важливо для цілісності процесу двофакторної автентифікації та унеможливлює обхід MFA.

Після генерації TOTP секрету та прив’язки аутентифікатора користувач має підтвердити, що він володіє цим секретом, шляхом введення одноразового коду. Цей процес обробляється через функцію confirmTOTPAndSetClaim, яка також реалізована у Firebase Cloud Functions з використанням Twilio API.

Фрагмент методу confirmTOTPAndSetClaim:

```
const resp = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .factors(verificationSid)
        .update({ authPayload: code });
      if (resp.status !== "verified") {
        throw new functions.https.HttpsError(
          "internal",
          "TOTP code verification failed."
        );
      }
      await firebaseAdmin
        .auth()
        .setCustomUserClaims(uid, { mfa_enrolled: true, mfa_passed: true });
```

Функція виконує наступні завдання:

– перевіряє автентифікацію користувача через;
– приймає параметри: ідентифікатор користувача, ідентифікатор TOTP фактору Twilio цього користувача та код, що був введений користувачем;
– здійснює перевірку коду за допомогою Twilio Verify API;
– у разі успішної верифікації, оновлює JWT токен користувача в Firebase Auth, додаючи параметри mfa_enrolled та mfa_passed із значеннями true.

Ці дані мають вирішальне значення в системі контролю доступу. Firebase Security Rules перевіряють їх, щоб дозволити доступ до зашифрованих даних лише після проходження MFA.

Таким чином, функція confirmTOTPAndSetClaim виконує критичну роль у перевірці другого фактору, закриваючи процес реєстрації MFA. Вона гарантує, що ці дані, які зберігаються в JWT токені автентифікації не можуть бути присвоєні без реального підтвердження коду, забезпечуючи високу довіру до системи авторизації та автентифікації у застосунку.

У процесі подальших входів користувача після проходження базової автентифікації, застосунок ініціює окрему перевірку TOTP через функцію createChallenge. Вона реалізована як Firebase Cloud Function із використанням Twilio Verify API. Фрагмент коду:

```
const challenge = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .challenges.create({
          authPayload: code,
          factorSid: factorSid,
        });
      if (challenge.status === "approved") {
        const updatedClaims = {
          mfa_passed: true,
          mfa_enrolled: true,
        };
        await admin.auth().setCustomUserClaims(userIdentifier,
updatedClaims);
      }
```

Ця функція виконує такі дії:

– перевіряє, чи користувач автентифікований;
– приймає параметри: ідентифікатор користувача, ідентифікатор TOTP фактора Twilio цього користувача та код, що був введений користувачем;
– здійснює перевірку коду за допомогою Twilio Verify API;

– Twilio Verify API створює перевірку на правильність коду для вказаного фактору.

Ця функція є частиною перевірки MFA під час входу. Її застосування дозволяє виконувати багатофакторну перевірку у довільний момент часу, не порушуючи принципів безпеки та зручності користувача.

### 3.4 Тестування застосунку та аналіз результатів

Метою тестування є перевірка працездатності ключових функцій захищеного мобільного застосунку, зокрема механізмів автентифікації, шифрування, взаємодії з базою даних Firestore та захисту від типових загроз безпеці. Особливу увагу приділено перевірці реалізації двофакторної автентифікації на основі TOTP, а також коректності обробки даних у зашифрованому вигляді.

Тестування проводилося вручну та частково автоматизовано з використанням симулятора IOS девайсу та Postman для перевірки HTTP-викликів. Верифікація результатів здійснювалася шляхом аналізу:

– відповідей від Cloud Functions;
– записів у Firestore;
– поведінки застосунку у випадку помилок, недійсних кодів та паролів;
– наявності чи відсутності доступу до захищених ресурсів за умови неуспішного проходження MFA.

Основні критерії прийнятності:

– всі записи користувачів повинні зберігатися у зашифрованому вигляді;
– доступ до зашифрованої інформації дозволений лише після проходження двох факторів автентифікації;
– помилки автентифікації, неправильні коди або несанкціоновані дії мають блокувати доступ;
– система повинна обробляти помилки й не розкривати жодної службової або внутрішньої інформації.

Методика тестування базується на позитивних і негативних сценаріях для перевірки результатів взаємодії між компонентами без доступу до внутрішньої

логіки їх реалізації. У наступних підрозділах буде подано результати тестування та проведено оцінку рівня захищеності застосунку.

3.4.1 Інтеграційне ручне тестування

Тестування проводилося вручну за сценаріями типового використання мобільного застосунку з перевіркою всіх етапів обробки даних та автентифікації користувача. Нижче наведено результати проходження ключових етапів з описом очікуваної та фактичної поведінки системи.

Тест 1: Реєстрація користувача.

– вхідні дані: правильні пошта та пароль;
– очікуваний результат: створено новий акаунт, користувач отримав унікальний ідентифікатор, відображено екран генерації MFA;
– фактичний результат: Успішно. Користувача створено у Firebase Authentication, користувача направлено на екран реєстрації TOTP (рис. 3.2). При надто слабкому паролі чи неправильній пошті, з’являється повідомлення (рис. 3.3).

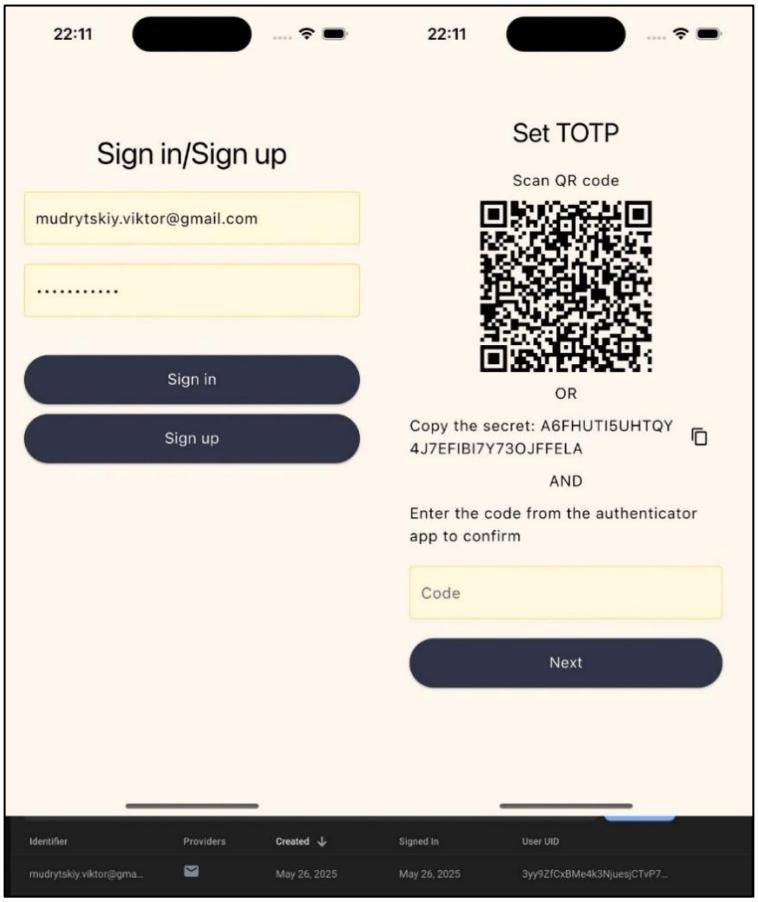



Рисунок 3.2 – Результат реєстрації користувача

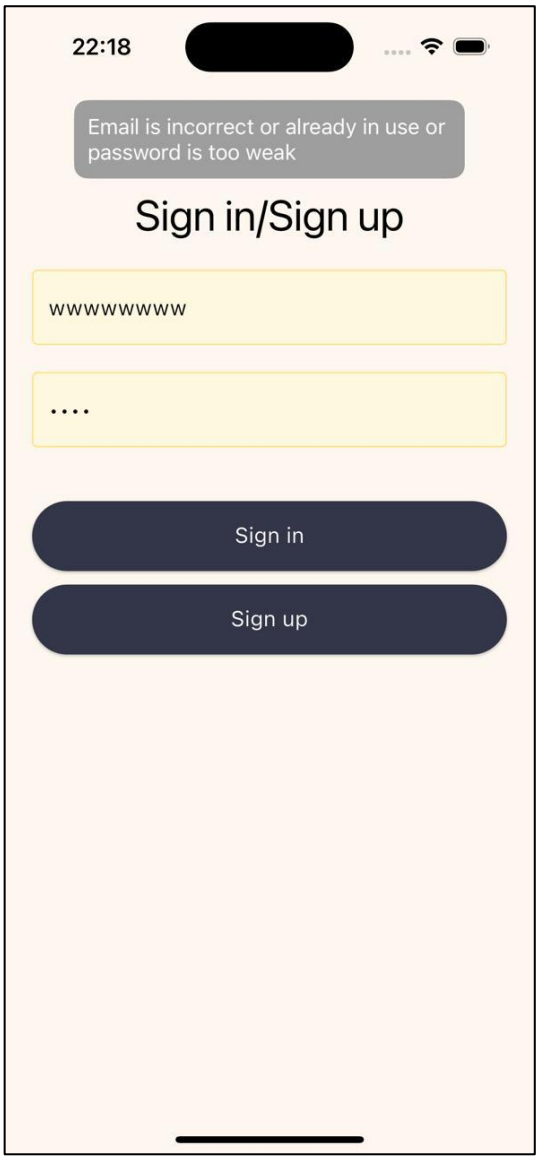



Рисунок 3.3 – Помилка введених даних під час реєстрації

Тест 2: Генерація та підтвердження TOTP фактору.

– очікуваний результат: відображено QR-код та секрет, готовий до сканування аутентифікатором, при успішному введені коду користувач попадає на головний екран, користувач має бути створений у базі даних Firestore у документі зі шляхом users/uid;
– фактичний результат: Успішно. Користувача створено у Firestore базі даних. Користувача направлено на головний екран (рис. 3.4), при неправильному вводі коду з’являється помилка (рис. 3.5).

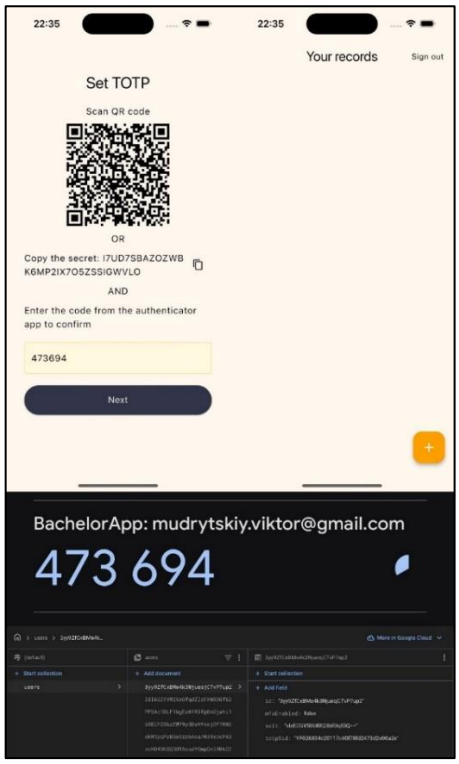



Рисунок 3.4 – Результат успішного тестування генерації та підтвердження TOTP

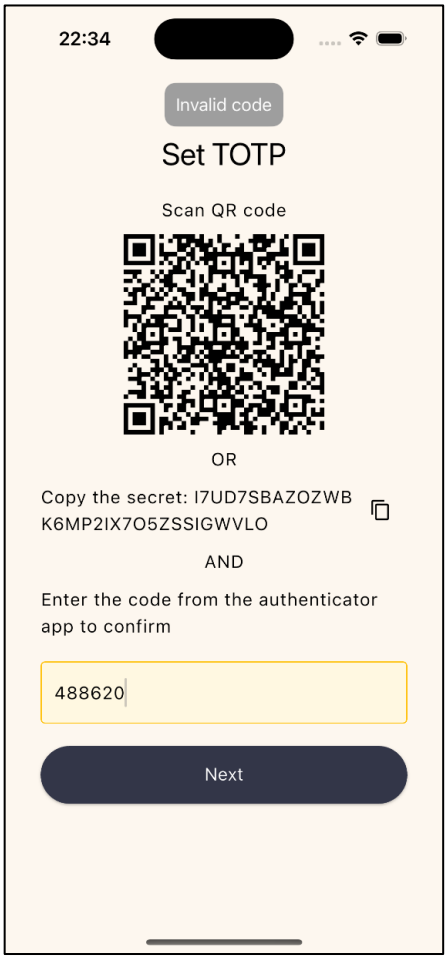



Рисунок 3.5 – Помилка при вводі коду

Тест 3: Вхід та підтвердження TOTP-фактору.

– очікуваний результат: користувач вводить пошту та пароль, його направляє на екран підтвердження коду, при правильному коді користувач попадає на головний екран;
– фактичний результат: Успішно. Користувача направлено на екран підтвердження TOTP, після введення правильного коду його направлено на головний екран (рис. 3.6), при неправильному вводі пошти, паролю або коду з’являються помилки на відповідних екранах (рис. 3.7).

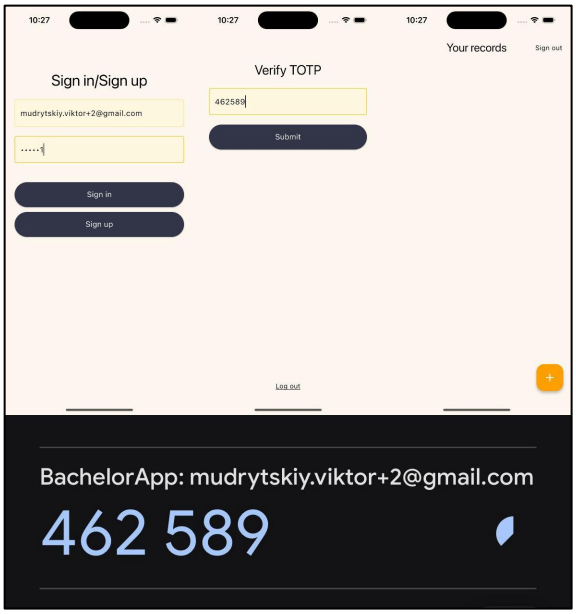



Рисунок 3.6 – Результати тестування входу

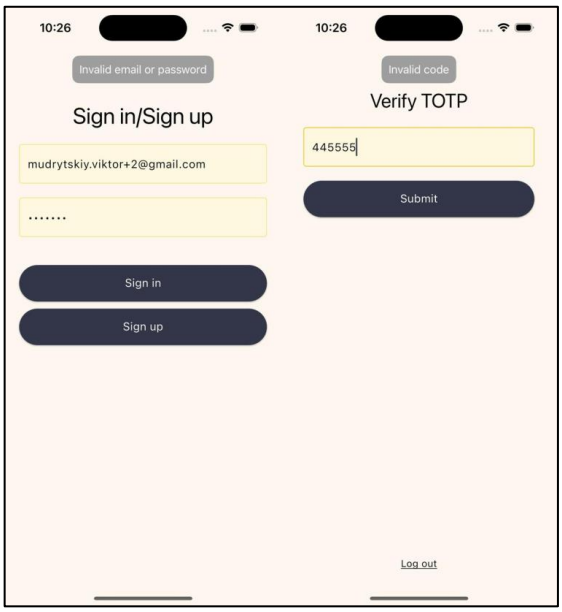



Рисунок 3.7 – Помилки під час входу

Тест 4: Створення та редагування записів.

