

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Мобільний додаток з аутентифікацією за допомогою технології
розпізнавання обличчя»**

Виконав студент групи ЗППЗм-24-2
КІСЕЛЕВИЧ Антон Олександрович

Керівник роботи:
ВАКАЛЮК Тетяна Анатоліївна

Рецензент:
ЄФРЕМОВ Юрій Миколайович

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«18» вересня 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу

Здобувач вищої освіти: **КІСЕЛЕВИЧ Антон Олександрович**

Керівник роботи: **ВАКАЛЮК Тетяна Анатоліївна** Тема роботи: **«Мобільний додаток з аутентифікацією за допомогою технології розпізнавання обличчя»**, затверджена Наказом закладу вищої освіти від **«18» вересня 2025 р. № 492/с.**

Вихідні дані для роботи: Об'єктом дослідження виступають мобільні застосунки для організації та збереження інформації. Предметом дослідження є методи та засоби забезпечення безпечного доступу до мобільних нотаток шляхом впровадження технологій біометричної аутентифікації.

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Вакалюк Т.А	18.09.2025р.	18.09.2025
2	Вакалюк Т.А	18.09.2025	18.09.2025
3	Вакалюк Т.А	18.09.2025	18.09.2025

РЕФЕРАТ

Звіт з наукової практики магістра спеціальності 121 «Інженерія програмного забезпечення» присвячено розробці мобільного застосунку для створення та зберігання нотаток із використанням біометричної аутентифікації на основі розпізнавання обличчя. Робота складається з прототипу мобільного застосунку та пояснювальної записки.

Пояснювальна записка містить 78 сторінок, 5 ілюстрацій, 4 таблиці та список із 43 використаних джерел.

У першому розділі здійснено аналіз аналогів програмних рішень (Google Keep, Evernote, Microsoft OneNote, Apple Notes, Notion), визначено їхні функціональні можливості та обмеження щодо захисту даних. Особливу увагу приділено методам та алгоритмам біометричної аутентифікації (Eigenfaces, Fisherfaces, LBPH, FaceNet, ArcFace, MobileFaceNets).

У другому розділі подано варіанти використання застосунку, зокрема: реєстрація користувача, автентифікація за обличчям, резервний вхід, створення, редагування та пошук нотаток, синхронізація між пристроями та управління сесіями. Розглянуто архітектуру системи, побудовану на основі Java Spring Boot та Swift/SwiftUI, а також реалізовано базу даних (H2/SQL) із використанням JPA/Hibernate.

У практичній частині наведено приклади реалізації сутностей (User, Face, Role), інтеграцію моделі FaceNet для формування біометричних ембеддингів, використання OpenCV для попередньої обробки зображень, а також механізми безпеки на базі Spring Security і JWT. Додатково описано процес контейнеризації системи за допомогою Docker та Docker Compose.

У висновках узагальнено результати практики: створено прототип мобільного застосунку для безпечного зберігання нотаток із використанням біометричної аутентифікації, що поєднує високий рівень захисту даних та зручність користування. Отримані результати можуть бути використані як основа

для подальшого вдосконалення мобільних систем із підвищеними вимогами до безпеки.

КЛЮЧОВІ СЛОВА: МОБІЛЬНИЙ ЗАСТОСУНОК, НОТАТКИ, БІОМЕТРІЯ, АУТЕНТИФІКАЦІЯ, FACE RECOGNITION, SWIFT, SPRING BOOT, OPENCV, TENSORFLOW.

ABSTRACT

The report on the scientific practice of a Master's student in Software Engineering is devoted to the development of a mobile application for creating and storing notes with biometric authentication based on face recognition. The work consists of a mobile application prototype and an explanatory note.

The explanatory note contains 78 pages, 5 figures, 4 tables , and 43 references.

The first chapter analyzes existing note-taking applications (Google Keep, Evernote, Microsoft OneNote, Apple Notes, Notion), identifying their functionality and limitations regarding data protection. Particular attention is paid to biometric authentication methods and algorithms (Eigenfaces, Fisherfaces, LBPH, FaceNet, ArcFace, MobileFaceNets).

The second chapter presents the main use cases of the application, such as user registration, face authentication, backup login, note creation and editing, search, synchronization across devices, and session management. The system architecture was designed using Java Spring Boot for the backend and Swift/SwiftUI for the frontend, with a database implemented in H2/SQL using JPA/Hibernate.

The practical part demonstrates the implementation of entities (User, Face, Role), integration of the FaceNet model for generating biometric embeddings, OpenCV for image preprocessing, and security mechanisms based on Spring Security and JWT. Additionally, the process of containerization using Docker and Docker Compose is described.

The conclusions summarize the results of the practice: a prototype of a mobile application for secure note storage with biometric authentication was developed, combining high-level data protection with user convenience. The results may serve as a foundation for further development of mobile systems with enhanced security requirements.

KEYWORDS: MOBILE APPLICATION, NOTES, BIOMETRICS, AUTHENTICATION, FACE RECOGNITION, SWIFT, SPRING BOOT, OPENCV, TENSORFLOW.

ЗМІСТ

ВСТУП.....	8
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ ОБЛИЧЧЯ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ	10
1.1 Аналіз аналогів програмного продукту	10
1.2 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	13
1.3 Огляд наявних методів, моделей, алгоритмів розпізнавання обличчя.....	18
1.4 Архітектура застосунку	21
Висновки до першого розділу	24
РОЗДІЛ 2. РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ ОБЛИЧЧЯ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ	25
2.1 Визначення варіантів використання застосунку	25
2.2 Практична реалізація бекенду.....	32
2.3 Практична реалізація мобільного застосунку	47
2.4 База даних застосунку.....	52
Висновки до другого розділу	56
РОЗДІЛ 3. ТЕСТУВАННЯ ЗАСТОСУНКУ ТА РОЗГОРТАННЯ	58
3.1 Опис тест кейсів	58
3.2 Розгортання бекенд частини застосунку	64
3.3 Розгортання фронтенд частини застосунку.....	66
3.4 Робота з застосунком	69
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	78

ДОДАТКИ.....	83
--------------	----

ВСТУП

Сучасний розвиток мобільних технологій збільшує потребу в ефективних та безпечних засобах збереження й організації особистої інформації. Мобільні нотатки стали невід'ємною частиною і повсякденного життя користувачів, проте традиційні методи захисту даних, такі як паролі чи PIN-коди, нерідко виявляються вразливими до несанкціонованого доступу. У цьому контексті актуальним постає впровадження біометричних технологій, зокрема розпізнавання обличчя, що поєднує високий рівень надійності та зручності у використанні. Саме інтеграція біометричної аутентифікації в мобільні застосунки для нотаток визначає сучасні тенденції в напрямі інформаційної безпеки й користувацького досвіду.

Метою дослідження є розробка та обґрунтування мобільного застосунку для створення й зберігання нотаток із використанням технології аутентифікації за допомогою розпізнавання обличчя як основного засобу захисту даних.

Завданням даної роботи є аналіз сучасних мобільних застосунків для нотаток з метою визначення їх функціональних можливостей і обмежень у сфері захисту даних; дослідження методів та алгоритмів біометричної аутентифікації і оцінка їх придатності для інтеграції у мобільні системи; обґрунтування вибору інструментальних засобів для розробки серверної й клієнтської частин; побудова архітектури програмного забезпечення з підтримкою біометричного захисту; реалізація основних модулів бекенду та клієнтського застосунку; тестування функціональних компонентів і перевірка коректності роботи механізмів аутентифікації; виконання розгортання системи та оцінка її можливостей подальшого розвитку.

Об'єктом дослідження виступають мобільні застосунки для організації та збереження інформації. Предметом дослідження є методи та засоби забезпечення безпечного доступу до мобільних нотаток шляхом впровадження технологій біометричної аутентифікації.

Наукова новизна роботи полягає у розробленні підходу до інтеграції алгоритмів розпізнавання обличчя в архітектуру мобільного застосунку для нотаток з урахуванням специфіки взаємодії користувача з персональним мобільним пристроєм. У роботі пропонується поєднання високого рівня захисту даних із збереженням швидкодії та зручності користування.

Практичне значення дослідження полягає у створенні прототипу мобільного застосунку, що може бути використаний для підвищення безпеки персональної інформації користувачів. Результати роботи мають перспективу застосування в освітньому процесі при підготовці фахівців з інформаційних технологій, а також у подальших дослідженнях, спрямованих на вдосконалення засобів біометричної аутентифікації в мобільних системах.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ ОБЛИЧЧЯ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ

1.1 Аналіз аналогів програмного продукту

У сучасному світі мобільні застосунки для створення та зберігання нотаток займають провідне місце серед інструментів організації особистої та робочої інформації. Серед найбільш відомих аналогів можна відзначити Google Keep, Evernote, Microsoft OneNote, Apple Notes та низку менш поширених застосунків, які орієнтовані на вузькі категорії користувачів [1]. Вони забезпечують швидке створення, редагування й пошук інформації, синхронізацію між пристроями та інтеграцію з хмарними сервісами [2].

Google Keep відзначається мінімалістичним інтерфейсом і можливістю швидкого створення нагадувань, списків та голосових нотаток. Проте його функціонал обмежений, а рівень захисту даних здебільшого базується на загальній аутентифікації Google-акаунта, без застосування спеціалізованих біометричних механізмів.

Evernote орієнтований на професійне використання та має розширений функціонал для структурування великої кількості матеріалів. Він підтримує прикріплення файлів, роботу з таблицями, тегами та спільний доступ. Однак, незважаючи на багатий функціонал, у ньому не передбачено вбудованої аутентифікації за допомогою розпізнавання обличчя, а доступ до нотаток обмежується стандартними методами захисту пристрою.