– очікуваний результат: авторизований користувач створює запис та зберігає його. Запис з’являється на головному екрані. При натисканні на нього користувача направляє на екран редагування, де він може редагувати запис. При повернені на головний екран запис має змінитись. Записи мають зберігатись між повторними заходами в застосунок. Записи мають зберігатись у зашифрованому вигляді;
– фактичний результат: Успішно. Користувач має змогу створювати та редагувати записи, всі зміни відображено на головному екрані. Записи зберігаються у Firestore у зашифрованому вигляді (рис. 3.8). Дані збережені та відображені на повторних входах в систему.

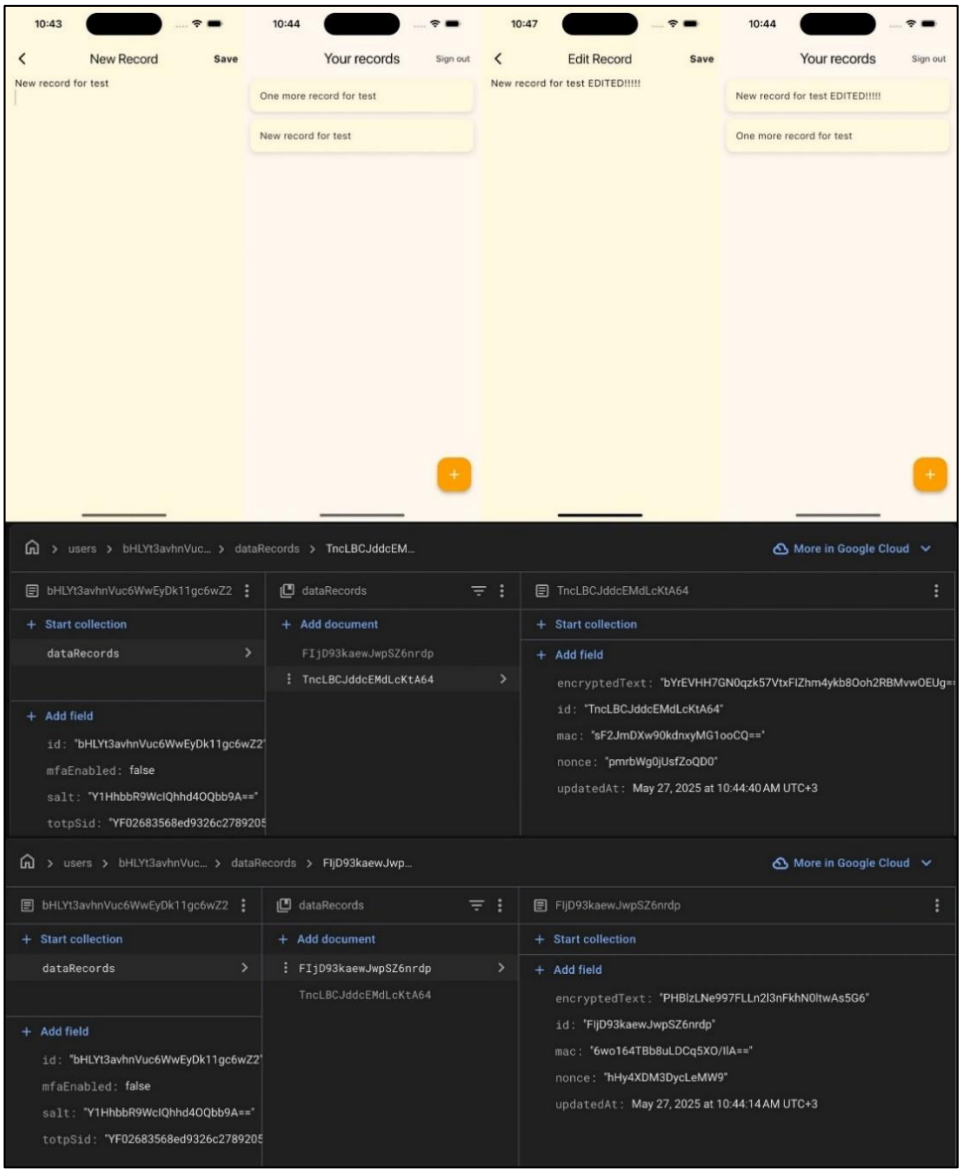



Рисунок 3.8 – Результати тестування створення та редагування записів

Результати ручного інтеграційного тестування засвідчили, що всі основні компоненти застосунку працюють відповідно до очікуваної логіки. Механізми автентифікації, зокрема реалізація багатофакторного захисту через TOTP, функціонують правильно, не допускаючи доступу до чутливої інформації без проходження повної перевірки. Шифрування та розшифрування даних відбуваються локально, у Firestore не зафіксовано збереження відкритих даних. Дії, пов’язані з помилками або некоректними вхідними даними, обробляються з відповідними повідомленнями, без витоку службової інформації. Отже, застосунок демонструє узгоджену, стійку та безпечну поведінку в умовах реального використання.

### 3.4.2 Модульне тестування криптографічної логіки

З метою перевірки коректності окремих елементів було проведено модульне тестування методів класу EncryptioService, зокрема: генерації солі та шифрування і розшифрування даних на основі ключа. Тестування виконувалося за допомогою бібліотеки test у середовищі Dart. Основною функцією перевірки у тестах є expect, з її допомогою вирішується чи тест проходить чи ні.

Вхідні дані для тестів:

- пароль: testPassword123;
- текст: This is a secret note.

Фрагмент тесту на довжину створення випадкової солі:

```
final salt = encryptionService.generateRandomSalt(length: 16);
final bytes = base64Decode(salt);
expect(bytes.length, 16);
```

Цей тест перевіряє, що метод generateRandomSalt() дійсно генерує сіль правильної довжини, і вона коректно кодується у формат Base64.

Фрагмент тесту на генерацію секретного ключа:

```
final salt = encryptionService.generateRandomSalt(length: 16);
      final key1 = await encryptionService.deriveKeyFromPassword(
        password: testPassword,
        salt: salt,
      );
      final key2 = await encryptionService.deriveKeyFromPassword(
        password: testPassword,
        salt: salt,
      );
      expect(key1, equals(key2));
```

Даний тест перевіряє чи з одного й того самого пароля та солі завжди генерується ідентичний криптографічний ключ.

Код тесту на коректне шифрування та розшифрування даних:

```
final salt = encryptionService.generateRandomSalt();
      final key = await encryptionService.deriveKeyFromPassword(
        password: testPassword,
        salt: salt,
      );
      final encrypted = await encryptionService.encryptPlainText(testText,
key);
      final decrypted = await encryptionService.decryptRecord(
        encrData: encrypted,
        key: key,
      );
      expect(decrypted, equals(testText));
```

Тестування перевіряє, що зашифрований текст можна успішно розшифрувати тим самим ключем, отриманим на основі введеного користувачем пароля та випадкової солі.

Фрагмент коду для тесту розшифрування із неправильним ключем:

```
final salt = encryptionService.generateRandomSalt();
      final correctKey = await encryptionService.deriveKeyFromPassword(
        password: testPassword, salt: salt,);
      final wrongKey = await encryptionService.deriveKeyFromPassword(
        password: 'wrongPassword',salt: salt,);
      final encrypted = await encryptionService.encryptPlainText(
        testText,correctKey,);
      expect(
        () async => await encryptionService.decryptRecord(
          encrData: encrypted, key: wrongKey,),throwsException,);
```

Цей тест моделює ситуацію, коли користувач або зловмисник вводить неправильний пароль, унаслідок чого генерується інший ключ. У такому випадку розшифрування має завершитися помилкою.

Результати тестування представлено на рисунку 3.9.

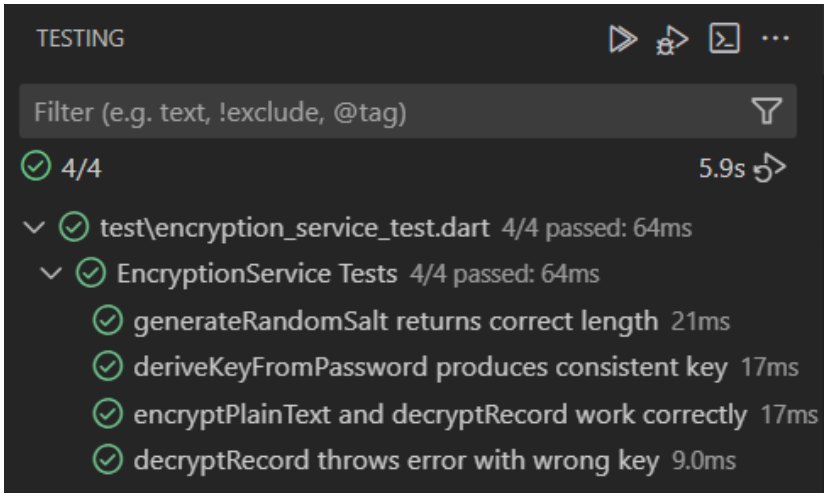



Рисунок 3.9 – Результати модульного тестування

Проведене модульне тестування підтвердило коректність і надійність реалізації ключових криптографічних функцій застосунку. Усі тести показали стабільні результати: генерація солі відбувається з дотриманням заданої довжини та форматування, виведення ключа є детермінованим, а операції шифрування та розшифрування зберігають цілісність даних. Спроби розшифрувати дані з неправильним ключем очікувано завершуються помилкою, що підтверджує захищеність реалізації від несанкціонованого доступу. Отримані результати свідчать про готовність криптографічного модуля до використання в реальних умовах експлуатації

3.4.3 Тестування доступу до Firestore з різними рівнями авторизації

Для перевірки ефективності реалізованих правил безпеки в Firestore було проведено серію тестів, що імітують звернення до бази даних у різних умовах. Основною метою є переконатися, що доступ до захищених документів дозволений лише після проходження повної автентифікації, включно з підтвердженням другого фактору TOTP, і тільки для власного облікового запису.

Тестування виконувалося вручну через Postman, із використанням REST API до Firestore. Запити формувалися з різними токенами авторизації або без них, що дозволило перевірити реакцію системи на несанкціоновані звернення. Усі результати звірялися з активними правилами безпеки Firestore, які визначають доступ на основі ідентифікатора користувача та даних із JWT токена автентифікації.

Сценарій 1: Доступ до даних без токену автентифікації.

У цьому сценарії перевіряється, чи дозволяє Firestore доступ до даних без будь-якої авторизації. Відправлено запит на читання документа з колекції users, без включення токена авторизації.

Очікувана поведінка: Firestore має відхилити запит із повідомленням PERMISSION_DENIED, згідно з правилом:

allow read: if request.auth != null

Результат тестування показано на рисунку 3.10.

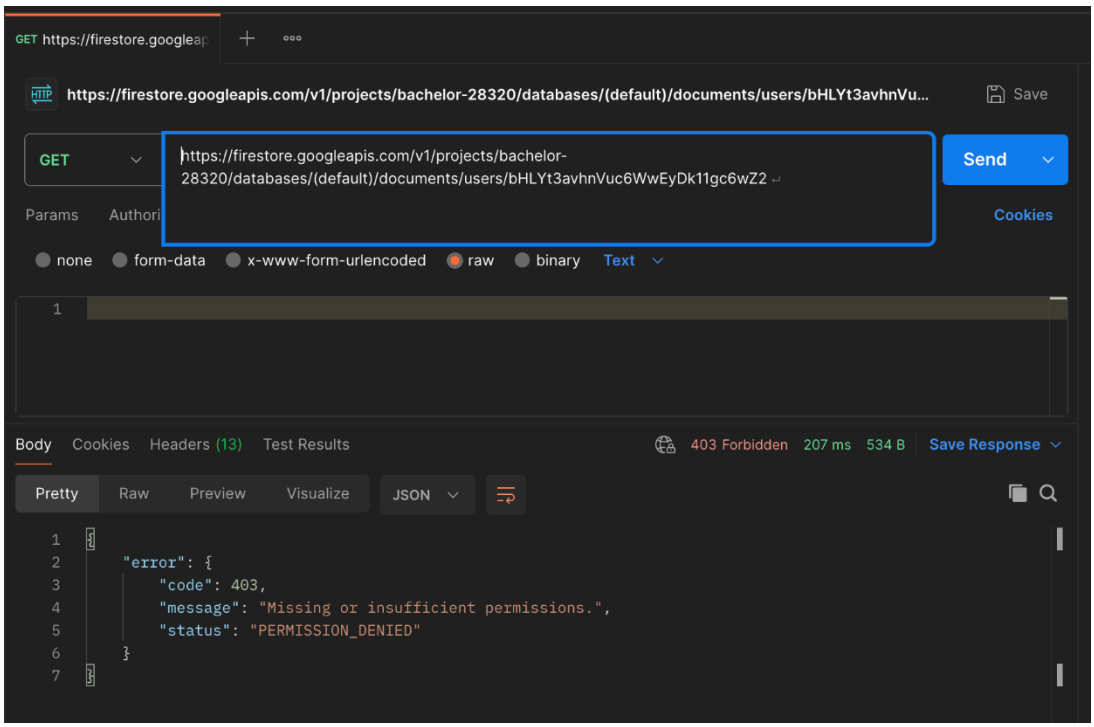



Рисунок 3.10 – Спроба отримати дані без токену автентифікації

Фактичний результат: Отримано статус 403 та помилку PERMISSION_DENIED. Доступ до будь-якої частини документа без автентифікації неможливий.

Базовий захист працює належним чином – сторонній користувач, який не увійшов у систему, не має доступу до Firestore.

Сценарій 2: Авторизований користувач, але без проходження двофакторної автентифікації.

Цей сценарій моделює ситуацію, коли користувач успішно проходить базову автентифікацію за допомогою пошти та пароля, але не підтверджує другий фактор автентифікації. У такому випадку токен авторизації не містить поле mfa_passed, що має призвести до відмови в доступі до вкладеної колекції dataRecords.

Очікувана поведінка: Запит має бути відхилений згідно з умовою правил:

allow read: if request.auth.token.mfa_passed == true

Фактичний результат: Отримано статус 403 та помилку PERMISSION_DENIED. Доступ до даних заблоковано, оскільки токен не містить підтвердження другого фактору (рис. 3.11).

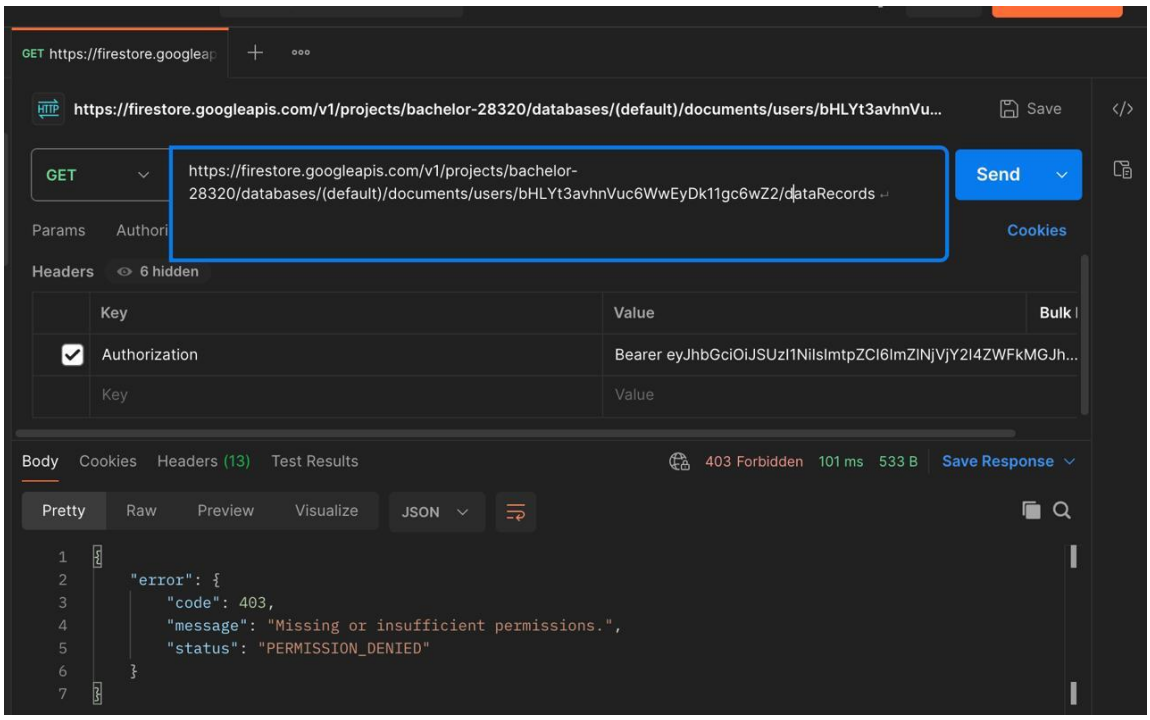



Рисунок 3.11 – Спроба отримати конфіденційні дані, не пройшовши другий фактор автентифікації

Система чітко розділяє базову автентифікацію та багатофакторну. Навіть після успішного входу з email і паролем користувач не має доступу до захищених записів до моменту проходження MFA. Це забезпечує належний рівень контролю доступу до чутливої інформації.

Сценарій 3: Авторизований користувач, який пройшов два фактори автентифікації має доступ до своєї колекції.

Цей сценарій демонструє стандартну та правильну послідовність дій автентифікованого користувача, який успішно пройшов як базову авторизацію, так і підтвердження TOTP. Після підтвердження другого фактору користувачу присвоюється mfa_passed зі значенням true, ці дані додаютьсядо токена автентифікації.

Очікувана поведінка: Firestore має дозволити читання вкладеної колекції dataRecords, через правило:

request.auth.uid == userId &&

request.auth.token.mfa_passed == true

Фактичний результат: Запит успішний. Отримано список записів у підколекції користувача (рис. 3.12).

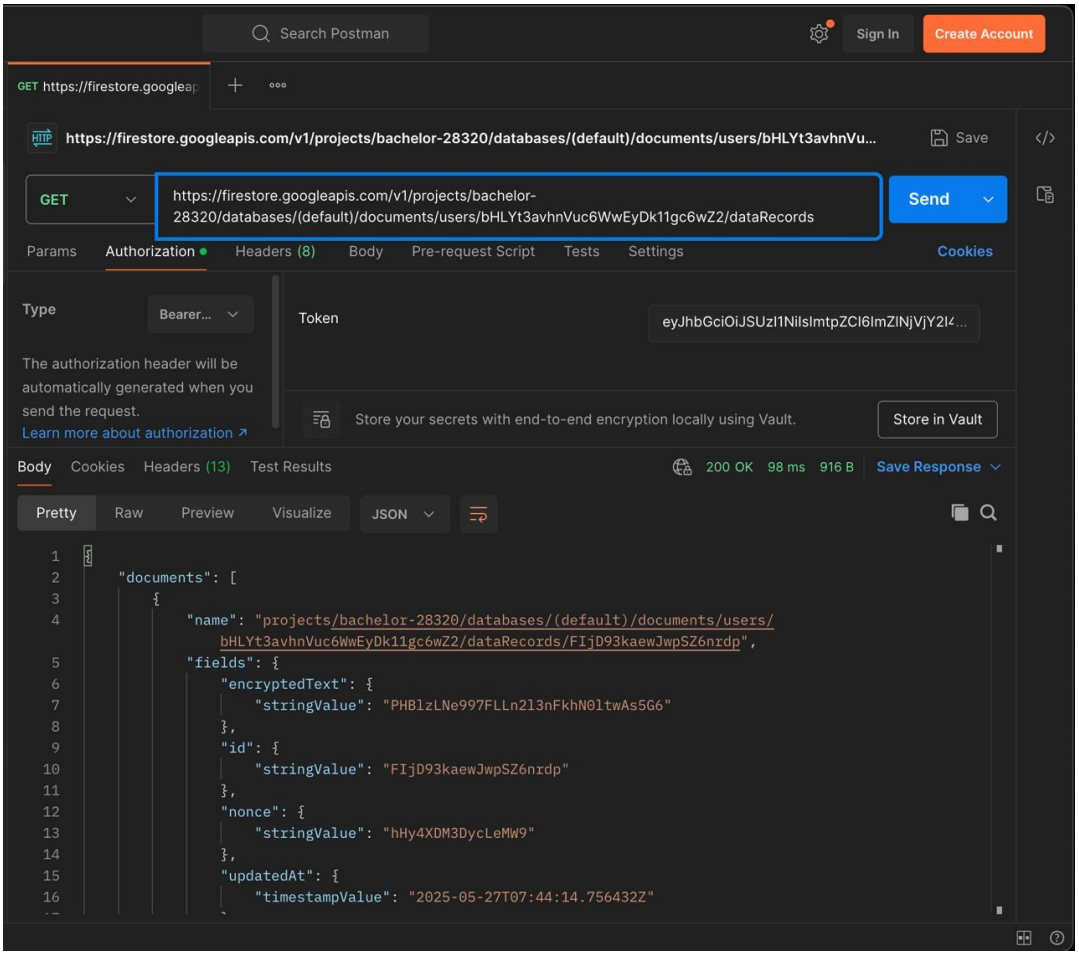



Рисунок 3.12 – Успішне отримання даних із токеном, який містить запис, що підтверджує проходження другого фактору автентифікації

Система дозволяє доступ до конфіденційної інформації лише після повного проходження багатофакторної автентифікації. Вказане поле у токені виконує роль перевіреного маркера довіри.

Сценарій 4: Спроба доступу до даних іншого користувача.

Цей тестовий сценарій моделює спробу автентифікованого користувача, який успішно пройшов двофакторну автентифікацію, отримати доступ до даних іншого користувача. Така дія має бути заборонена незалежно від наявності правильного токену автентифікації, оскільки ідентифікатор користувача у запиті не збігається з ідентифікатором авторизованого користувача.

Очікувана поведінка: Відповідно до правила Firestore доступ має бути заборонено, оскільки request.auth.uid ≠ userId.

Фрагмент правила:

allow read: if request.auth.uid == userId && request.auth.token.mfa_passed == true;

Фактичний результат: Запит було відхилено. Отримано помилку PERMISSION_DENIED (рис. 3.13).

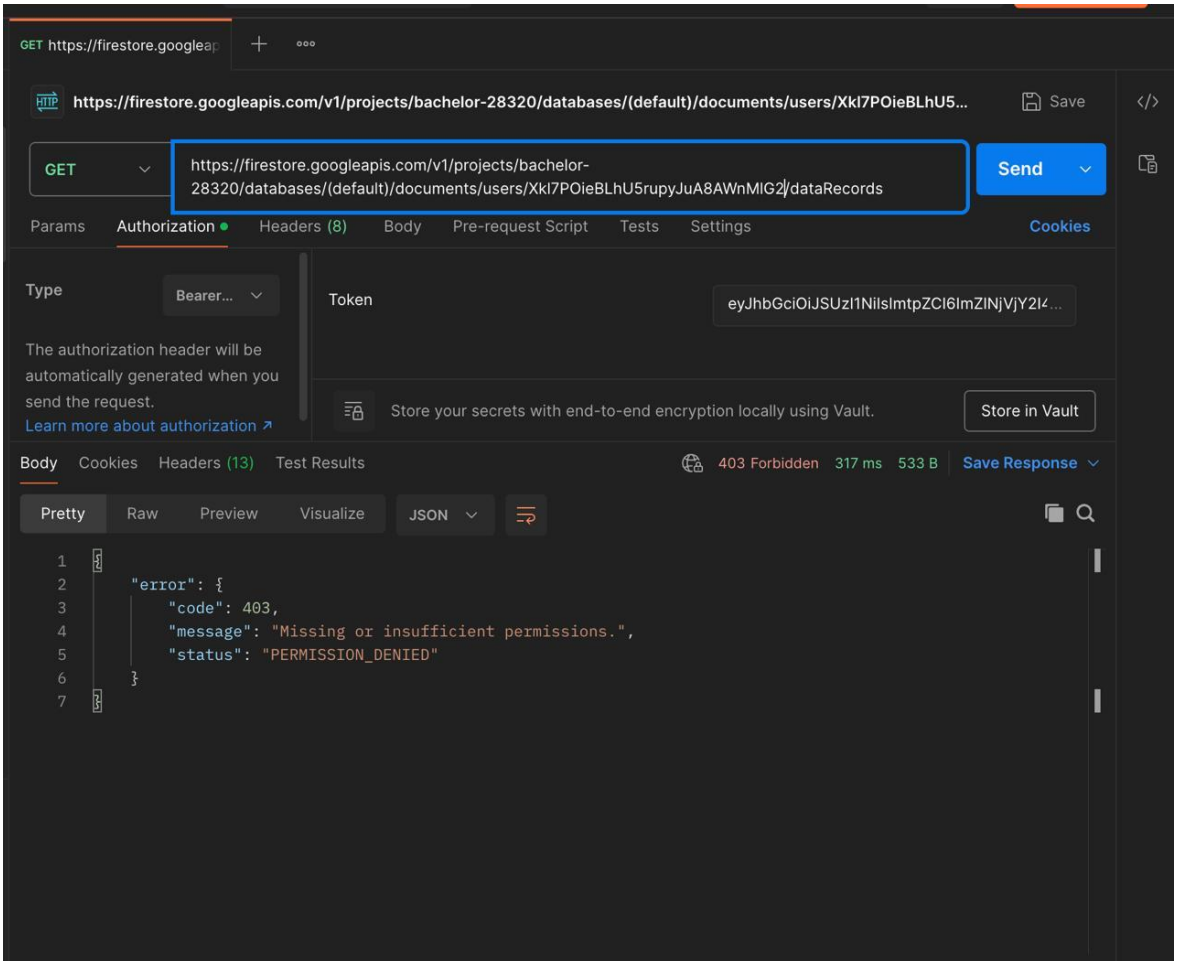



Рисунок 3.13 – Спроба отримання даних іншого користувача

Система обмежує доступ до колекцій лише в межах власного облікового запису користувача. Навіть за умови повної автентифікації, доступ заблоковано.

Результати тестування доступу до Firestore підтвердили правильність реалізації політик безпеки на основі ідентифікації користувача та додаткових даних у токені автентифікації. Усі сценарії були обрані для перевірки умов: відсутність авторизації, часткова автентифікація без проходження другого фактору, повна

автентифікація, а також спроба доступу до чужих даних. У всіх випадках система поводилася згідно з визначеними правилами, доступ дозволявся лише тоді, коли користувач був автентифікований, успішно пройшов багатофакторну перевірку та здійснював звернення до власних даних.

Це підтверджує, що застосовані Firebase Security Rules ефективно забезпечують модель доступу за принципом найменших привілеїв, а параметри у токенах JWT працюють як надійні механізми умовного контролю доступу. Таким чином, система захищає конфіденційну інформацію не лише від зовнішніх користувачів, але й від потенційних внутрішніх порушень.

### 3.5 Висновки до розділу

У третьому розділі було реалізовано захищений мобільний застосунок для зберігання конфіденційних даних, а також виконано повноцінне тестування його функціональності та безпеки. Графічний інтерфейс побудовано на основі Flutter з використанням архітектури BLoC. Всі критичні операції із шифрування та розшифрування даних відбуваються на пристрої користувача. Для генерації криптографічного ключа застосовується алгоритм HKDF, а для симетричного шифрування ChaCha20-Poly1305. Дані зберігаються у Firestore лише у зашифрованому вигляді.

Серверна логіка реалізована через Firebase Cloud Functions на Node.js. Вона забезпечує реєстрацію TOTP факторів, перевірку одноразових кодів і динамічне встановлення додаткових даних у токенах користувачів.

Ручне та модульне тестування показало стабільну роботу всіх компонентів: генерації ключів, шифрування, розшифрування, автентифікації та контролю доступу. Firestore Security Rules ефективно блокують спроби неавторизованого доступу, що підтверджено серією запитів через Postman.

Загалом, застосунок демонструє відповідність сучасним вимогам безпеки до мобільних систем, поєднуючи надійне шифрування, багатофакторну автентифікацію, контрольований доступ до хмарного сховища та захищену взаємодію між клієнтською та серверною частинами.

# ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було розроблено захищений мобільний застосунок для зберігання конфіденційних даних, що базується на сучасних принципах інформаційної безпеки. Основною метою дослідження стало підвищення рівня захисту персональних даних користувачів за рахунок впровадження комбінованих механізмів автентифікації, шифрування та обмеження доступу до зашифрованої інформації.