Microsoft OneNote є частиною екосистеми Microsoft 365 та орієнтований на інтеграцію з іншими сервісами компанії. Його перевагою є підтримка рукописного вводу, можливість роботи з мультимедійним контентом та синхронізація у хмарному сховищі. Безпека даних забезпечується здебільшого корпоративними політиками доступу, однак спеціалізованої біометричної перевірки також не передбачено [3].

Apple Notes як стандартний застосунок операційної системи iOS має тісну інтеграцію з екосистемою Apple [4]. Він підтримує захист паролем, а також доступ до нотаток із використанням Touch ID чи Face ID, проте ця функціональність обмежена виключно пристроями Apple і не переноситься на інші платформи.

Окрім зазначених рішень, на ринку існують менш поширені застосунки, такі як Zoho Notebook чи Notion, які поєднують функціонал для нотаток із можливостями управління проектами [5]. Вони дозволяють створювати спільні робочі простори, проте знову ж таки покладаються на стандартні механізми захисту, без цілеспрямованої реалізації біометричних алгоритмів.

Таким чином, проведений аналіз показує, що жоден із розглянутих аналогів не поєднує функціонал зручного зберігання та організації нотаток із широким використанням технологій розпізнавання обличчя як основного механізму аутентифікації. Це підтверджує доцільність та актуальність створення мобільного застосунку, у якому забезпечується інтеграція біометричного захисту з метою підвищення рівня інформаційної безпеки та покращення користувацького досвіду.

Порівняльний аналіз функціональних можливостей наявних рішень наведено в таблицях 1.1 – 1.3

Таблиця 1.1

Аналіз функціональних можливостей наявних рішень

Застосунок	Платформи	Основний функціонал
Google Keep	Android, iOS, Web	Текстові нотатки, списки, нагадування, голосові нотатки
Evernote	Android, iOS, Web, Desktop	Розширені нотатки, теги, вкладення, пошук
Microsoft OneNote	Android, iOS, Web, Desktop	Нотатки, рукописний ввід, мультимедіа, блокноти
Apple Notes	iOS, macOS	Нотатки, сканування документів, мультимедіа
Zoho Notebook	Android, iOS, Web	Нотатки-картки, впорядкування інформації
Notion	Android, iOS, Web, Desktop	Нотатки, бази даних, управління проектами

Таблиця 1.2

Аналіз функціональних можливостей наявних рішень

Застосунок	Хмарна синхронізація	Спільна робота
Google Keep	Так	Обмежена
Evernote	Так	Так
Microsoft OneNote	Так	Так
Apple Notes	Так (iCloud)	Обмежена
Zoho Notebook	Так	Обмежена
Notion	Так	Так

Таблиця 1.3

Аналіз функціональних можливостей наявних рішень

Застосунок	Захист паролем	Біометрія	Кросплатформна біометрія	Цільова аудиторія
Google Keep	Обліковий запис Google	Ні	Ні	Широке коло користувачів
Evernote	Так	Частково	Ні	Професійні користувачі
Microsoft OneNote	Корпоративні політики доступу	Ні	Ні	Освітні та корпоративні користувачі
Apple Notes	Так	Так	Ні	Користувачі екосистеми Apple
Zoho Notebook	Так	Ні	Ні	Малі команди
Notion	Так	Ні	Ні	Командна та проектна робота

Як видно з таблиць 1.1–1.3, більшість сучасних застосунків для зберігання нотаток забезпечують базовий функціонал організації інформації та хмарної синхронізації, проте не реалізують біометричну аутентифікацію як окремий, цілеспрямований механізм захисту користувацьких даних, зокрема з використанням технологій розпізнавання обличчя. Такий підхід залишає певний простір для підвищення рівня безпеки, адже сучасні користувачі дедалі більше очікують комплексного захисту своїх цифрових матеріалів. Крім того, інтеграція біометричних технологій могла б суттєво розширити можливості застосунків та забезпечити більш високий рівень довіри з боку аудиторії.

1.2 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Розробка сучасних інформаційних систем вимагає не лише визначення архітектурних підходів, але й ретельного обґрунтування вибору інструментальних засобів. Від мови програмування, фреймворків, бібліотек та систем управління залежностями значною мірою залежить продуктивність застосунку, швидкість його розробки, зручність супроводу та перспективи масштабування . У даному проєкті вибір інструментів здійснювався з урахуванням вимог до кросплатформенності, сумісності з бібліотеками для обробки зображень та нейронних мереж, підтримки сучасних парадигм розробки, а також наявності активної спільноти й якісної документації .

Основною мовою програмування для серверної частини обрано Java 17 . Цей вибір зумовлений низкою чинників. По-перше, Java є кросплатформенною завдяки віртуальній машині JVM, що дозволяє розгортати застосунок на будь-якій операційній системі без зміни коду . По-друге, екосистема Java має зрілу підтримку бібліотек комп'ютерного зору та машинного навчання, включно з Java-біндингами до OpenCV та TensorFlow. По-третє, Java 17 як LTS-версія отримує тривалу підтримку та регулярні оновлення, що важливо для стабільності та безпеки промислових рішень [6]. Додатковою перевагою є можливості багатопотокової обробки, що забезпечує ефективну роботу алгоритмів у паралельному режимі, а також широкий набір інструментів для оптимізації продуктивності JVM. Серед альтернатив варто відзначити Python, який є домінуючою мовою у сфері Data Science та AI, однак поступається Java в продуктивності під час CPU-інтенсивних обчислень; C++, що забезпечує максимальну швидкість, але ускладнює розробку та супровід; а також C#, який є потужним, проте тісно прив'язаний до екосистеми Windows і має обмежену кросплатформенність у порівнянні з Java.

Для організації серверної логіки застосунку обрано Spring Boot 3.2.5, що є одним із найпоширеніших Java-фреймворків [7]. Його головна перевага полягає в автоконфігурації, яка суттєво скорочує час на початкове налаштування проєкту.

Spring Boot пропонує готові рішення для реалізації REST API, інтеграції з базами даних, побудови безпечної архітектури, а також забезпечує сумісність із контейнерними середовищами на кшталт Docker. У поєднанні з концепціями IoC та DI фреймворк сприяє побудові масштабованих і легко підтримуваних застосунків.

Для роботи з даними використовується H2 Database, що є вбудованою in-memory базою. Основна причина її використання — швидкість та простота розгортання, що є важливим під час розробки й тестування. Вона підтримує SQL-стандарт, інтегрується з JPA/Hibernate та легко керується через Liquibase, який відповідає за версіонування та контроль міграцій [8]. Хоча для промислових систем частіше застосовують PostgreSQL або MySQL, H2 є доцільним вибором для прототипування та розробки мобільних рішень завдяки своїй легкості та мінімальним вимогам до налаштування.

У частині обробки зображень і генерації ембеддингів застосовується комбінація OpenCV 4.5.1-2 та TensorFlow 1.15.0. OpenCV використовується для виявлення облич на зображеннях, що завантажуються користувачем [9]. Воно забезпечує широкий набір функцій для попередньої обробки, таких як зміна розміру, фільтрація та конвертація форматів, а також включає LBP-каскади, які забезпечують швидку й ефективну детекцію. TensorFlow, у свою чергу, відповідає за роботу з попередньо натренованою моделлю FaceNet, яка генерує 512-вимірні вектори-ембеддинги. Використання цієї моделі дозволяє досягати точності розпізнавання близької до 99 %, що робить її придатною для завдань автентифікації.

Забезпечення безпеки даних реалізовано на основі Spring Security у поєднанні з JWT-токенами та алгоритмом хешування паролів BCrypt [21] [31] [34]. Така комбінація дозволяє побудувати stateless-систему авторизації, яка зручна для розподілених і масштабованих застосунків [12] [15] [18]. JWT використовується для передачі інформації про користувача у зашифрованому вигляді, а BCrypt гарантує надійний захист паролів від атак на основі словників чи rainbow tables [35].

Серед інструментів розгортання та деплою обрано Docker, який забезпечує ізоляцію середовищ, повторюваність результатів та можливість масштабування завдяки контейнеризації [13] [16]. Multi-stage збірки дозволяють мінімізувати розмір фінального образу, включивши лише необхідні залежності. Це особливо важливо для роботи на ARM64-пристроях і мобільних системах, де ресурси обмежені.

Оскільки застосунок орієнтований на мобільні пристрої, особливу увагу приділено вибору технологій для клієнтської частини [20] [22] [23] [26]. Для цього використано Swift як основну мову програмування та SwiftUI як сучасний фреймворк для побудови інтерфейсу користувача. Swift забезпечує високу продуктивність завдяки компіляції у машинний код, водночас залишаючись безпечною та зручною мовою завдяки системі опціоналів і сучасному синтаксису. SwiftUI, як декларативний UI-фреймворк, дозволяє швидко створювати адаптивні інтерфейси, що автоматично підлаштовуються під різні розміри екрану та підтримують темний і світлий режими. Інтеграція з фреймворком Combine забезпечує реактивну взаємодію між даними та інтерфейсом, що критично важливо для мобільних додатків, які працюють з мережею та асинхронними запитами .

Узагальнюючи, можна стверджувати, що комбінація Java, Spring Boot, OpenCV, TensorFlow на рівні серверної частини та Swift та SwiftUI на рівні клієнтської забезпечує баланс між продуктивністю, масштабованістю, безпекою та сучасним користувацьким досвідом . Це рішення відповідає вимогам дипломної роботи й водночас демонструє використання передових технологій, які широко застосовуються в сучасній індустрії розробки програмного забезпечення.

Архітектура мобільного застосунку для розпізнавання обличчя за допомогою штучного інтелекту

Загальна архітектура програмного забезпечення побудована на принципах шарової моделі, що передбачає поділ функціональності на окремі рівні з чітким розмежуванням відповідальності. На верхньому рівні розташовано Presentation Layer, який реалізує механізми взаємодії користувача із системою через REST-

контролери, обробку винятків та налаштування політик CORS. Наступний рівень це Business Layer, що містить основну логіку системи, зокрема сервіси аутентифікації, обробки облич та управління користувачами. Data Access Layer забезпечує взаємодію з базою даних через репозиторії, які реалізують шаблон Repository Pattern і надають абстрактний доступ до сутностей користувачів, ролей та ембеддингів. На нижньому рівні розташована база даних H2 у поєднанні з механізмом керування міграціями Liquibase, що гарантує узгодженість і відтворюваність структури даних.

Архітектура Spring Boot побудована відповідно до принципів інверсії керування та залежностей. Використання Dependency Injection дозволяє забезпечити слабе зв'язування між компонентами завдяки впровадженню залежностей через конструктори, що відповідає принципам єдиної відповідальності та сегрегації інтерфейсів. Управління конфігурацією здійснюється за допомогою класів, позначених анотаціями @Configuration і @EnableMethodSecurity. Окремі компоненти визначаються як біни за допомогою анотації @Bean, а значення властивостей передаються через механізм @Value. Такий підхід дозволяє централізовано налаштовувати систему та підвищує її гнучкість.

Безпекова архітектура реалізована з використанням Spring Security, де ключову роль відіграє фільтраційний ланцюг. Його структура включає низку послідовних фільтрів, серед яких обробники CORS, контексту безпеки, а також спеціалізований JwtAuthenticationFilter, що забезпечує перевірку та валідацію токенів. У межах цього процесу реалізується повний цикл аутентифікації: від отримання логін-запиту від клієнта, перевірки облікових даних у базі даних і формування JWT-токена до подальшої валідації токена під час кожного звернення до захищених ресурсів. Такий підхід забезпечує безстанну модель безпеки, що зручно інтегрується в розподілені системи.

Дані системи організовано у вигляді сутностей, пов'язаних між собою в межах моделі «користувач–роль–обличчя». Користувач може мати кілька ролей,

що відповідає відношенню «багато-до-багатьох», а також асоційований набір ембеддингів, сформованих на основі аналізу його обличчя, які зберігаються у сутності «Face». Реалізація патерна Repository забезпечує абстрагування рівня доступу до даних від бізнес-логіки, що спрощує тестування та полегшує масштабування системи.

Компонент штучного інтелекту й машинного навчання представлений обробним конвеєром для роботи з обличчями. На вхід надходить зображення у форматі MultipartFile, яке спочатку проходить через модуль OpenCV для детекції облич за допомогою LBP-каскаду. Виділена ділянка підлягає попередній обробці, зокрема нормалізації та зміни розміру, після чого передається до моделі FaceNet, яка формує 512-вимірний вектор-ембеддинг. Для оптимізації використовується стратегія завантаження моделей під час ініціалізації застосунку, що дозволяє одноразово завантажити ресурси OpenCV та TensorFlow й уникати повторних витрат при подальших викликах.

Інфраструктура системи розгортається за допомогою контейнеризації. Використано Docker multi-stage build, де перший етап відповідає за збирання застосунку з використанням Maven і формування JAR-файлу, а другий етап відповідає за створення мінімального можливого середовища виконання на базі JRE. У процесі розгортання здійснюється інсталяція залежностей OpenCV, копіювання моделей нейронних мереж, створення некореневого користувача та конфігурування перевірки стану контейнера. Оркестрація контейнерів може виконуватися за допомогою Docker Compose, де визначено параметри середовища, порти, томи та health-check для контролю працездатності сервісу.

Комунікація із зовнішніми системами реалізована у вигляді REST-архітектури. Основні ендпоінти охоплюють операції реєстрації та авторизації користувачів, зокрема реєстрацію за допомогою пароля або фотографії, а також отримання інформації про користувача. Усі запити проходять типовий цикл обробки: від валідації даних на рівні контролера, виконання бізнес-логіки у сервісному шарі та звернення до рівня доступу до даних до завершальної обробки

ембеддингів у модулі машинного навчання та формування відповіді клієнту у форматі HTTP-відповіді [36].

Загалом архітектура системи демонструє поєднання класичної багатошарової моделі та сучасних підходів до розробки безпечних і масштабованих застосунків. Її ключовими характеристиками є чітке розмежування відповідальності між рівнями, інтеграція Spring Boot із гнучкою системою конфігурацій, використання Spring Security з механізмом JWT, застосування патерна Repository для роботи з даними, побудова конвеєра обробки облич із використанням OpenCV та TensorFlow, а також контейнеризація з багатостадійною збіркою для підвищення ефективності розгортання.

1.3 Огляд наявних методів, моделей, алгоритмів розпізнавання обличчя

Аутентифікація за допомогою розпізнавання облич є одним із найдинамічніших напрямів розвитку біометричних технологій. Вона базується на використанні індивідуальних особливостей обличчя для ідентифікації особи та дозволяє поєднати високий рівень безпеки з простотою використання [10]. У практиці застосовуються як класичні алгоритми комп'ютерного зору, так і сучасні моделі глибокого навчання, що забезпечують точність на рівні, співставному з людським сприйняттям.

Одним із перших підходів були методи Eigenfaces, які використовують аналіз головних компонент для відображення зображень облич у простір з меншою розмірністю. Такий підхід забезпечував швидкість обчислень і простоту реалізації, проте мав високу чутливість до зміни освітлення та виразів обличчя [40]. Іншим класичним методом став підхід Fisherfaces, що ґрунтується на лінійному дискримінантному аналізі. Він показував кращу стійкість до варіацій освітлення, однак поступався сучасним глибоким методам за точністю .

Метод LBPH (Local Binary Patterns Histogram), який часто використовується в бібліотеці OpenCV, став перехідним рішенням між класичними та сучасними алгоритмами . Він аналізує локальні текстурні ознаки обличчя та формує

гістограму для опису зображення. Перевагою LBPН є швидкість і стійкість до локальних змін, але недоліком — низька точність на великих наборах даних.

Сучасні підходи базуються на згорткових нейронних мережах (CNN), які дозволяють витягати інформативні ознаки без ручного опису текстур [11] [17] [19]. Одним із найбільш відомих є FaceNet, що перетворює обличчя у 512-вимірний вектор та застосовує функцію втрат типу triplet loss для навчання на зіставлення схожих і відмінних зразків [38] [41] [43]. Такий підхід забезпечує високу точність та можливість масштабування на великі бази користувачів.

Подальшим розвитком стала модель ArcFace, у якій використано модифіковану функцію втрат із кутовим (angular) марджином. Це дозволило підвищити дискримінативність ознак і значно покращити результати на відкритих бенчмарках, таких як LFW чи MegaFace. ArcFace вважається одним із найточніших алгоритмів сучасності.

Для мобільних пристроїв було запропоновано полегшені архітектури, зокрема MobileFaceNets. Вони оптимізовані для роботи на пристроях із обмеженими ресурсами, зберігаючи при цьому достатній рівень точності. Завдяки використанню глибинних сепарабельних згорток ці моделі поєднують високу швидкість з малою кількістю параметрів.

Метод DeepFace, розроблений компанією Facebook, став одним із перших успішних застосувань глибинних нейронних мереж у завданні розпізнавання облич. Він використовував дев'ятишарову CNN і досяг точності понад 97 % на наборі LFW, що було проривом у 2014 році.

Ще одним цікавим підходом є Dlib ResNet Face Recognition, який реалізує архітектуру ResNet із функцією втрат triplet loss. Ця реалізація є відкритою, активно використовується у дослідницьких і комерційних проєктах і дозволяє отримувати вектори-ембеддинги з високим рівнем дискримінативності.

Метод SphereFace пропонує інший варіант модифікації функції втрат, вводячи мультиплікативний кутовий марджин. Це дало змогу значно покращити

результати класифікації у порівнянні з традиційними softmax-втратами та наблизило якість алгоритмів до рівня людського розпізнавання.

Таким чином, аналіз існуючих методів показує еволюцію від класичних статистичних підходів (Eigenfaces, Fisherfaces, LBPH) до сучасних алгоритмів глибинного навчання (FaceNet, ArcFace, MobileFaceNets, DeepFace, SphereFace, Dlib ResNet). Класичні методи зберігають значення для навчальних і невеликих систем, проте їх точність є недостатньою для масштабних проєктів. Натомість сучасні CNN-моделі забезпечують високу дискримінативність і надійність, хоча потребують більших обчислювальних ресурсів [25]. Для мобільних рішень найбільш перспективними виглядають полегшені архітектури на зразок MobileFaceNets, які поєднують баланс між точністю та продуктивністю, що особливо важливо для реалізації біометричної аутентифікації у мобільних додатках. В таблиці 1.4 наведено порівняльну таблицю методів.

Таблиця 1.4

Порівняльна таблиця методів розпізнавання обличчя

Метод	Тип підходу	Швидкість виконання	Точність	Стійкість до освітлення	Вимоги до ресурсів	Сфера застосування
Eigenfaces	Класичний (PCA)	Висока	Середня	Низька	Низькі	Навчальні системи, прості додатки
Fisherfaces	Класичний (LDA)	Висока	Середня	Середня	Низькі	Демонстраційні та невеликі проєкти
LBPH	Класичний (текстурний аналіз)	Висока	Середня	Середня	Низькі	Реальний час, прості системи контролю
DeepFace	CNN (9 шарів)	Середня	Висока	Висока	Високі	Соцмережі, масштабні сервіси
FaceNet	CNN + Triplet Loss	Середня	Висока	Висока	Високі	Системи з великою базою користувачів

ArcFace	CNN + Angular Loss	Середня	Дуже висока	Дуже висока	Високі	Біометричні системи високої точності
SphereFace	CNN + Multiplicative Angular Loss	Середня	Висока	Висока	Високі	Наукові та комерційні дослідження
Dlib ResNet	ResNet + Triplet Loss	Середня	Висока	Висока	Середні	Open-source проекти, прототипування
MobileFaceNets	Полегшена CNN	Висока	Висока	Висока	Низькі	Мобільні застосунки, IoT-пристрої

1.4 Архітектура застосунку

Архітектура бекенд частини застосунку базується на фреймворку Spring Boot 3.2.5 із використанням мови програмування Java 17. Вона реалізує багаторівневу структуру, що забезпечує логічне розділення відповідальностей між компонентами системи, підвищує її масштабованість, зручність тестування та спрощує подальше розширення функціональності. Серверна частина функціонує як RESTful-вебсервіс, який забезпечує обмін даними між клієнтським застосунком і сервером через HTTP-інтерфейс.