У першому розділі було здійснено комплексний аналіз сучасних методів захисту даних у мобільних середовищах. Розглянуто методи автентифікації, включно з паролями, біометрією та одноразовими кодами, а також алгоритми симетричного та асиметричного шифрування. Обґрунтовано вибір технологій з урахуванням специфіки мобільних платформ і потреб у високій продуктивності, зручності та стійкості до атак.

Другий розділ було присвячено проєктуванню архітектури мобільного застосунку. Запропонована модель поєднує локальне симетричне шифрування, виведення ключів з секретної інформації користувачів, а також зберігання зашифрованих даних віддалено. Автентифікація реалізована з використанням двох факторів.

У третьому розділі було розглянуто процес реалізації та тестування застосунку. Проведено серію інтеграційних, модульних і авторизаційних тестів, які підтвердили працездатність та відповідність реалізованого рішення вимогам безпеки.

Загалом, реалізований захищений мобільний застосунок підтверджує доцільність використання сучасних методів автентифікації та шифрування на практиці. Отримані результати та впроваджені рішення сприяють підвищенню безпеки мобільних застосунків, зменшенню ризику несанкціонованого доступу до конфіденційної інформації та зміцненню загальної стійкості системи до атак.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Flutter Local Storage vs Cloud Storage. URL: https://medium.com/easy-flutter/flutter-local-storage-vs-cloud-storage-fad49cb3dd2f (accessed: 07.03.2025).
2. Гарнага В. А. Використання підходів Zero Trust Architecture при розробці мобільних додатків. Матеріали конференції *Молодь в науці: дослідження, проблеми, перспективи* ВНТУ. Вінниця : ВНТУ, 2025. 4 с. URL: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25380 (дата звернення: 21.03.2025).
3. PHP Українське керівництво. URL: https://php.org.ua/manual/uk/faq.passwords.md (дата звернення: 23.03.2025).
4. TOTP vs. HOTP: Which one should you use. URL: https://www.descope.com/blog/post/totp-vs-hotp (accessed: 21.04.2025).
5. TOTP – Time-based One-Time Password. URL: https://www.twilio.com/docs/glossary/totp (accessed: 23.04.2025).
6. What Does OTP Mean? URL: https://www.twilio.com/en-us/blog/what-does-otp-mean (accessed: 28.04.2025).
7. Symmetric Key Encryption – Why, Where and How it’s Used in Banking. URL: https://www.cryptomathic.com/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking (accessed: 29.04.2025).
8. PBKDF2 Hashing Algorithm. URL: https://nishothan-17.medium.com/pbkdf2-hashing-algorithm-841d5cc9178d (accessed: 30.04.2025).
9. What is Asymmetric Encryption? URL: https://www.geeksforgeeks.org/what-is-asymmetric-encryption/ (accessed: 01.05.2025).
10. Zero Knowledge Security Model. URL: https://www.lastpass.com/security/zero-knowledge-security (accessed: 03.05.2025).
11. Add multi-factor authentication to your web app. URL: https://firebase.google.com/docs/auth/web/multi-factor (accessed: 05.05.2025).

12. ChaCha20-Poly1305 Authenticated Encryption. URL: https://www.adalabs.com/adalabs-chacha20poly1305.pdf (accessed: 06.05.2025).
13. Функція виведення ключа. URL: https://www.vpnunlimited.com/ua/help/cybersecurity/key-derivation-function (дата звернення: 07.05.2025).
14. Create custom tokens. URL: https://firebase.google.com/docs/auth/admin/create-custom-tokens (accessed: 07.05.2025).
15. Verify TOTP Quickstart. URL: https://www.twilio.com/docs/verify/quickstarts/totp (accessed: 09.05.2025).
16. Understanding AES Encryption and AES-GCM Mode: An In-depth Exploration Using Java. URL: https://medium.com/@pravallikayakkala123/understanding-aes-encryption-and-aes-gcm-mode-an-in-depth-exploration-using-java-e03be85a3faa (accessed: 10.05.2025).
17. Jay P. Cipher Suites AEAD – ChaCha20-Poly1305 Example. URL: https://medium.com/@jaypmedia/cipher-suites-aead-chacha20-poly1305-example-812a609229f7 (accessed: 13.05.2025).
18. Криптографія для розробників. URL: https://medium.com/@jstify.community/криптографія-для-розробників-chapter-5-26a83199ebc8 (дата звернення: 14.05.2025).
19. Getting to Know HKDF. URL: https://nearform.com/insights/getting-to-know-hkdf/ (accessed: 14.05.2025).
20. Apple Developer. Keychain Services. URL: https://developer.apple.com/documentation/security/keychain-services (accessed 15.05.2025).
21. Android Developers. Android Keystore System. URL: https://developer.android.com/privacy-and-security/keystore (accessed: 16.05.2025).
22. Cloud Firestore Documentation. URL: https://firebase.google.com/docs/firestore (accessed: 17.05.2025).

23. Що таке формат даних Base64. URL: https://iris.svoboda.cx.ua/ukraincyam/shho-take-format-danikh-base64.html (дата звернення: 17.05.2025).
24. Get data with Cloud Firestore. URL: https://firebase.google.com/docs/firestore/query-data/get-data (accessed: 18.05.2025).
25. Cloud Functions for Firebase Documentation. URL: https://firebase.google.com/docs/functions (accessed: 20.05.2025).
26. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт. Вінниця: Вінницький національний технічний університет, 2023. 62 с. URL: https://iq.vntu.edu.ua//fm/fdb/775/BDR/MB_БКР_2023.pdf

# ДОДАТКИ

# Додаток А

## Код програми

auth_repo.dart

```
import 'dart:async';
import 'dart:convert';

import 'package:cloud_functions/cloud_functions.dart';
import 'package:confidential_notes/lib.dart';
import 'package:firebase_auth/firebase_auth.dart';

enum UserAuthStatus { loggedIn, enrolledMFA, passedMFA, loggedOut }

class AuthRepo with LoggerMixin {
  AuthRepo() {
    _controller = StreamController.broadcast();
    _authChangeSubscription = _auth.authStateChanges().listen(
      _authStatusListener,
    );
  }

  final FirebaseAuth _auth = FirebaseAuth.instance;
  final FirebaseFunctions _functions = FirebaseFunctions.instance;

  late StreamController<UserAuthStatus> _controller;
  StreamSubscription<User?>? _authChangeSubscription;

  Stream<UserAuthStatus> get authStatus => _controller.stream;
  Future<UserAuthStatus> get lastStatus => _controller.stream.last;
  User? get currentUser => _auth.currentUser;

  /// Throws if user is not logged in
  String get uid => _auth.currentUser!.uid;

  bool _ignoreEvent = true;

  Future<void> reloadUser() async {
    if (currentUser == null) {
      return;
    }
    await _auth.currentUser!.reload();
    logSuccess('User reloaded');
  }

  Future<void> _authStatusListener(User? user) async {
    if (_ignoreEvent) {
      _ignoreEvent = false;
      return;
    }
    final status = await checkUserSatatus();
    _controller.add(status);
  }

  Future<UserAuthStatus> checkUserSatatus() async {
    final idToken = await _auth.currentUser?.getIdToken(true);
    if (idToken == null) {
      logSuccess('User is logged out');
      return UserAuthStatus.loggedOut;
    }
```

```
    final parts = idToken.split('.');
    if (parts.length != 3) {
      logSuccess('User jwt is invalid');
      return UserAuthStatus.loggedOut;
    }

    final payload = utf8.decode(
      base64Url.decode(base64Url.normalize(parts[1])),
    );
    final claims = json.decode(payload);
    final hasEnrolledMfa = (claims['mfa_enrolled'] as bool?) ?? false;
    final hasPassedMfa = (claims['mfa_passed'] as bool?) ?? false;
    UserAuthStatus status;
    if (hasPassedMfa) {
      status = UserAuthStatus.passedMFA;
    } else if (hasEnrolledMfa) {
      status = UserAuthStatus.enrolledMFA;
    } else {
      status = UserAuthStatus.loggedIn;
    }

    logSuccess(
      'User is logged in Firebase ${_auth.currentUser?.uid}, TOTP status:
$status',
    );
    return status;
  }

  Future<String?> login({
    required String email,
    required String password,
  }) async {
    final result = await FirebaseFunctions.instance
        .httpsCallable('loginUserAndResetMFA')
        .call({'email': email, 'password': password});

    final customToken = result.data['customToken'];

    final userCredential = await FirebaseAuth.instance.signInWithCustomToken(
      customToken,
    );

    await userCredential.user?.getIdToken(true);
    logSuccess('User logged in: $email');
    return userCredential.user?.uid;
  }

  Future<String?> registerUser({
    required String password,
    required String email,
  }) async {
    final userCredential = await _auth.createUserWithEmailAndPassword(
      email: email,
      password: password,
    );

    final uid = userCredential.user!.uid;

    logSuccess('User registered: $email, $uid');
    return uid;
  }

  Future<void> logout() async {
```

```
    await _auth.signOut();
    logSuccess('User logged out');
  }

  Future<EnrollTotpResponse> enrollTotp() async {
    final result = await _functions.httpsCallable('createTOTP').call({
      'userIdentifier': uid,
      'factorName': currentUser!.email,
    });
    logSuccess('TOTP created (unverified): ${result.data}');
    return EnrollTotpResponse.fromJson(result.data);
  }

  Future<void> verifyTotp({
    required String code,
    required String verifSid,
  }) async {
    final result = await _functions
        .httpsCallable('confirmTOTPAndSetClaim')
        .call({
          'userIdentifier': uid,
          'verificationSid': verifSid,
          'code': code,
        });
    logSuccess('TOTP verified for $uid: ${result.data}');
  }

  Future<bool> createChallenge({
    required String code,
    required String verifSid,
  }) async {
    final result = await _functions.httpsCallable('createChallenge').call({
      'userIdentifier': uid,
      'factorSid': verifSid,
      'code': code,
    });
    final isSuccess = result.data['success'];
    logSuccess('TOTP challenge result: $isSuccess');
    return isSuccess;
  }

  void dispose() {
    _controller.close();
    _authChangeSubscription?.cancel();
  }
}
```

## auth_bloc.dart

```
import 'dart:async';
import 'dart:developer';

import 'package:confidential_notes/lib.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:freezed_annotation/freezed_annotation.dart';

part 'auth_bloc.freezed.dart';
part 'auth_event.dart';
part 'auth_state.dart';

class AuthBloc extends Bloc<AuthEvent, AuthState>
```

```
  with OverlayMixin, NavigaionMixin, LoggerMixin {
final AuthRepo _authRepo;

final SecureStorageService _secureStorageService;
final GlobalKey<NavigatorState> _navigatorKey;

AuthBloc({
  required AuthRepo authRepo,
  required SecureStorageService secureStorageService,
  required GlobalKey<NavigatorState> navigatorKey,
}) : _authRepo = authRepo,
     _navigatorKey = navigatorKey,
     _secureStorageService = secureStorageService,
     super(_Initial()) {
  on<AuthEvent>(_mapper);
}

Future<void> _mapper(AuthEvent event, Emitter<AuthState> emit) async {
  await event.map(
    signIn: (event) async => await _signIn(event, emit),
    signUp: (event) async => await _signUp(event, emit),
    resolveState: (event) async => await _resolveState(event, emit),
    signOut: (event) async => await _signOut(event, emit),
    init: (event) async => await _init(event, emit),
  );
}

Future<void> _init(_Init event, Emitter<AuthState> emit) async {
  try {
    add(AuthEvent.signOut());
    replaceAllWithKey(key: _navigatorKey, page: const AuthScreen());

    // add(AuthEvent.resolveState(status: status));
  } catch (e) {
    showErrorToast('Failed to initialize auth state');
    log(e.toString());
    emit(_LoggedOut());
  }
}

Future<void> _signIn(_SignIn event, Emitter<AuthState> emit) async {
  emit(_Processing());
  try {
    await _authRepo.login(email: event.email, password: event.password);
    final status = await _authRepo.checkUserSatatus();
    add(AuthEvent.resolveState(status: status, password: event.password));
  } catch (e) {
    showErrorToast('Invalid email or password');
    emit(_LoggedOut());
  }
}

Future<void> _signUp(_SignUp event, Emitter<AuthState> emit) async {
  emit(_Processing());
  try {
    await _authRepo.registerUser(
      email: event.email,
      password: event.password,
    );
    add(
      AuthEvent.resolveState(
        status: UserAuthStatus.loggedIn,
        password: event.password,
      ),
```

```
    );
  } catch (e) {
    showErrorToast(
      'Email is incorrect or already in use or password is too weak',
    );
    emit(_LoggedOut());
  }
}

Future<void> _resolveState(_Resolve event, Emitter<AuthState> emit) async {
  try {
    final bool shouldAnimate = state.maybeMap(
      orElse: () => true,
      initial: (value) => false,
    );
    switch (event.status) {
      case UserAuthStatus.loggedIn:
        emit(
          AuthState.loggedIn(
            user: _authRepo.currentUser!,
            password: event.password,
          ),
        );
        replaceAllWithKey(
          key: _navigatorKey,
          page: const EnrollTotpScreen(),
          animate: shouldAnimate,
        );

        break;
      case UserAuthStatus.loggedOut:
        replaceAllWithKey(
          key: _navigatorKey,
          page: const AuthScreen(),
          animate: shouldAnimate,
        );
        emit(AuthState.loggedOut());
        break;
      case UserAuthStatus.enrolledMFA:
        emit(
          AuthState.enrolledMFA(
            user: _authRepo.currentUser!,
            password: event.password,
          ),
        );
        replaceAllWithKey(
          key: _navigatorKey,
          page: const VerifyTotpScreen(),
          animate: shouldAnimate,
        );

        break;
      case UserAuthStatus.passedMFA:
        replaceAllWithKey(
          key: _navigatorKey,
          page: const HomeScreen(),
          animate: shouldAnimate,
        );
        emit(AuthState.passedMFA(user: _authRepo.currentUser!));
        break;
    }
  } catch (e) {
    showErrorToast('Failed to resolve auth state');
    log(e.toString());
```

```
      replaceAllWithKey(key: _navigatorKey, page: const AuthScreen());
      emit(AuthState.loggedOut());
    }
  }

  Future<void> _signOut(_SignOut event, Emitter<AuthState> emit) async {
    emit(_Processing());
    try {
      await _authRepo.logout();
      await _secureStorageService.deleteKey();
      final status = await _authRepo.checkUserSatatus();
      add(AuthEvent.resolveState(status: status));

      emit(_LoggedOut());
    } catch (e) {
      showErrorToast('Failed to sign out');
      emit(_LoggedOut());
    }
  }
}
```