У структурі проекту виділено кілька взаємопов'язаних шарів. Контролери (AuthController, UserController) приймають HTTP-запити, виконують первинну валідацію даних і спрямовують їх у сервісний рівень. Сервісна логіка реалізована в класах AuthenticationServiceImpl, UserServiceImpl і FaceServiceImpl, які відповідають за авторизацію користувачів, реєстрацію, створення токенів доступу, обробку зображень облич і генерацію векторних ембедингів за допомогою нейронної моделі FaceNet. Репозиторії (UserRepository, FaceRepository, RoleRepository) забезпечують роботу з базою даних через JPA/Hibernate, надаючи уніфікований механізм збереження та вибірки об'єктів. Модельна частина системи описує сутності користувачів, ролей і біометричних шаблонів, визначаючи між ними відношення типу «один-до-багатьох» і «багато-до-багатьох».

Суттєвим елементом архітектури є модуль безпеки, який реалізує безстанну авторизацію на основі технології JWT. Кожен запит супроводжується токеном, що перевіряється фільтром `JwtAuthenticationFilter` і розшифровується компонентом `JwtTokenProvider`. Для захисту облікових даних використовується шифрування паролів алгоритмом `BCrypt`, а політики доступу визначаються у класі `WebSecurityConfig`. Окремі конфігураційні модулі відповідають за CORS-доступ, маршрутизацію та інтеграцію з бібліотекою `MapStruct` для перетворення об'єктів.

Інтелектуальний компонент системи базується на поєднанні бібліотек `OpenCV` та `TensorFlow`. Під час реєстрації користувача система здійснює детекцію обличчя, виділення його області та генерацію 512-вимірної ембедінгу, який зберігається в базі даних у вигляді двійкових даних, асоційованих із користувачем. У процесі входу створений ембедінг порівнюється з наявними записами за метрикою косинусної схожості, і при перевищенні порогового значення автентифікація вважається успішною.

Застосування багаторівневої архітектури забезпечує високу надійність і розширюваність системи, дозволяючи незалежно розвивати окремі компоненти, підтримувати чисту структуру коду та реалізовувати сучасні підходи до інтеграції алгоритмів штучного інтелекту в середовищі веб-сервісів.

Архітектура мобільного застосунку побудована з використанням сучасного фреймворку `SwiftUI` у поєднанні з архітектурним підходом `MVVM` (`Model–View–ViewModel`), що забезпечує розмежування логіки даних, бізнес-процесів і користувацького інтерфейсу [14]. Така структура дозволяє підтримувати реактивність інтерфейсу, мінімізує кількість імперативного коду та спрощує тестування окремих компонентів. Застосунок є клієнтською частиною комплексної системи біометричної автентифікації, що взаємодіє з сервером через `RESTful-API`.

У центрі архітектури розташована модель стану `AppViewModel`, позначена атрибутом `@MainActor`, яка виступає як основний координатор між інтерфейсом користувача та зовнішніми сервісами. Вона містить спостережувані властивості (`@Published`) — токен доступу, профіль користувача, список нотаток,

повідомлення про помилки та індикатори завантаження. Будь-які зміни цих властивостей автоматично відображаються у візуальних компонентах завдяки реактивній природі SwiftUI. Комунікація з сервером реалізується через об'єкт `APIClient`, який використовує асинхронну модель `async/await` для виконання мережевих запитів у неблокувальному режимі.

Інтерфейс користувача побудовано за принципом композиції незалежних представлень (views). Головним контейнером є `RootView`, який визначає логіку переходів між екранами авторизації (`AuthView`) та робочою областю (`NotesView`) залежно від стану користувача. Кожен екран складається з підкомпонентів — форм входу, реєстрації та біометричної аутентифікації, що відображають і змінюють дані через зв'язок із `AppViewModel`. Елемент `FaceLoginView` забезпечує роботу з камерою пристрою, створює фотографію користувача, а далі передає її у бекенд для обробки системою розпізнавання облич.

Збереження даних здійснюється на двох рівнях: короткострокове — у пам'яті програми через властивості `@State` і `@Published`, та довгострокове — у локальних сховищах. Клас `KeychainService` відповідає за безпечне зберігання токенів автентифікації, тоді як `NotesStore` зберігає створені користувачем нотатки у вигляді JSON-файлів у каталозі `Documents`. Така комбінація гарантує збереження конфіденційних даних між сесіями та дозволяє використовувати застосунок у режимі офлайн.

Загалом архітектура мобільного клієнта поєднує декларативну модель побудови інтерфейсу, реактивне оновлення стану та асинхронну взаємодію з сервером. Це забезпечує високу стабільність, безпеку та адаптивність системи, а також створює передумови для подальшої інтеграції додаткових сервісів — зокрема, модулів локального розпізнавання облич або аналітики поведінки користувачів.

Висновки до першого розділу

У першому розділі було здійснено комплексний аналіз сучасних мобільних застосунків для зберігання нотаток, виявлено їхні ключові функціональні особливості та обмеження, а також розглянуто наявні методи та алгоритми аутентифікації за допомогою розпізнавання облич. Виявлено, що популярні сервіси, такі як Google Keep, Evernote, Microsoft OneNote чи Apple Notes, пропонують широкий спектр можливостей для організації інформації, проте не забезпечують цілеспрямованої реалізації біометричного захисту, окрім часткової інтеграції стандартних механізмів у межах екосистеми Apple. Це засвідчує відсутність комплексного рішення, що поєднує зручність роботи з нотатками та високий рівень безпеки завдяки використанню розпізнавання облич як основного механізму доступу.

Обґрунтування вибору інструментальних засобів показало, що поєднання Java, Spring Boot, OpenCV та TensorFlow для серверної частини з Swift і SwiftUI для клієнтської є оптимальним варіантом, оскільки забезпечує баланс між продуктивністю, масштабованістю, безпекою та сучасним користувацьким досвідом. Продемонстровано, що обрана архітектура на основі шарової моделі, Spring Security, JWT-токенів і контейнеризації за допомогою Docker створює надійне та масштабоване середовище для впровадження біометричної аутентифікації.

Огляд алгоритмів аутентифікації за обличчям підтвердив еволюцію від класичних статистичних підходів до сучасних моделей глибинного навчання, що забезпечують значно вищу точність та стійкість до змін зовнішніх умов. Встановлено, що найбільш перспективними для мобільних застосунків є полегшені архітектури, зокрема MobileFaceNets, які поєднують високу швидкість роботи з достатнім рівнем точності та низькими вимогами до ресурсів. Таким чином, результати першого розділу підтверджують доцільність створення мобільного застосунку для нотаток з інтегрованою біометричною аутентифікацією, що задовольняє сучасні потреби в інформаційній безпеці та зручності користування.

РОЗДІЛ 2. РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ РОЗПІЗНАВАННЯ ОБЛИЧЧЯ ЗА ДОПОМОГОЮ ШТУЧНОГО ІНТЕЛЕКТУ

2.1 Визначення варіантів використання застосунку

Проектований мобільний застосунок «Мобільні нотатки з аутентифікацією за допомогою розпізнавання обличчя» орієнтований на створення безпечного середовища для збереження особистих даних користувача. Його використання передбачає низку варіантів сценаріїв взаємодії, кожен з яких пов'язаний із виконанням конкретних завдань.

Базовим варіантом використання є реєстрація користувача. У цьому сценарії новий користувач створює обліковий запис, вносячи персональні дані, такі як електронна адреса та пароль, після чого виконується прив'язка біометричного шаблону обличчя. Система здійснює детекцію обличчя за допомогою алгоритмів комп'ютерного зору, генерує ембеддинг та зберігає його в базі даних разом із обліковим записом користувача. Цей процес дозволяє надалі використовувати біометричні ознаки як головний механізм автентифікації.

Другим важливим сценарієм є аутентифікація користувача. Коли користувач відкриває застосунок, камера мобільного пристрою фіксує його обличчя, система генерує ембеддинг у реальному часі та порівнює його з даними, що зберігаються у базі. У разі збігу користувач отримує доступ до своїх нотаток. Такий підхід дозволяє відмовитися від введення паролів, забезпечуючи зручність і водночас високий рівень безпеки.

Ключовим функціональним сценарієм є створення нотаток. Після успішної аутентифікації користувач отримує можливість створювати текстові записи, які можуть містити службову чи особисту інформацію. Нотатки зберігаються локально та/або у хмарному сховищі, при цьому доступ до них можливий лише після біометричної перевірки.

Подальшим варіантом використання є редагування та видалення нотаток. Користувач має змогу змінювати зміст уже створених записів або повністю видаляти їх. Кожна така операція відбувається у межах сеансу, що підтверджений успішною аутентифікацією, що гарантує захист від несанкціонованого втручання [27].

Ще один сценарій пов'язаний із переглядом та пошуком нотаток, що є базовою, але водночас критично важливою функцією з позиції зручності повсякденної роботи користувача. Застосунок надає можливість структурувати збережені записи за категоріями, тегамі або іншими логічними ознаками, що спрощує навігацію в межах персонального сховища даних. Крім того, реалізований механізм пошуку за ключовими словами дозволяє оперативно знаходити необхідну інформацію навіть за умови накопичення значного обсягу нотаток, що підвищує ефективність використання системи та зменшує часові витрати користувача.

Важливим функціональним варіантом використання є авторизація доступу з різних пристроїв. Система підтримує механізм автентифікації на основі JWT-токенів, що забезпечує можливість одночасної роботи користувача з кількох пристроїв без втрати цілісності сесії. Кожен запит до серверної частини супроводжується перевіркою дійсності токена, що гарантує коректність ідентифікації користувача та мінімізує ризики несанкціонованого доступу. Такий підхід дозволяє поєднати гнучкість використання з дотриманням сучасних вимог інформаційної безпеки.

Окрему увагу слід приділити сценарію відновлення доступу, який має практичне значення у реальних умовах експлуатації застосунку. У разі виникнення проблем із біометричною автентифікацією, наприклад через технічні обмеження камери, зміну зовнішності користувача або збоїв в роботі сенсорів, система передбачає альтернативні механізми входу. До них належать резервна авторизація за допомогою пароля або підтвердження особи через електронну пошту. Це рішення забезпечує оптимальний баланс між високим рівнем захисту персональних даних та доступністю сервісу для користувача, підвищуючи загальну надійність і

зручність системи. На рисунку 2.1 наведено UML-діаграму варіантів використання, яка наочно відображає описані сценарії взаємодії користувача із застосунком.

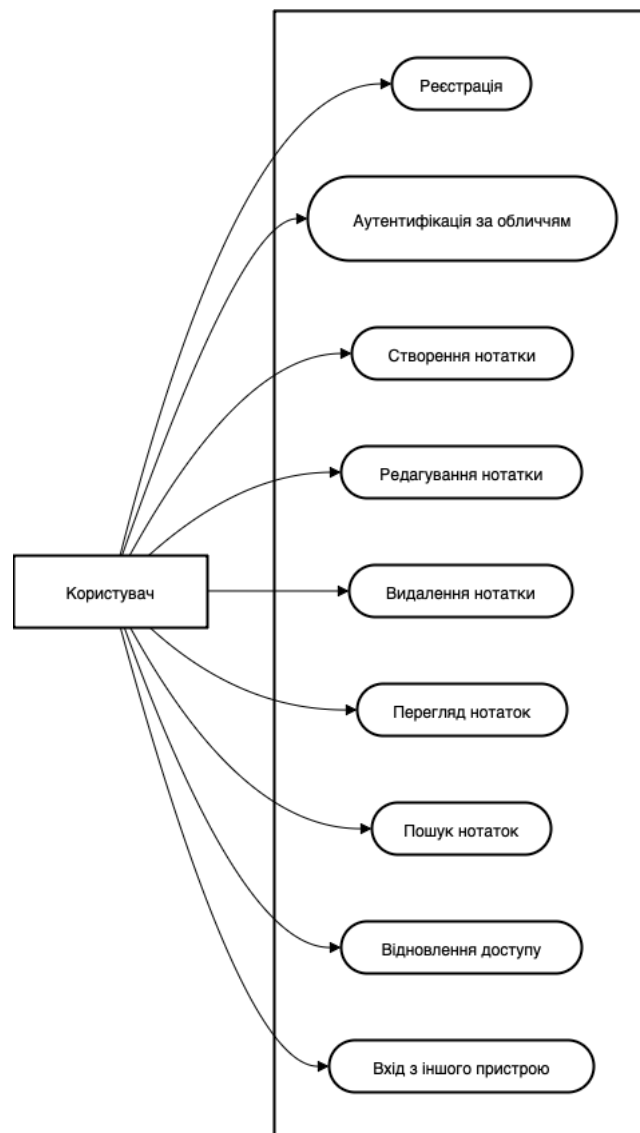


Рис 2.1 – Діаграма варіантів використання

Таким чином, основні варіанти використання застосунку охоплюють процеси реєстрації, автентифікації, створення та редагування нотаток, пошуку й перегляду збереженої інформації, а також відновлення доступу в разі потреби. Кожен сценарій реалізований із урахуванням вимог до інформаційної безпеки та користувацької зручності, що робить запропонований застосунок конкурентоспроможним серед аналогів. В таблиці 2.1 наведено короткий опис варіантів використання та залежності між ними

Варіанти використання застосунку

ID	Назва вимоги	Опис	Пріоритет	Критерії приймання	Примітки
FR-01	Реєстрація облікового запису	Користувач може створити обліковий запис за email і паролем з подальшою прив'язкою біометрії.	Критичний	Given новий користувач; When вводить валідні email/пароль; Then обліковий запис створено та відправлено підтвердження (опційно).	Валідація формату email та складності пароля.
FR-02	Біометрична реєстрація (ембеддинг)	Система приймає фото/відео обличчя, детектує обличчя (OpenCV) і створює ембеддинг (FaceNet).	Критичний	Given наявне якісне фото; When користувач надає доступ до камери; Then ембеддинг збережено та пов'язано з обліковим записом.	Перевірка наявності обличчя, контроль якості кадру.
FR-03	Аутентифікація за обличчям	Користувач входить до застосунку за допомогою камери, без введення пароля.	Критичний	Given зареєстрований користувач; When подібність > поріг; Then видати JWT та надати доступ.	Поріг подібності типово 0.6; налаштовується.

Продовження табл. 2.1

FR-04	Резервний вхід (пароль/OTP)	У разі невдачі біометрії користувач може увійти за паролем або одноразовим кодом на email.	Високий	Given невдала біометрія; When користувач обирає резерв; Then успішний вхід після перевірки.	FR-01	Рекомендовано 2FA.
FR-05	Створення нотатки	Користувач може створити нову нотатку з текстом і/або вкладеннями.	Високий	Given автентифікація; When користувач зберігає нотатку; Then запис доступний у списку.	UC-03	Підтримка офлайн-режиму (кеш).
FR-06	Редагування нотатки	Користувач оновлює вміст існуючої нотатки.	Середній	Given існуюча нотатка; When користувач вносить зміни; Then зміни синхронізовано на всіх пристроях.	FR-05	Журнал змін (опційно).
FR-07	Видалення нотатки	Користувач видаляє нотатку з можливістю відновлення протягом періоду утримання.	Середній	Given існуюча нотатка; When користувач підтвердив видалення; Then запис переміщено до «Смітника».	FR-05	Параметричний retention період.
FR-08	Перегляд та пошук нотаток	Користувач переглядає список, фільтрує та здійснює пошук за ключовими словами/тегами.	Високий	Given набір нотаток; When введено запит; Then повернено релевантні результати ≤ 500 мс локально.	FR-05	Пошук локальний/серверний.

Продовження табл. 2.1

FR-09	Вкладення медіа	Нотатка може містити фото, аудіо чи файли з обмеженням розміру.	Середній	Given вибране вкладення; When розмір $\leq$ ліміт; Then файл збережено та доступний.	FR-05	Сканування типів; прев'ю для зображень.
FR-10	Синхронізація між пристроями	Дані користувача синхронізуються при вході на інший пристрій.	Високий	Given новий пристрій; When виконано вхід; Then нотатки синхронізовано ≤ 5 с при мережі ≥ 10 Мбіт/с.	FR-03, FR-05	Конфлікти вирішуються стратегією злиття.
FR-11	Налаштування порогу подібності	Адміністратор або користувач (за політикою) може змінювати поріг similarity.	Середній	Given нове значення; When збережено; Then новий поріг застосовано до наступних входів.	FR-03	Валідація діапазону [0..1].
FR-12	Аудит дій користувача	Система журналює ключові події: входи, створення/видалення нотаток, зміну біометрії.	Середній	Given виконано дію; When перегляд аудиту; Then подія присутня з мітками часу та ідентифікатором сеансу.	FR-03, FR-05	Зберігання логів згідно з політикою retention.
FR-13	Обмеження спроб входу	Система блокує надмірні невдалі спроби аутентифікації.	Високий	Given N невдалих спроб; When ліміт перевищено; Then обліковий запис/клієнт тимчасово заблоковано.	FR-03, FR-04	Розумний back-off.

Закінчення табл. 2.1

FR-14	Аудит дій користувача	Система журналює ключові події: входи, створення/видалення нотаток, зміну біометрії.	Середній	Given виконано дію; When перегляд аудиту; Then подія присутня з мітками часу та ідентифікатором сеансу.	FR-03, FR-05	Зберігання логів згідно з політикою retention.
FR-15	Обмеження спроб входу	Система блокує надмірні невдалі спроби аутентифікації.	Високий	Given N невдалих спроб; When ліміт перевищено; Then обліковий запис/клієнт тимчасово заблоковано.	FR-03, FR-04	Розумний back-off.
FR-16	Локальний захист Face ID/Touch ID (iOS)	Доступ до застосунку/секцій можливий після локальної біометрії iOS.	Високий	Given пристрій з Face ID/Touch ID; When ввімкнено локальний захист; Then доступ можливий лише після успішної перевірки.	Мобільний клієнт	Використовувати LocalAuthentication.
FR-17	Локальний захист Face ID/Touch ID (iOS)	Доступ до застосунку/секцій можливий після локальної біометрії iOS.	Високий	Given пристрій з Face ID/Touch ID; When ввімкнено локальний захист; Then доступ можливий лише після успішної перевірки.	Мобільний клієнт	Використовувати LocalAuthentication.
FR-18	Сповіщення про критичні події	Надсилання сповіщень про входи з нових пристроїв, зміну біометрії, відновлення доступу.	Низький	Given подія безпеки; When сталася; Then користувач отримує повідомлення (push/email).	FR-12, FR-13	Керовані користувачем налаштування.