## data_records_bloc.dart

```
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:freezed_annotation/freezed_annotation.dart';

import '../../../lib.dart';

part 'data_records_bloc.freezed.dart';
part 'data_records_event.dart';
part 'data_records_state.dart';

class DataRecordsBloc extends Bloc<DataRecordsEvent, DataRecordsState>
    with OverlayMixin {
  final DataRecordRepo _dataRecordRepo;
  final EncryptionService _encryptionService;
  final SecureStorageService _secureStorageService;
  DataRecordsBloc({
    required DataRecordRepo dataRecordRepo,
    required EncryptionService encryptionService,
    required SecureStorageService secureStorageService,
  }) : _dataRecordRepo = dataRecordRepo,
       _encryptionService = encryptionService,
       _secureStorageService = secureStorageService,
       super(_Loading()) {
    on<DataRecordsEvent>(_mapper);
  }

  Future<void> _mapper(
    DataRecordsEvent event,
    Emitter<DataRecordsState> emit,
  ) async {
    await event.map(
      fetchAll: (_FetchAll value) async => await _fetchAll(value, emit),
      setRecord: (_SetRecord value) async => await _setRecord(value, emit),
      deleteRecord:
          (_DeleteRecord value) async => await _deleteRecord(value, emit),
    );
  }

  Future<void> _fetchAll(
    _FetchAll event,
    Emitter<DataRecordsState> emit,
```

```
) async {
  try {
    emit(const DataRecordsState.loading());
    final encrypted = await _dataRecordRepo.getDataRecords();
    final records = <DataRecordModel>[];
    for (final rec in encrypted) {
      final decrypted = await _encryptionService.decryptRecord(
        encrData: EncryptionData(
          encryptedText: rec.encryptedText,
          nonce: rec.nonce,
          mac: rec.mac,
        ),
        key: await _secureStorageService.getKey() ?? '',
      );
      records.add(rec.copyWith(encryptedText: decrypted));
    }

    emit(DataRecordsState.idle(records: records));
  } catch (e) {
    showErrorToast('Failed to fetch records');
  }
}

Future<void> _setRecord(
  _SetRecord event,
  Emitter<DataRecordsState> emit,
) async {
  try {
    var recToSet = event.record;
    if (!event.isEdit || recToSet.id.isEmpty) {
      recToSet = recToSet.copyWith(id: _dataRecordRepo.createDataRecordId());
      _insertInFirstAndUpdate(recToSet, emit);
    } else {
      _insertInFirstAndUpdate(recToSet, emit);
    }
    recToSet = recToSet.copyWith(updatedAt: DateTime.now());
    final encrypted = await _encryptionService.encryptPlainText(
      recToSet.encryptedText,
      await _secureStorageService.getKey() ?? '',
    );
    await _dataRecordRepo.setDataRecord(
      recToSet.copyWith(
        encryptedText: encrypted.encryptedText,
        nonce: encrypted.nonce,
        mac: encrypted.mac,
      ),
    );
  } catch (e) {
    showErrorToast('Failed to set the record');
  }
}

Future<void> _deleteRecord(
  _DeleteRecord event,
  Emitter<DataRecordsState> emit,
) async {
  try {
    _deleteRecordList(event.id, emit);
    await _dataRecordRepo.deleteDataRecord(event.id);
  } catch (e) {
    showErrorToast('Failed to delete the record');
  }
}
```

```
  void _insertInFirstAndUpdate(
    DataRecordModel record,
    Emitter<DataRecordsState> emit,
  ) {
    state.mapOrNull(
      idle: (idle) {
        final iOfRec = idle.records.indexWhere((e) => e.id == record.id);
        if (iOfRec == -1) {
          final records = [record, ...idle.records];
          emit(DataRecordsState.idle(records: records));
          return;
        }

        final newRecords = List<DataRecordModel>.from(idle.records);
        newRecords.removeAt(iOfRec);
        newRecords.insert(0, record);
        emit(DataRecordsState.idle(records: newRecords));
      },
    );
  }

  void _deleteRecordList(String id, Emitter<DataRecordsState> emit) {
    state.mapOrNull(
      idle: (idle) {
        final records = idle.records.where((e) => e.id != id).toList();
        emit(DataRecordsState.idle(records: records));
      },
    );
  }
}
```

## cloud_functions.ts

```
import * as admin from "firebase-admin";
import * as functions from "firebase-functions/v1";
import { user } from "firebase-functions/v1/auth";
import twilio from "twilio";

const firebaseAdmin = admin.initializeApp({
  credential: admin.credential.cert("./bachelor-28320-b04353db3706.json"),
});

export const createTOTP = functions.https.onCall(
  async (data, context: functions.https.CallableContext) => {
    // Ensure user is authenticated
    const uid = context.auth?.uid;
    if (!uid) {
      throw new functions.https.HttpsError(
        "unauthenticated",
        "User must be authenticated."
      );
    }

    const { userIdentifier, factorName } = data;

    if (!userIdentifier) {
      throw new functions.https.HttpsError(
        "invalid-argument",
        "User identifier is required."
      );
    }
```

```
    try {
      const newFactor = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .newFactors.create({
          factorType: "totp",
          friendlyName: factorName,
        });

      return {
        success: true,
        message: "TOTP setup initiated. Scan QR in an Authenticator app.",
        verificationSid: newFactor.sid,
        secret: newFactor.binding["secret"],
      };
    } catch (error: any) {
      throw new functions.https.HttpsError("internal", error.message);
    }
  }
);

export const confirmTOTPAndSetClaim = functions.https.onCall(
  async (data, context: functions.https.CallableContext) => {
    // Ensure user is authenticated
    const uid = context.auth?.uid;
    if (!uid) {
      throw new functions.https.HttpsError(
        "unauthenticated",
        "User must be authenticated."
      );
    }

    const { userIdentifier, verificationSid, code } = data;

    if (!userIdentifier) {
      throw new functions.https.HttpsError(
        "invalid-argument",
        "User identifier is required."
      );
    }

    try {
      const resp = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .factors(verificationSid)
        .update({ authPayload: code });

      if (resp.status !== "verified") {
        throw new functions.https.HttpsError(
          "internal",
          "TOTP code verification failed."
        );
      }

      await firebaseAdmin
        .auth()
        .setCustomUserClaims(uid, { mfa_enrolled: true, mfa_passed: true });

      return {
        success: true,
        message: "TOTP code verified successfully.",
      };
    } catch (error: any) {
```

```
      throw new functions.https.HttpsError("internal", error.message);
    }
  }
);

export const loginUserAndResetMFA = functions.https.onCall(
  async (data, context) => {
    const { email, password } = data;

    if (!email || !password) {
      throw new functions.https.HttpsError(
        "invalid-argument",
        "Email and password are required."
      );
    }

    const verifyRes = await fetch(
      `https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=$
{FIREBASE_API_KEY}`,
      {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
          email,
          password,
          returnSecureToken: true,
        }),
      }
    );

    if (!verifyRes.ok) {
      throw new Error("Invalid credentials");
    }

    try {
      const userRecord = await admin.auth().getUserByEmail(email);
      const customToken = await admin.auth().createCustomToken(userRecord.uid);

      const currentClaims =
        (await admin.auth().getUser(userRecord.uid)).customClaims || {};

      const updatedClaims = {
        mfa_passed: false,
        mfa_enrolled: currentClaims.mfa_enrolled ?? false,
      };

      await admin.auth().setCustomUserClaims(userRecord.uid, updatedClaims);

      return {
        success: true,
        message: "User authenticated, MFA reset.",
        customToken: customToken,
      };
    } catch (error: any) {
      throw new functions.https.HttpsError(
        "unauthenticated",
        "Authentication failed: " + error.message
      );
    }
  }
);

export const createChallenge = functions.https.onCall(
  async (data, context: functions.https.CallableContext) => {
```

```
    // Ensure user is authenticated
    const uid = context.auth?.uid;
    if (!uid) {
      throw new functions.https.HttpsError(
        "unauthenticated",
        "User must be authenticated."
      );
    }

    const { userIdentifier, factorSid, code } = data;

    if (!userIdentifier) {
      throw new functions.https.HttpsError(
        "invalid-argument",
        "User identifier is required."
      );
    }

    try {
      const challenge = await client.verify.v2
        .services(TWILIO_VERIFY_SERVICE_SID)
        .entities(userIdentifier)
        .challenges.create({
          authPayload: code,
          factorSid: factorSid,
        });

      if (challenge.status === "approved") {
        const updatedClaims = {
          mfa_passed: true,
          mfa_enrolled: true,
        };

        await admin.auth().setCustomUserClaims(userIdentifier, updatedClaims);
      }

      return {
        success: challenge.status === "approved",
        message: "Push challenge initiated. Check your device.",
      };
    } catch (error: any) {
      throw new functions.https.HttpsError("internal", error.message);
    }
  }
);
```

79

**Додаток Б**

**ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ**

Назва роботи: Захищений мобільний застосунок для зберігання конфіденційних даних

Автор роботи: Мудрицький Віктор Васильович

Тип роботи: бакалаврська кваліфікаційна робота

Підрозділ кафедра захисту інформації ФІТКІ, група 1 БС-21б

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism **0.49** %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

√ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Завідувач кафедри ЗІ д. т. н., проф. Володимир ЛУЖЕЦЬКИЙ

Гарант освітньої програми «Безпека інформаційних та комунікаційних систем» к. т. н., доц. Леонід КУПЕРШТЕЙН

Особа, відповідальна за перевірку Валентина КАПЛУН

З висновком експертної комісії ознайомлений(-на)

Керівник Володимир ГАРНАГА

Здобувач Віктор МУДРИЦЬКИЙ

80

**Додаток В**

**ІЛЮСТРАТИВНА ЧАСТИНА**

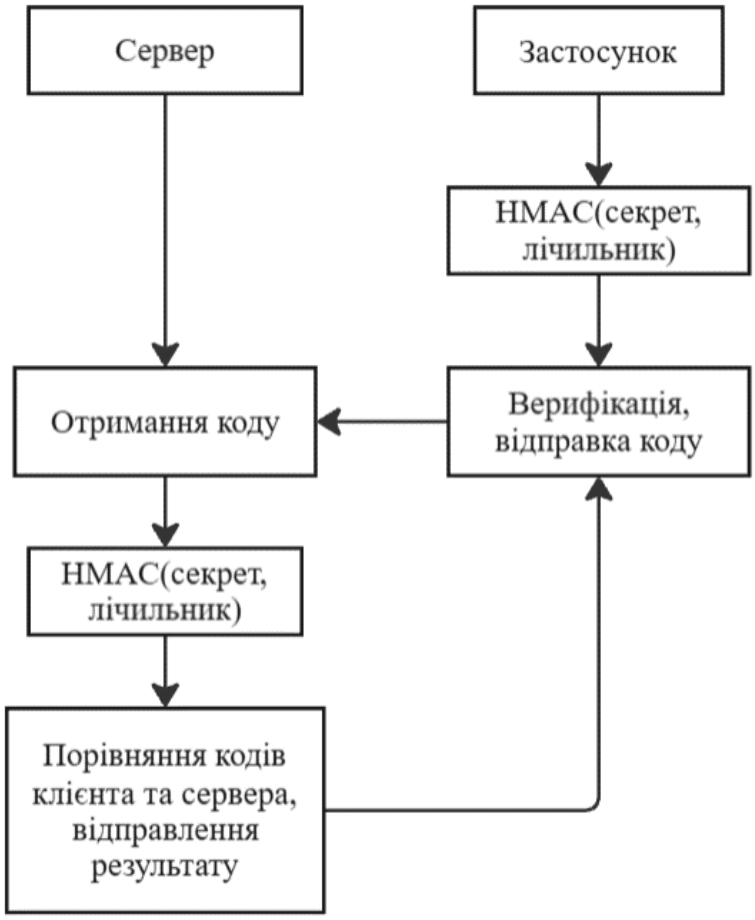
ЗАХИЩЕНИЙ МОБІЛЬНИЙ ЗАСТОСУНОК ДЛЯ ЗБЕРІГАННЯ КОНФІДЕНЦІЙНИХ ДАНИХ

Виконав: студент 4 курсу групи 1БС-21б
спеціальності 125 Кібербезпека
_____ Віктор МУДРИЦЬКИЙ
13 червня 2025 р.

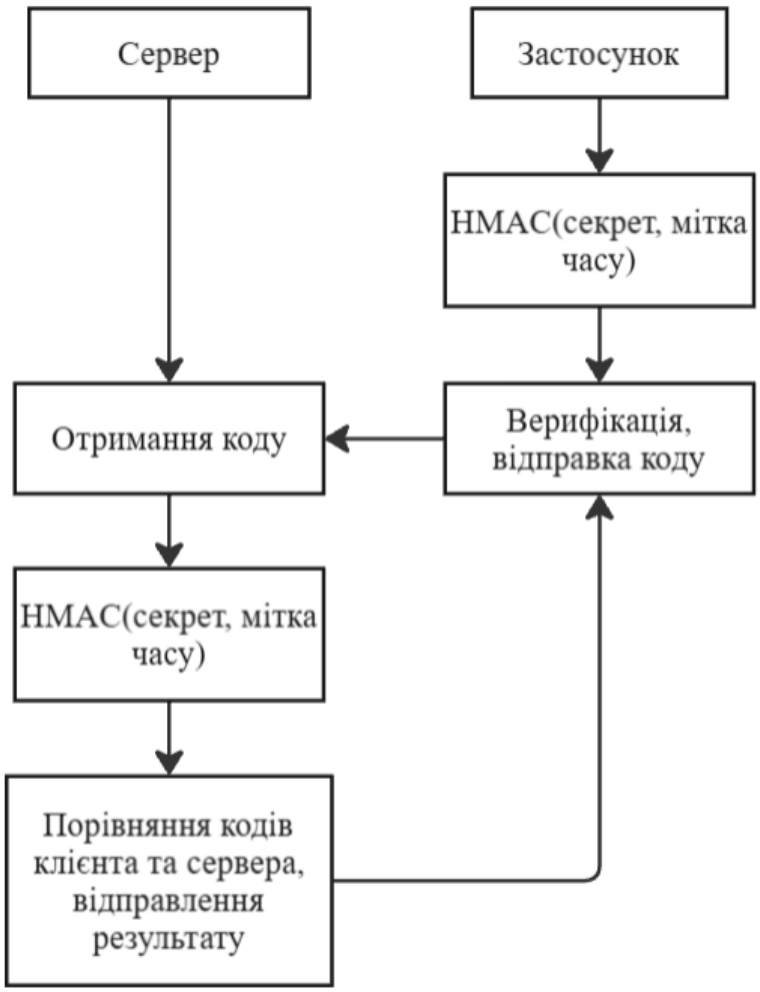
Керівник: к. т. н., доцент каф. ЗІ
_____ Володимир ГАРНАГА
13 червня 2025 р.