2.2 Практична реалізація бекенду

Клас User представляє базову сутність користувача в системі. Використання анотацій @Entity і @Table забезпечує автоматичне зіставлення класу з таблицею бази даних [33] [37]. Поле id є первинним ключем і генерується автоматично. Поле email унікальне й обов'язкове, що унеможлиблює дублювання користувачів. Атрибути firstName і lastName зберігають персональні дані, а password використовується для класичної аутентифікації.

Реалізація інтерфейсу UserDetails дозволяє інтегрувати користувача у Spring Security, що робить цю сутність ключовим елементом механізму авторизації та контролю доступу. У повній версії клас доповнюється зв'язками з ролями й ембедингами облич, що створює комплексну модель користувача з підтримкою біометричної аутентифікації.

@Entity

@Table(name = "users")

public class User implements UserDetails {

 @Id

 @GeneratedValue(strategy = GenerationType.IDENTITY)

 private Long id;

 @Column(unique = true, nullable = false)

 private String email;

 private String firstName;

 private String lastName;

 @Column(nullable = false)

 private String password;

}

Сутність Face зберігає біометричні дані користувача у вигляді ембеддингів . Поле faceEmbedding визначено як @Lob, що дозволяє зберігати великі масиви байтів, які генеруються нейронною мережею під час обробки зображення обличчя . Це ключовий елемент системи, адже саме векторні представлення слугують основою для порівняння й аутентифікації.

Анотація @ManyToOne задає зв'язок між ембеддингом і користувачем, до якого він належить. Таким чином, один користувач може мати кілька зареєстрованих зразків обличчя, що підвищує точність і зручність входу . Завдяки цьому забезпечується підтримка персоналізованої та надійної біометричної аутентифікації .

@Entity

@Table(name = "faces")

public class Face {

 @Id

 @GeneratedValue(strategy = GenerationType.IDENTITY)

 private Long id;

 @Lob

 @Column(name = "face_embedding", nullable = false)

 private byte[] faceEmbedding;

 @ManyToOne(fetch = FetchType.LAZY)

 @JoinColumn(name = "user_id", nullable = false)

 private User user;

}

UserRepository забезпечує доступ до таблиці користувачів через механізми Spring Data JPA [30]. Наслідування від JpaRepository дає змогу автоматично

отримати базові CRUD-операції (create, read, update, delete), що суттєво спрощує розробку. Метод `findByEmail` визначений окремо, оскільки електронна пошта є унікальним ідентифікатором користувача у системі .

Завдяки використанню `Optional` при пошуку виключається можливість `NullPointerException`, а також полегшується обробка відсутніх записів. Такий підхід робить репозиторій надійним інструментом для роботи з даними користувачів, особливо в контексті аутентифікації та перевірки унікальності при реєстрації.

@Repository

```
public interface UserRepository extends JpaRepository<User, Long> {  
    Optional<User> findByEmail(String email);  
}
```

Практична реалізація розпізнавання обличчя

`FaceRepository` реалізує доступ до даних про біометричні ембеддинги облич . Оскільки сутність `Face` безпосередньо пов'язана з користувачем, даний репозиторій дозволяє ефективно виконувати CRUD-операції над біометричними зразками.

Навіть без додаткових методів інтерфейс забезпечує повний набір можливостей для збереження та видалення ембеддингів, що критично важливо під час навчання системи на нових зразках або видалення даних користувача. При потребі його можна легко розширити методами для пошуку облич за `user_id` чи іншими критеріями.

@Repository

```
public interface FaceRepository extends JpaRepository<Face, Long> {  
    // Базові CRUD-операції  
}
```

`RoleRepository` відповідає за роботу з таблицею ролей. Метод `findByName` дозволяє швидко отримати роль за її назвою (`ROLE_USER` чи `ROLE_ADMIN`). Це

особливо важливо при реєстрації, коли користувачу потрібно автоматично призначити стандартну роль.

Використання Spring Data JPA забезпечує прозоре керування ролями та інтеграцію з механізмами Spring Security [24]. Таким чином, доступ до функцій додатку можна гнучко регулювати через призначення різних рівнів прав користувачам.

@Repository

```
public interface RoleRepository extends JpaRepository<Role, Long> {  
    Optional<Role> findByName(Role.RoleName name);  
}
```

На першому етапі виконується детекція обличчя на зображенні, яке надходить від клієнтського застосунку у вигляді файлу. Для цього використовується бібліотека OpenCV та каскадний класифікатор локальних бінарних шаблонів (LBP), що забезпечує швидку та ефективну обробку зображень у реальному часі. Завантажене фото попередньо перетворюється у формат Mat, після чого алгоритм detectMultiScale виконує пошук області, що відповідає обличчю. У разі успішної детекції система повертає обрізаний фрагмент зображення, який надалі використовується для формування біометричного шаблону.

```
public Mat detectFace(MultipartFile photo) throws IOException {  
    Mat img = convertMultipartFileToMat(photo);  
    MatOfRect faceDetections = new MatOfRect();  
    faceDetector.detectMultiScale(img, faceDetections);  
    for (Rect rect : faceDetections.toArray()) {  
        return new Mat(img, rect);  
    }  
    return null;  
}
```

```
}
```

Наступним етапом є генерація векторного представлення обличчя за допомогою нейронної мережі FaceNet. Перед передачею зображення у модель виконується нормалізація даних, приведення колірному простору до RGB та перетворення типу даних у формат з плаваючою комою. Обличчя масштабується до розміру 160×160 пікселів, що відповідає вхідним вимогам моделі. Для виконання інференсу використовується TensorFlow Java API, який завантажує попередньо натреновану модель у вигляді замороженого графа.

```
public byte[] getEmbedding(Mat photo) {  
    if (photo.channels() == 1) {  
        Imgproc.cvtColor(photo, photo, Imgproc.COLOR_GRAY2RGB);  
    } else if (photo.channels() == 4) {  
        Imgproc.cvtColor(photo, photo, Imgproc.COLOR_BGRA2BGR);  
    }  
    photo.convertTo(photo, CvType.CV_32F);  
    try (Graph graph = new Graph()) {  
        graph.importGraphDef(faceNetGraphDef);  
        try (Session session = new Session(graph)) {  
            float[] imageTensor = convertMatToFloatArray(photo);  
            Tensor<Float> inputTensor = Tensor.create(  
                new long[]{1, 160, 160, 3},  
                FloatBuffer.wrap(imageTensor)  
            );
```

Результатом роботи нейронної мережі є 512-вимірний вектор ознак, який є компактним і стійким описом індивідуальних рис обличчя користувача. Такий підхід дозволяє уникнути зберігання вихідних зображень, підвищуючи рівень захисту персональних даних.

Для визначення схожості між двома обличчями використовується метрика косинусної подібності. Векторні представлення декодуються з байтового формату

у масиви чисел з плаваючою комою, після чого обчислюється значення косинуса кута між ними. Чим ближче отримане значення до одиниці, тим більш схожими є обличчя. У системі використовується порогове значення 0,6, яке експериментально підібране як компроміс між точністю та кількістю хибних спрацьовувань.

```
public boolean compareEmbedding(byte[] face1, byte[] face2) {  
    float[] embedding1 = new float[face1.length / Float.BYTES];  
    float[] embedding2 = new float[face2.length / Float.BYTES];  
    ByteBuffer.wrap(face1).asFloatBuffer().get(embedding1);  
    ByteBuffer.wrap(face2).asFloatBuffer().get(embedding2);  
  
    double dotProduct = 0.0;  
    double normA = 0.0;  
    double normB = 0.0;  
    for (int i = 0; i < embedding1.length; i++) {  
        dotProduct += embedding1[i] * embedding2[i];  
        normA += Math.pow(embedding1[i], 2);  
        normB += Math.pow(embedding2[i], 2);  
    }  
    double similarity = dotProduct / (Math.sqrt(normA) * Math.sqrt(normB));  
    return similarity > 0.6;  
}
```

Завершальним етапом є інтеграція всіх компонентів у єдиний механізм автентифікації. Після отримання фото система виконує детекцію обличчя, генерує відповідний ембеддинг та порівнює його з усіма збереженими біометричними шаблонами у базі даних. У разі перевищення порогового значення подібності користувач вважається автентифікованим, і система повертає ідентифікатор відповідного облікового запису.

```
public Long loginWithFace(MultipartFile photo) throws IOException {
```

```

Mat detectedFace = detectFace(photo);
if (detectedFace != null) {
    byte[] currentEmbedding = getEmbedding(detectedFace);
    List<Face> allFaces = faceRepository.findAll();

    for (Face face : allFaces) {
        if (compareEmbedding(currentEmbedding, face.getFaceEmbedding())) {
            return face.getUser().getId();
        }
    }
}

```

Метод `init()` виконується після створення біну сервісу (`@PostConstruct`) і завантажує всі необхідні ресурси для роботи алгоритму розпізнавання. OpenCV ініціалізується локально, після чого підключається каскад LBP для детекції облич. Також завантажується попередньо натренована модель FaceNet у вигляді protobuf-файлу.

Такий підхід дозволяє уникнути повторного завантаження ресурсів при кожному запиті, що суттєво знижує затримки та підвищує продуктивність. Завдяки цьому система готова до обробки зображень відразу після старту й забезпечує стабільну роботу навіть у режимі високих навантажень.

`@PostConstruct`

```

public void init() throws IOException {
    OpenCV.loadLocally();
    faceDetector = new CascadeClassifier(
        "/app/extracted-resources/lbpcascade_frontalface_improved.xml"
    );
    faceNetGraphDef = Files.readAllBytes(
        new File("/app/extracted-resources/20180402-114759.pb").toPath()
    );
}

```

Метод `getEmbedding` перетворює вирізане обличчя у вектор-ембеддинг за допомогою TensorFlow-моделі. Зображення приводиться до стандартного формату RGB і масштабується до числових значень типу `float`. Після цього формується тензор з розмірністю `[1, 160, 160, 3]`, який подається на вхід нейронної мережі FaceNet.