# СХЕМА БЕЗПЕЧНОГО ПЕРЕДАВАННЯ ПАРОЛЮ НА СЕРВЕР

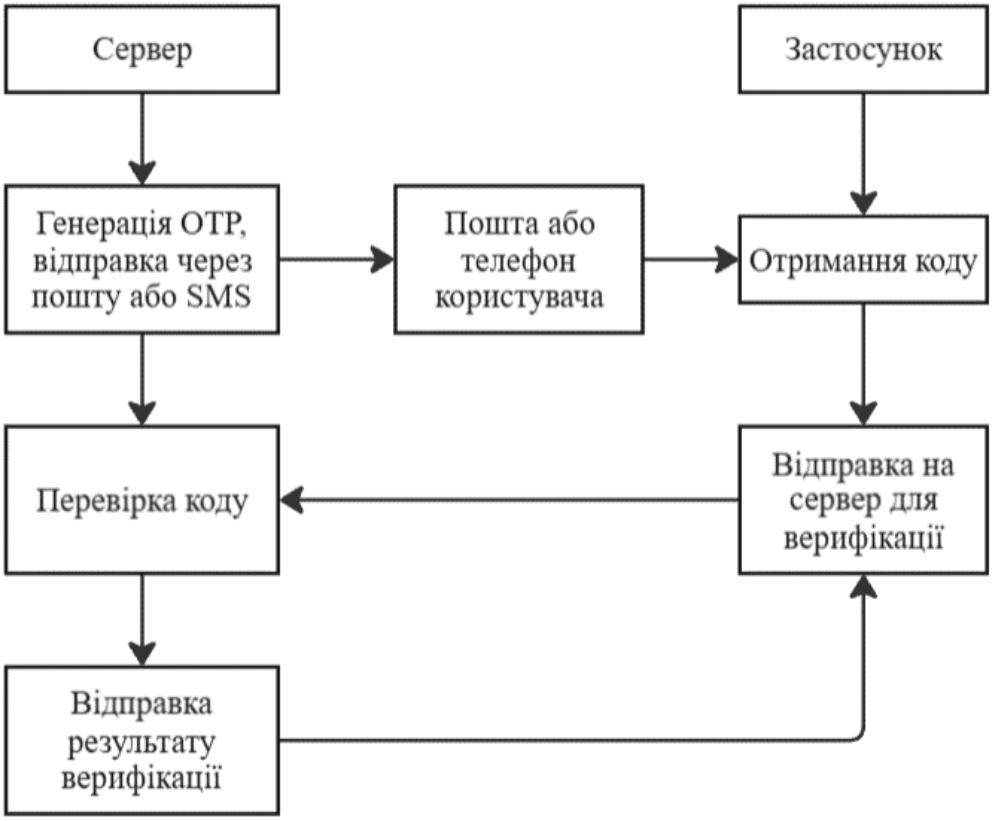
# СХЕМА РЕАЛІЗАЦІЇ HOTP



# СХЕМА РЕАЛІЗАЦІЇ ТОТР



# СХЕМА РЕАЛІЗАЦІЇ ОТР



# СХЕМА ПЕРЕДАВАННЯ ПАРОЛІВ У СИСТЕМІ LASTPASS

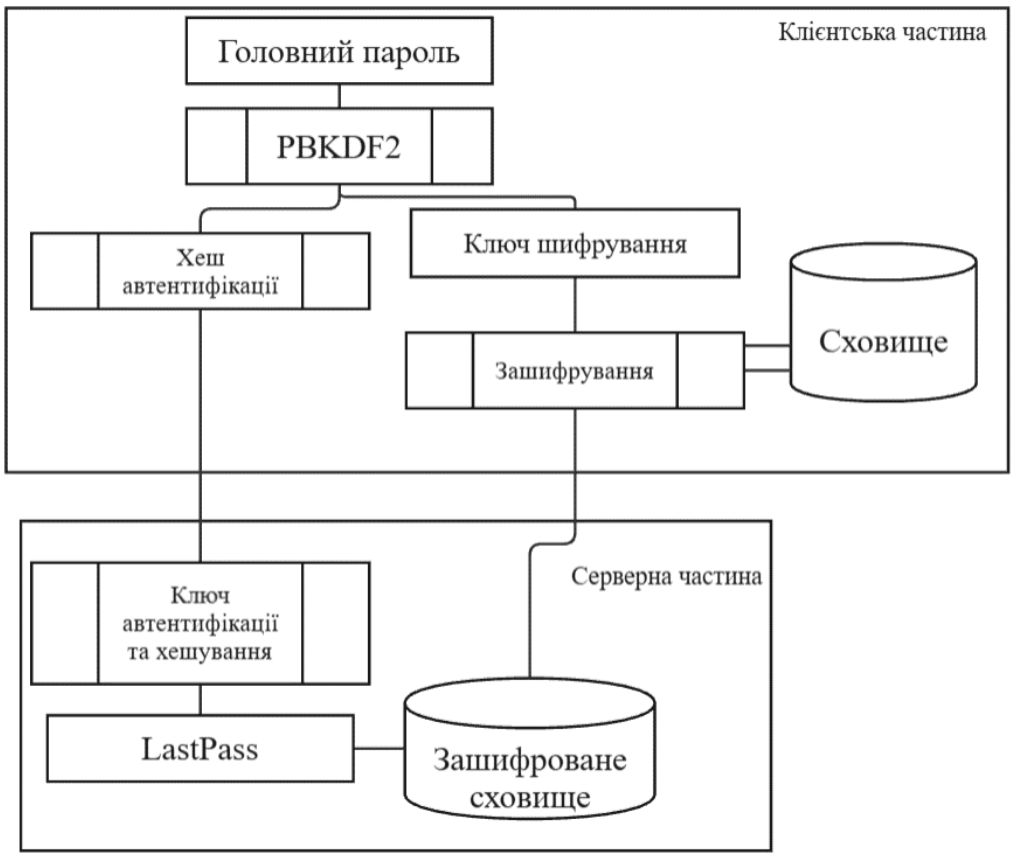


**СХЕМА РЕЄСТРАЦІЇ КОРИСТУВАЧА**

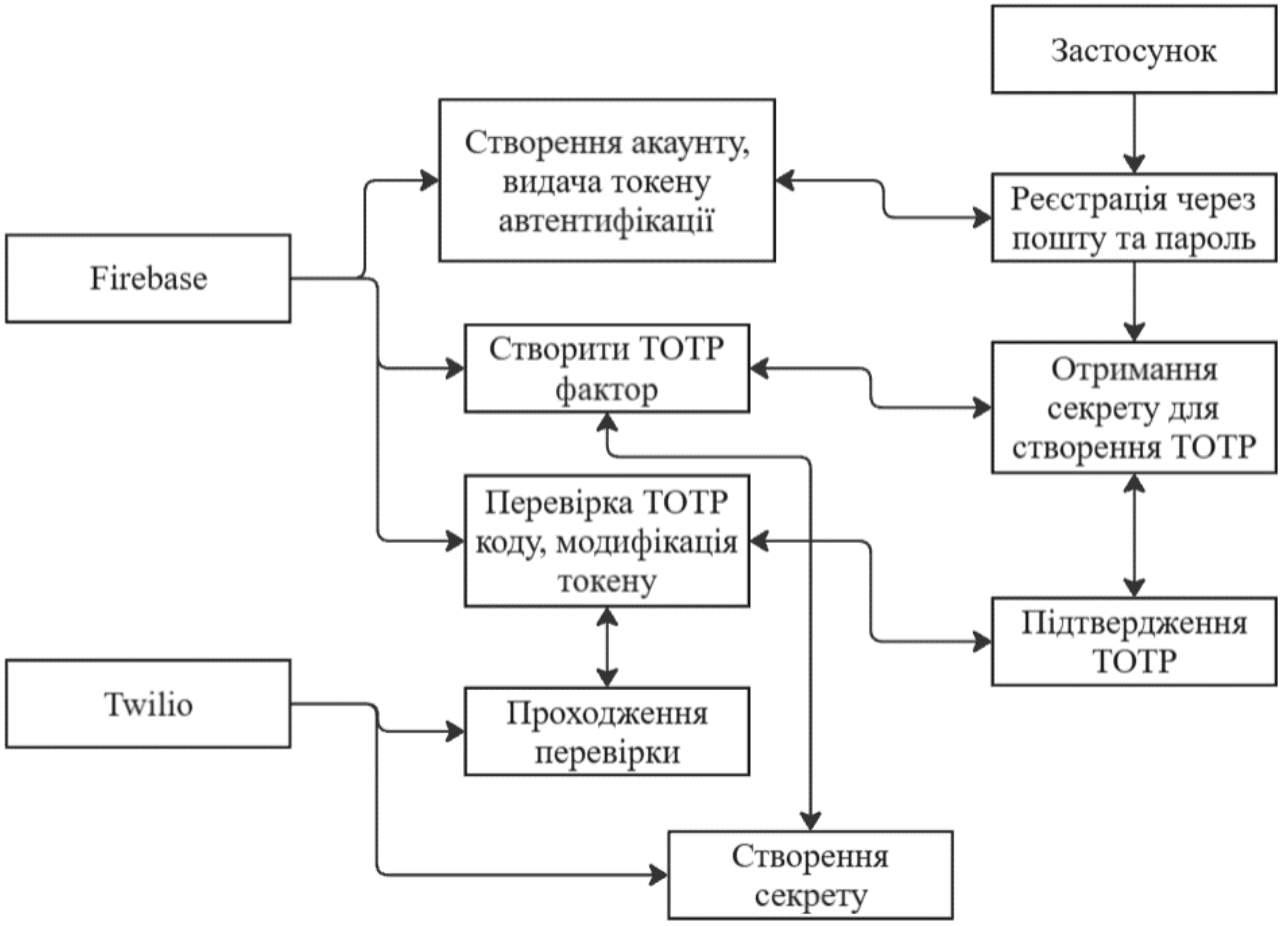
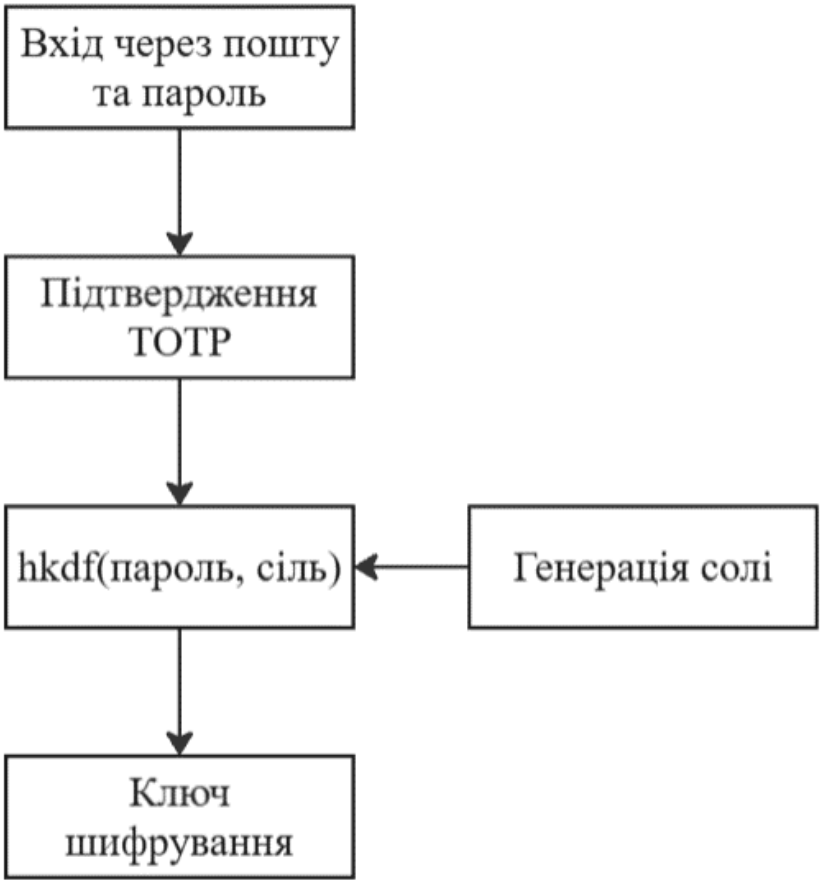
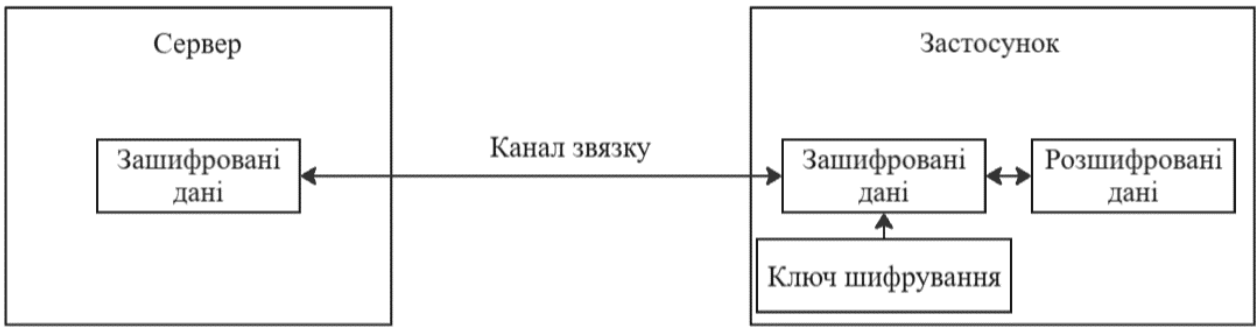