Результатом є масив чисел (512 значень), який у байтовому вигляді зберігається в базі. Саме ці вектори слугують основою для обчислення схожості між обличчями при вході в систему. Це дозволяє виконувати аутентифікацію без збереження самих фото, що підвищує безпеку та відповідає сучасним вимогам конфіденційності.

```
public byte[] getEmbedding(Mat photo) {
    Imgproc.cvtColor(photo, photo, Imgproc.COLOR_BGR2RGB);
    photo.convertTo(photo, CvType.CV_32F);
    try (Graph graph = new Graph()) {
        graph.importGraphDef(faceNetGraphDef);
        try (Session session = new Session(graph)) {
            float[] tensorData = convertMatToFloatArray(photo);
            Tensor<Float> inputTensor =
                Tensor.create(new long[]{1, 160, 160, 3}, FloatBuffer.wrap(tensorData));
            List<Tensor<?>> outputs = session.runner()
                .feed("input", inputTensor)
                .feed("phase_train", Tensor.create(false))
                .fetch("embeddings").run();
            return toByteArray(outputs.get(0));
        }
    }
}
```

Метод `compareEmbedding` обчислює схожість між двома обличчями за допомогою косинусної подібності [39] [42]. Кожен ембеддинг перетворюється у масив чисел, після чого обчислюється скалярний добуток та норми векторів . Результат нормалізується у вигляді коефіцієнта подібності від -1 до 1 .

Встановлений поріг 0.6 означає, що якщо схожість перевищує цей рівень, система вважає обличчя ідентичними . Це дозволяє балансувати між точністю та толерантністю до варіацій, таких як освітлення чи міміка . Такий підхід є ефективним і водночас обчислювально простим для роботи в реальному часі .

```
public boolean compareEmbedding(byte[] face1, byte[] face2) {  
    float[] emb1 = toFloatArray(face1);  
    float[] emb2 = toFloatArray(face2);  
  
    double dotProduct = 0, normA = 0, normB = 0;  
    for (int i = 0; i < emb1.length; i++) {  
        dotProduct += emb1[i] * emb2[i];  
        normA += Math.pow(emb1[i], 2);  
        normB += Math.pow(emb2[i], 2);  
    }  
  
    double similarity = dotProduct / (Math.sqrt(normA) * Math.sqrt(normB));  
    return similarity > 0.6;  
}
```

Авторизація та розгортання бекенду

Метод `register` відповідає за повний процес створення нового користувача. Спершу перевіряється унікальність електронної пошти, щоб уникнути дублювання. Далі з DTO створюється об'єкт користувача, пароль кодується за допомогою `BCryptPasswordEncoder` для захисту від атак типу `brute-force` та `rainbow-tables`.

Після цього користувачу призначається стандартна роль `ROLE_USER`, і він зберігається в базі даних. Завершальний етап — реєстрація біометрії: фотографія обробляється сервісом `FaceService`, де створюється ембеддинг і зберігається у зв'язку з користувачем. Таким чином, класична та біометрична аутентифікації поєднуються в єдиному процесі, що підвищує як безпеку, так і зручність для користувача.

```
@Override
```

```
@Transactional
```

```
public UserResponseDto register(UserRegisterRequestDto requestDto, MultipartFile photo) throws IOException {
```

```
    if (userRepository.findByEmail(requestDto.getEmail()).isPresent()) {  
        throw new RegistrationException("User with this email already exists");  
    }
```

```
    User user = userMapper.toModel(requestDto);
```

```
    user.setPassword(passwordEncoder.encode(requestDto.getPassword()));
```

```
    Role userRole = roleRepository.findByName(Role.RoleName.ROLE_USER)
```

```
        .orElseThrow(() -> new RuntimeException("Cannot find ROLE_USER"));
```

```
    user.setRoles(Set.of(userRole));
```

```
    User savedUser = userRepository.save(user);
```

```
    boolean isFaceRegistered = faceService.registerUserAndFace(savedUser, photo);
```

```
    if (!isFaceRegistered) throw new RegistrationException("Face registration failed");
```

```
    return userMapper.toDto(savedUser);
```

```
}
```

Метод `loginByPhoto` реалізує ключову функцію — вхід у систему за допомогою біометрії. Завантажене фото аналізується в `FaceService`, де генерується ембеддинг і порівнюється з наявними у базі. Якщо знаходиться відповідність,

витагується користувач, і для нього створюється JWT-токен, який передається клієнту.

У випадку помилок (наприклад, відсутність відповідного обличчя або проблеми з обробкою файлу) кидається виняток `FaceLoginException`. Такий підхід робить процес аутентифікації прозорим і безпечним: замість передачі паролів мережевим каналом використовується надійний криптографічний токен. Це також дозволяє будувати stateless-архітектуру, де сервер не зберігає сесій, а лише перевіряє валідність токена.

@Override

```
public UserLoginResponseDto loginByPhoto(MultipartFile photo) {
    try {
        Long userId = faceService.loginWithFace(photo);
        if (userId != null) {
            User user = userRepository.findById(userId)
                .orElseThrow(() -> new RuntimeException("User not found"));
            String token = jwtTokenProvider.generateToken(user.getEmail());
            return new UserLoginResponseDto(token);
        }
    } catch (IOException e) {
        throw new FaceLoginException("An error occurred while processing the photo", e);
    }
    throw new FaceLoginException("No matching face found for login");
}
```

Клас `UserController` надає REST-інтерфейси для взаємодії клієнта з системою аутентифікації [32]. Метод `/register` дозволяє зареєструвати нового користувача, отримуючи одночасно дані анкети (`UserRegisterRequestDto`) та фотографію (`MultipartFile`). Використання `@RequestPart` дає змогу обробляти multipart-запити, що важливо для передачі зображень.

Валідація даних (@Validated) перевіряє коректність введених параметрів, наприклад формат email чи довжину пароля. У результаті клієнт отримує DTO-відповідь з базовою інформацією про зареєстрованого користувача . Завдяки такій структурі API є простим і водночас гнучким для інтеграції з мобільним застосунком .

```
@RestController
```

```
@RequiredArgsConstructor
```

```
@RequestMapping("/auth")
```

```
public class UserController {
```

```
    private final AuthenticationService authenticationService;
```

```
    @PostMapping("/register")
```

```
    public ResponseEntity<UserResponseDto> register(
```

```
        @RequestPart("user") @Validated UserRegisterRequestDto requestDto,
```

```
        @RequestPart("photo") MultipartFile photo) throws IOException {
```

```
        return ResponseEntity.ok(authenticationService.register(requestDto, photo));
```

```
    }
```

```
}
```

Метод /login/photo надає можливість входу в систему без введення пароля — лише за фото. Він викликає сервіс аутентифікації, де перевіряється відповідність обличчя у базі даних . У разі успіху користувач отримує JWT-токен для подальших запитів.

Метод /info використовує контекст безпеки Spring Security для витягування email автентифікованого користувача . Це дозволяє клієнту отримати свої дані без повторного введення ідентифікаторів . Таким чином, API реалізує повний цикл взаємодії: від реєстрації й логіну до доступу до персональної інформації .

```
@PostMapping("/login/photo")
```

```

public ResponseEntity<UserLoginResponseDto> authenticateByPhoto(
    @RequestPart("photo") MultipartFile photo) {
    return ResponseEntity.ok(authenticationService.loginByPhoto(photo));
}

@GetMapping("/info")
public ResponseEntity<UserResponseDto> getUserDetails() {
    String email = SecurityContextHolder.getContext().getAuthentication().getName();
    return ResponseEntity.ok(authenticationService.getUserByEmail(email));
}

```

Клас `JwtTokenProvider` реалізує генерацію та перевірку tokenів JWT. У конструкторі зчитується секретний ключ із конфігурації й створюється криптографічний ключ для підпису. Метод `generateToken` формує token, який містить у собі email користувача, час створення та час завершення дії.

JWT-токени дозволяють організувати stateless-аутентифікацію: сервер не зберігає сесії, а лише перевіряє валідність токена у кожному запиті. Це підвищує масштабованість та безпеку системи. Використання криптографічного підпису гарантує, що token неможливо підробити чи змінити без знання секретного ключа.

```

@Component
public class JwtTokenProvider {
    private final SecretKey secretKey;

```

Клас `WebSecurityConfig` визначає правила доступу до ресурсів системи . Вимикається CSRF-захист (оскільки система працює у форматі REST API), а доступ до методів `/register` та `/login` відкритий для всіх. Решта запитів вимагають автентифікації . Сесійна політика визначена як `STATELESS`, тобто сервер не зберігає стан користувача .

JWT-фільтр (jwtFilter) інтегрується у фільтраційний ланцюг перед UsernamePasswordAuthenticationFilter . Це дозволяє перехоплювати кожен запит і перевіряти наявність та дійсність токена . Окремо визначено PasswordEncoder на основі BCrypt, що забезпечує надійне хешування паролів. У сукупності ці механізми формують безпечну та гнучку архітектуру доступу .

Dockerfile реалізує двоетапну збірку . На першому етапі (builder) створюється JAR-файл за допомогою Maven у середовищі JDK . Це дозволяє відокремити процес компіляції від виконання . На другому етапі (runtime) використовується легший образ JRE, де встановлюються лише необхідні залежності (OpenCV, curl, unzip).

@Configuration

@EnableMethodSecurity

@RequiredArgsConstructor

```
public class WebSecurityConfig {  
    private final JwtAuthenticationFilter jwtFilter;
```

@Bean

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
    http.csrf(AbstractHttpConfigurer::disable)  
        .authorizeHttpRequests(auth -> auth  
            .requestMatchers("/auth/register", "/auth/login", "/auth/login/photo")  
            .permitAll()  
            .anyRequest().authenticated()  
        .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))  
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);  
    return http.build();  
}
```

->

```

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
}

```

Модель нейронної мережі розпаковується із JAR-файлу у директорію /app/extracted-resources, щоб зробити її доступною для OpenCV і TensorFlow під час виконання. Відкривається порт 8081, і контейнер запускає застосунок через java - jar. Така стратегія зменшує розмір образу та підвищує його портативність.