**СХЕМА ВХОДУ КОРИСТУВАЧА**

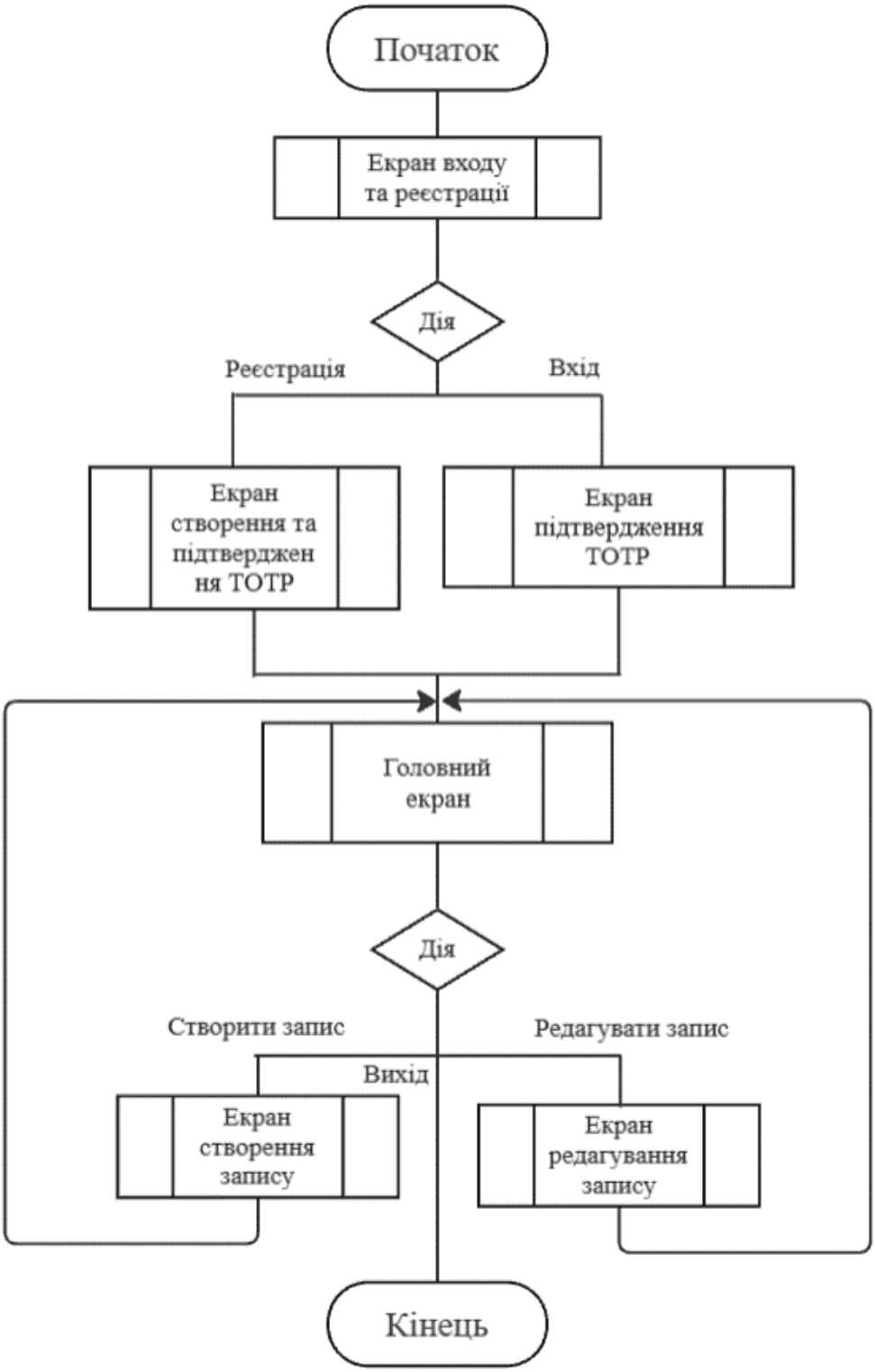
**СХЕМА ГЕНЕРАЦІЇ КЛЮЧА В АРХІТЕКТУРІ ДОДАТКУ**



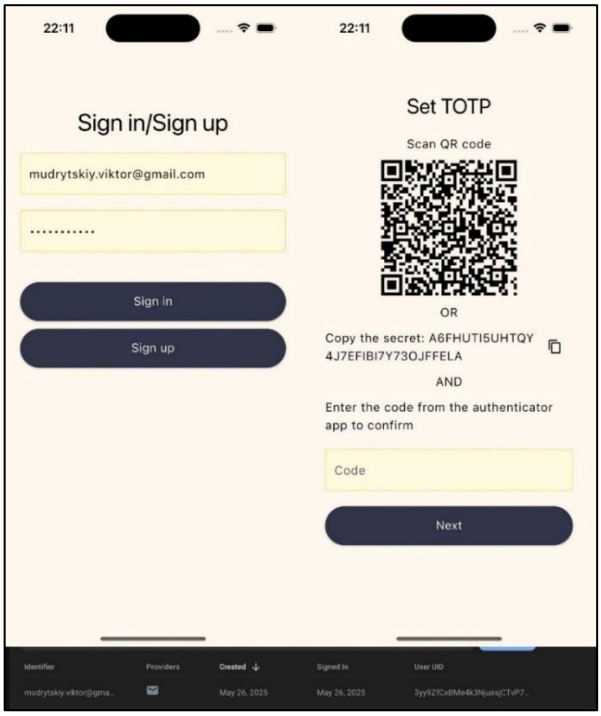
# СХЕМА ПЕРЕДАЧІ КОНФІДЕНЦІЙНИХ ДАНИХ



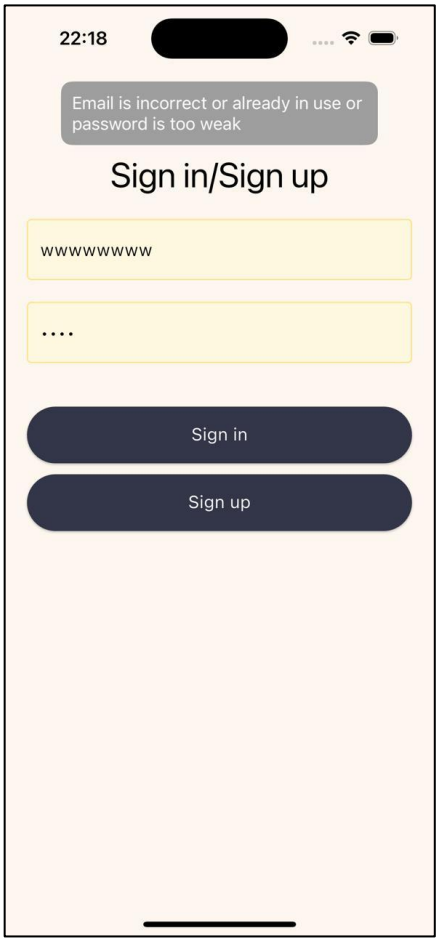
# СХЕМА РОБОТИ ЗАСТОСУНКУ



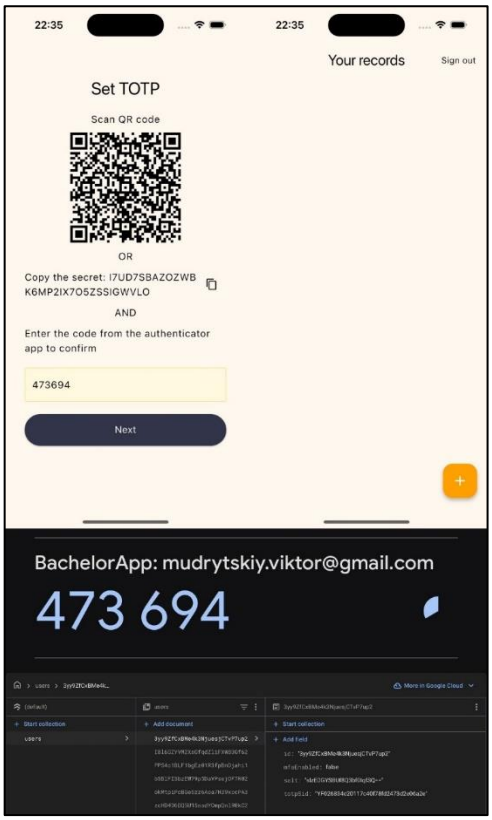
# РЕЗУЛЬТАТИ ТЕСТУВАННЯ ЗАСТОСУНКУ



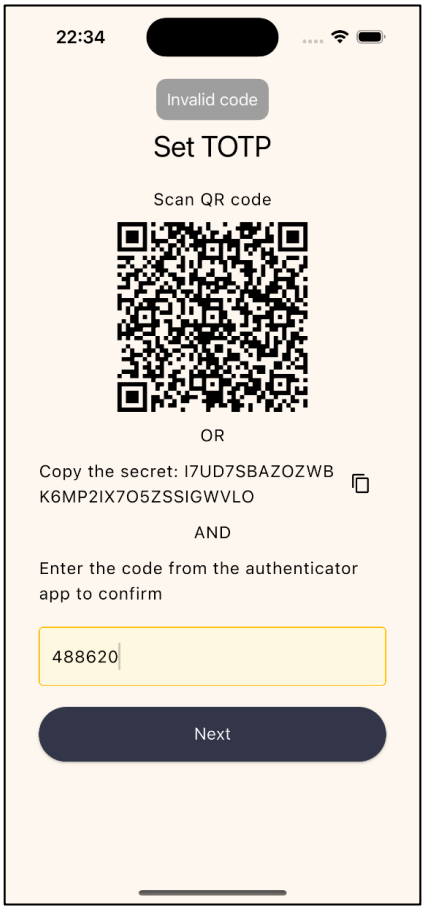

Результат реєстрації користувача



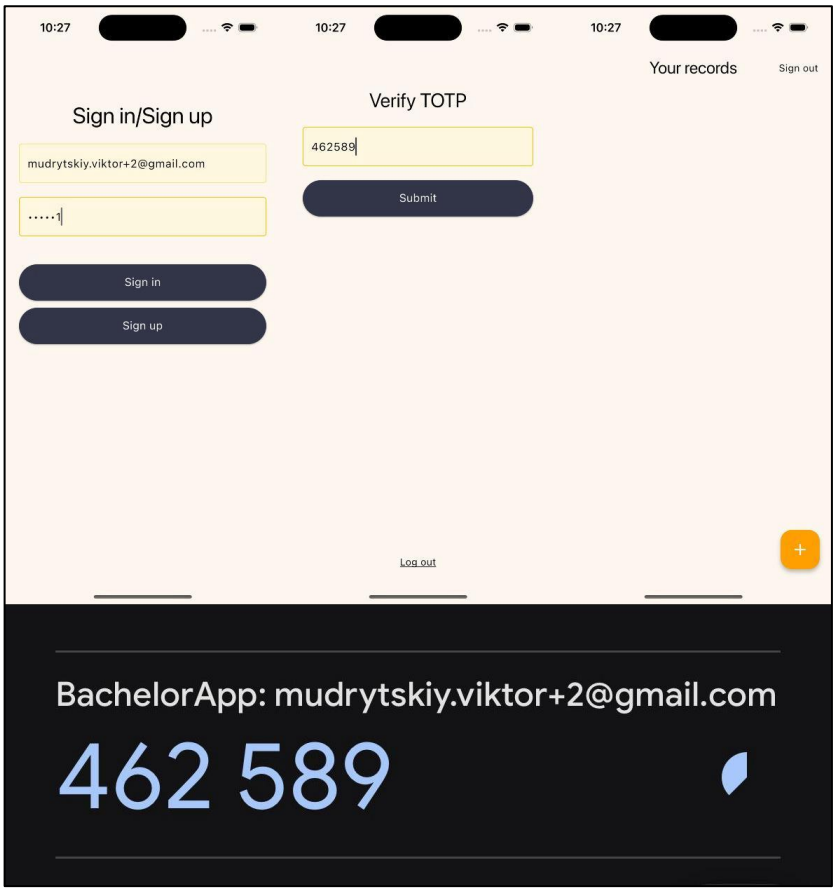

Помилка введених даних під час реєстрації



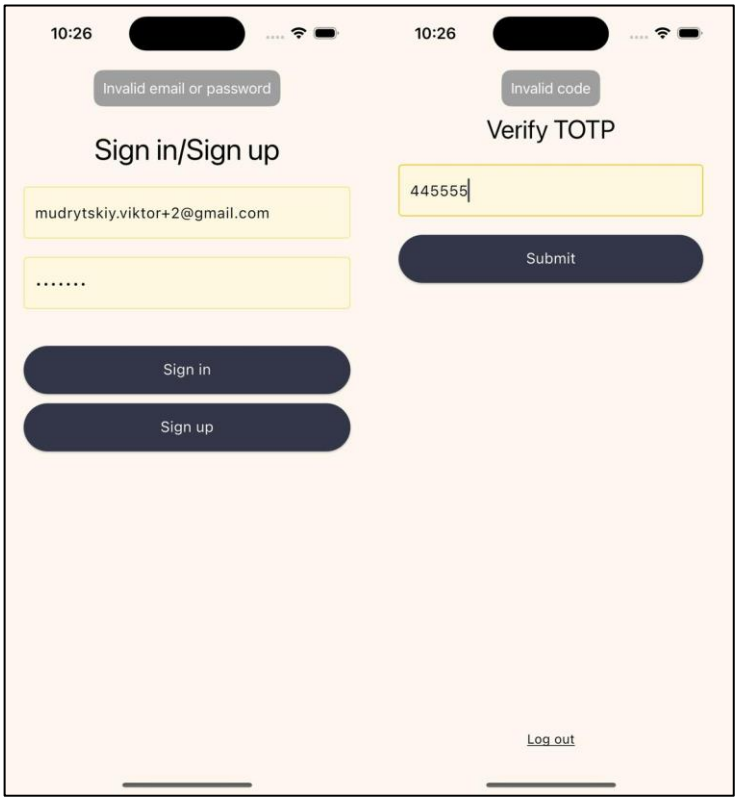

Результат успішного тестування генерації та підтвердження TOTP



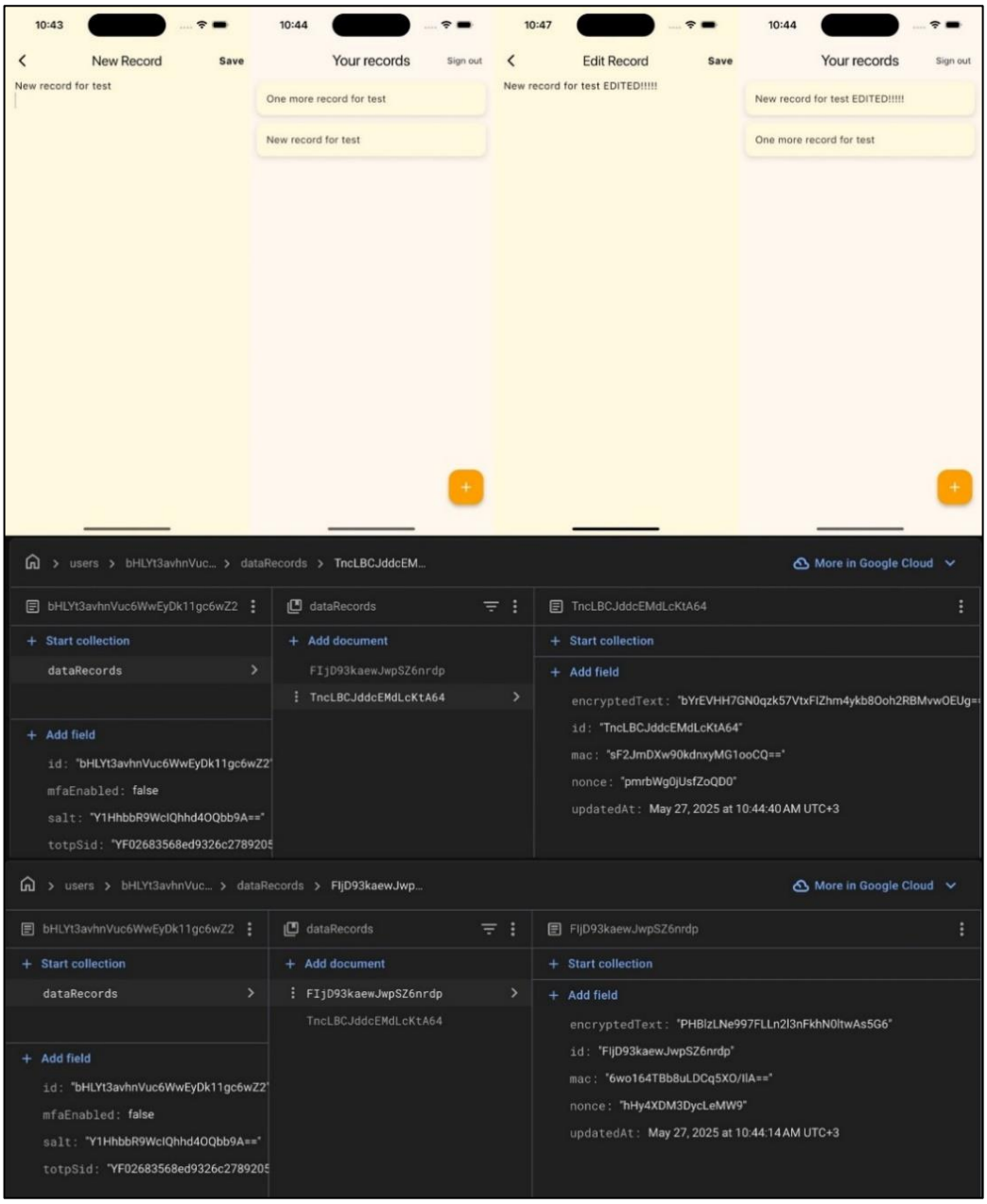

Помилка при вводі коду



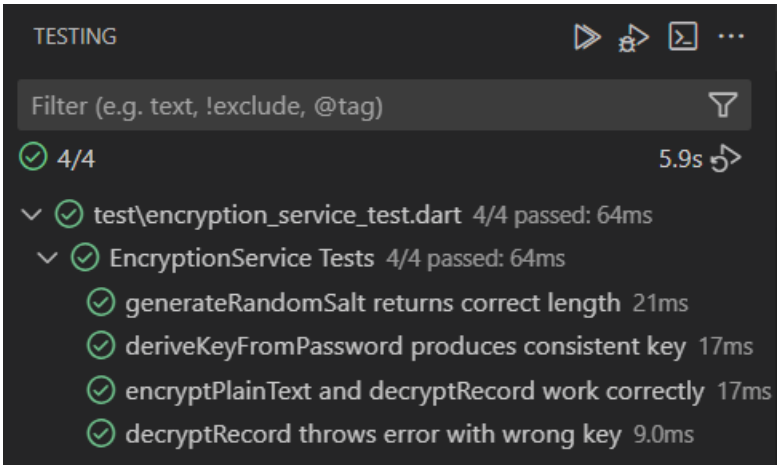

Результати тестування входу



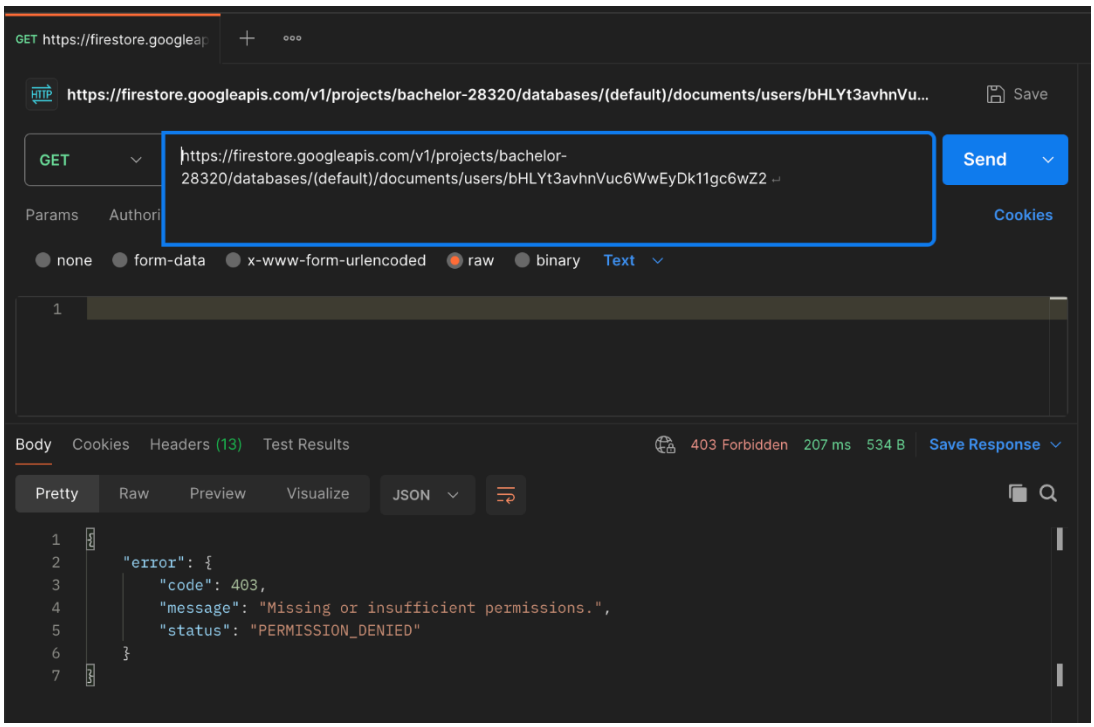

Помилки під час входу



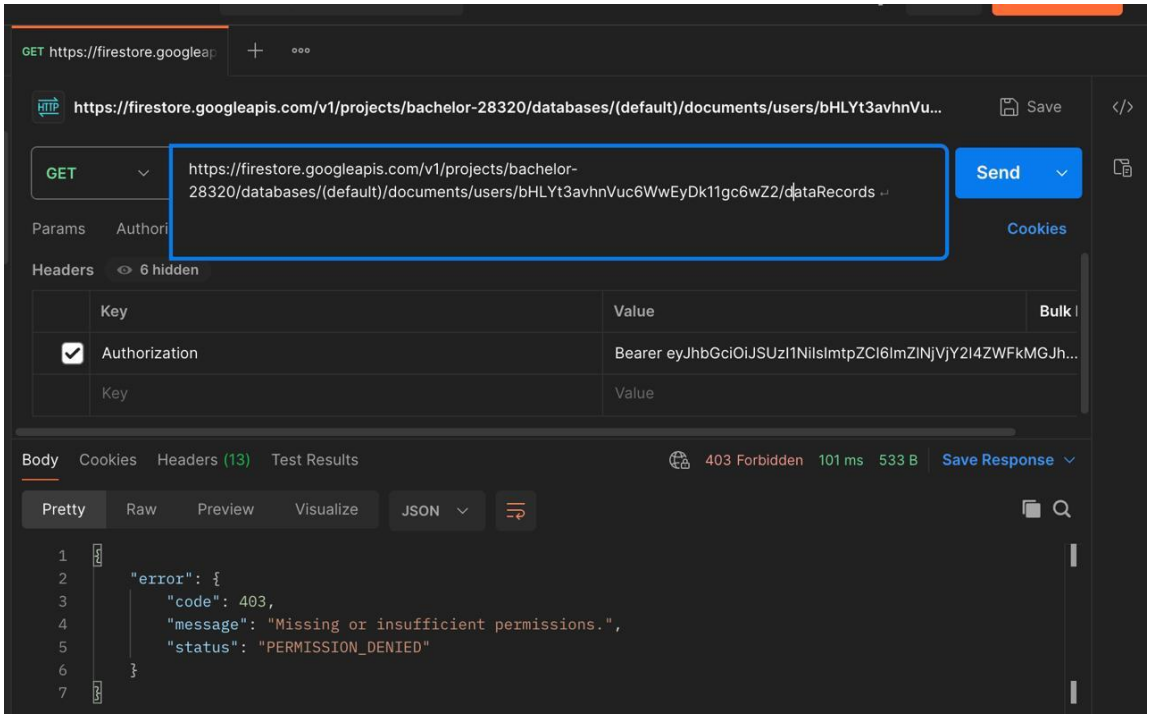

Результати тестування створення та редагування записів



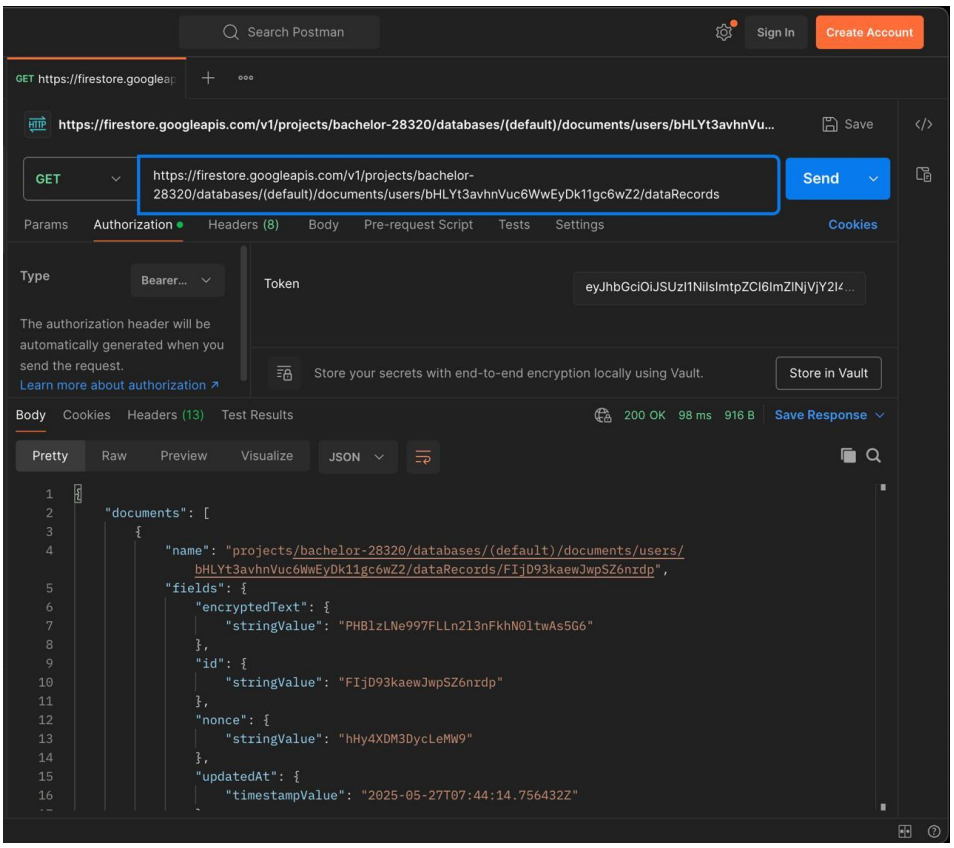

Результати модульного тестування



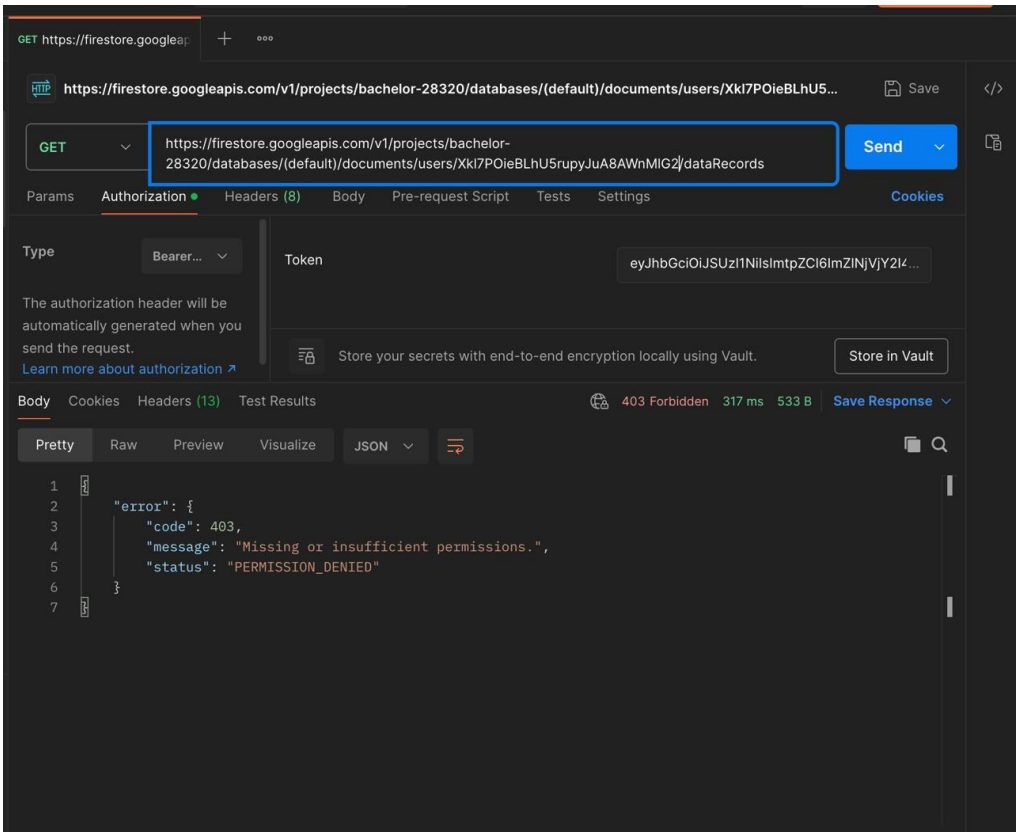

Спроба отримати дані без токену автентифікації




Спроба отримати конфіденційні дані, не пройшовши другий фактор автентифікації




Успішне отримання даних із токеном, який містить запис, що підтверджує проходження другого фактору автентифікації




Спроба отримання даних іншого користувача