```

FROM eclipse-temurin:17-jdk-jammy as builder
RUN apt-get update && apt-get install -y maven
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline -B
COPY src ./src

```

Файл docker-compose.yml автоматизує запуск застосунку у контейнері. Він будує образ із локального Dockerfile, відкриває порт 8081 і передає змінні середовища (профіль Spring Boot та параметри JVM). Моделі ШІ монтуються через volumes, що дозволяє зберігати їх окремо від контейнера та швидко оновлювати .

```

version: '3.8'

services:
  ai-face-authentication:
    build: .
    ports:
      - "8081:8081"
    environment:
      - SPRING_PROFILES_ACTIVE=dev

```

- JAVA_OPTS=-Xmx512m -Xms256m

Вбудований healthcheck регулярно перевіряє стан сервісу через ендпоінт /actuator/health. Це гарантує, що контейнер вважається працездатним лише тоді, коли сам застосунок успішно запустився. Таким чином, Docker Compose забезпечує зручне розгортання, моніторинг і подальше масштабування системи.

2.3 Практична реалізація мобільного застосунку

У цьому фрагменті реалізовано головний екран аутентифікації з можливістю перемикання між класичним входом, реєстрацією та входом за допомогою розпізнавання обличчя. Використано декларативний підхід SwiftUI, де стан authMode автоматично визначає, яка форма відображатиметься. Такий підхід мінімізує імперативну логіку, спрощує розширення функціональності та покращує читабельність коду.

```
struct AuthView: View {
    @EnvironmentObject private var viewModel: AppViewModel
    var body: some View {
        NavigationStack {
            VStack {
                Picker("Mode", selection: $viewModel.authMode) {
                    Text("Login").tag(AuthMode.login)
                    Text("Register").tag(AuthMode.register)
                    Text("Face Login").tag(AuthMode.face)
                }
                .pickerStyle(.segmented)
            }
        }
    }
}
```

Центральна бізнес-логіка керування станом реалізована у класі `AppViewModel`, який діє як сполучна ланка між інтерфейсом і мережевими запитами [29]. За допомогою `@Published` забезпечується реактивне оновлення інтерфейсу при зміні стану користувача, токена або даних. Асинхронна модель `async/await` у поєднанні з `APIClient` забезпечує плавну обробку аутентифікаційних операцій, тоді як `KeychainService` гарантує безпечне зберігання токенів доступу.

`@MainActor`

```
final class AppViewModel: ObservableObject {
    @Published var token: String?
    @Published var user: UserProfile?
    private let api = APIClient()
    private let keychain = KeychainService()
    func login(email: String, password: String) {
        Task {
            do {
                let auth = try await api.login(email: email, password: password)
                let profile = try await api.fetchCurrentUser(token: auth.token)
                token = auth.token
                user = profile
                keychain.storeToken(auth.token)
            } catch {
                print("Login error:", error)
            }
        }
    }
}
```

Функціонал управління нотатками реалізовано у вигляді окремого представлення `NotesView`, що використовує спільний стан з `AppViewModel`. Завдяки реактивності `SwiftUI` користувач миттєво бачить зміни у списку після

створення, редагування чи видалення нотаток . Дані серіалізуються у форматі JSON і зберігаються локально через NotesStore, що забезпечує автономну роботу застосунку навіть без мережевого з'єднання.

```
struct NotesView: View {
    @EnvironmentObject private var viewModel: AppViewModel
    var body: some View {
        NavigationStack {
            List(viewModel.notes) { note in
                Text(note.title)
            }
        }
    }
}
```

Компонент rootView відповідає за маршрутизацію між основними екранами застосунку. Якщо користувач автентифікований, відображається NotesView, інакше — екран авторизації AuthView. Через спостереження за @Published змінними user та token забезпечується автоматичне перемикання без ручного керування станом. Реалізовано відображення індикатора завантаження та системних повідомлень через SwiftUI Alert .

```
struct rootView: View {
    @EnvironmentObject private var viewModel: AppViewModel
    var body: some View {
        ZStack {
            if viewModel.user != nil, viewModel.token != nil {
                NotesView()
            } else {
                AuthView()
            }
        }
    }
}
```

```
}
```

Мережевий клієнт `APIClient` реалізує усі запити до серверної частини: реєстрацію, вхід, біометричну автентифікацію та отримання профілю користувача. Використано асинхронну модель `async/await`, що спрощує обробку відповідей без замикань. Кожен метод формує `URLRequest`, серіалізує дані у `JSON` і передає їх через загальну функцію `send`.

```
struct APIClient {  
    private let baseURL: URL  
    private let urlSession: URLSession = .shared  
    func login(email: String, password: String) async throws -> AuthResponse {  
        return try await send(request)  
    }  
}
```

Модуль `FaceLoginView` забезпечує автентифікацію користувача через розпізнавання обличчя. Інтерфейс дає змогу зробити фото або вибрати його з галереї, після чого викликається метод `loginWithPhoto`. Зображення надсилається на сервер для порівняння з попередньо зареєстрованими біометричними даними. Компонент реалізований декларативно, із плавними анімаціями та адаптивною логікою для станів зйомки, очікування та успіху.

```
struct FaceLoginView: View {  
    @EnvironmentObject private var viewModel: AppViewModel  
    @State private var capturedImage: UIImage?  
    var body: some View {  
        VStack {  
            Button("Capture Face") { openCamera() }  
            if let image = capturedImage {  
                Image(uiImage: image).resizable().scaledToFit()  
                Button("Authenticate") { viewModel.loginWithPhoto(image) }  
            }  
        }  
    }  
}
```

```

    }
  }
}

```

Безпечне зберігання токена реалізоване в сервісі KeychainService, який використовує системний API Security для роботи з Keychain. Цей підхід гарантує, що чутливі дані (токени або облікові ключі) не потрапляють у файлову систему користувача у відкритому вигляді. Реалізовано методи додавання, читання та видалення токенів, що дозволяє підтримувати автоматичний вхід без повторного введення пароля.

```

final class KeychainService {
  private let account = "ai-auth-token"
  func storeToken(_ token: String) {
    let data = token.data(using: .utf8)!
    let query: [String: Any] = [
      kSecClass as String: kSecClassGenericPassword,
      kSecAttrAccount as String: account,
      kSecValueData as String: data
    ]
    SecItemAdd(query as CFDictionary, nil)
  }
}

```

Модель UserProfile описує структуру користувача, отриману з API. Вона підтримує декодування ідентифікатора як рядка або цілого числа, що забезпечує сумісність із різними типами серверних реалізацій. Цей підхід зменшує ризик помилок серіалізації й полегшує обмін даними між фронтендом і бекендом.

```

struct UserProfile: Codable {
  let id: String
  let email: String
}

```

```
let firstName: String?  
let lastName: String?  
}
```

2.4 База даних застосунку

Конфігурація визначає підключення до H2 Database, яка працює в режимі in-memory. Це означає, що всі дані зберігаються у оперативній пам'яті й автоматично видаляються після завершення роботи застосунку. Такий підхід зручний для розробки, тестування та прототипування, оскільки забезпечує швидке створення таблиць і простоту розгортання без додаткових налаштувань СУБД.

Опція `ddl-auto: create-drop` вказує Hibernate створювати схему при запуску і видаляти її після завершення роботи. Це дозволяє кожного разу отримувати «чисту» базу для тестування нових змін. Водночас у промислових системах зазвичай використовуються PostgreSQL або MySQL із постійним збереженням даних.

```
spring:  
  datasource:  
    url: jdbc:h2:mem:testDb  
    username: admin  
    password: root  
    driver-class-name: org.h2.Driver  
  jpa:  
    hibernate:  
      ddl-auto: create-drop
```

Таблиця `users` містить основні атрибути користувачів: унікальний ідентифікатор `id`, електронну пошту, ім'я, прізвище та пароль. Поле `email` є унікальним, що гарантує однозначність облікового запису. Усі текстові поля мають обмеження `NOT NULL`, аби уникнути створення неповних записів.

Пароль зберігається у вигляді зашифрованого рядка, що забезпечує базовий рівень безпеки. Хоча в тестовій конфігурації дані тимчасові, структура таблиці відповідає стандартним вимогам до промислових систем аутентифікації.

```
CREATE TABLE users (  
    id BIGINT AUTO_INCREMENT PRIMARY KEY,  
    email VARCHAR(255) NOT NULL UNIQUE,  
    first_name VARCHAR(255) NOT NULL,  
    last_name VARCHAR(255) NOT NULL,  
    password VARCHAR(255) NOT NULL  
);  
  
@Entity  
@Table(name = "users")  
public class User implements UserDetails {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;
```

Клас User є JPA-сутністю, яка напряду відображається у таблицю users. Поле id автоматично генерується базою даних. Атрибут email позначено як унікальний і обов'язковий, що відповідає вимогам бізнес-логіки. Інші поля зберігають ім'я, прізвище та пароль користувача.

Розширення інтерфейсу UserDetails робить цю сутність сумісною з Spring Security. Це означає, що користувач одразу інтегрується у механізми авторизації та управління доступом. У повній версії клас включає також зв'язки з ролями та біометричними даними.

```
@Column(unique = true, nullable = false)  
private String email;  
private String firstName;
```

```

private String lastName;
@Column(nullable = false)
private String password;
}

```

Таблиця roles визначає рівні доступу у системі. Вона містить ідентифікатор ролі та її назву, яка є унікальною. Додавання початкових значень ROLE_USER і ROLE_ADMIN створює базовий поділ на звичайних користувачів і адміністраторів. Це дозволяє легко впроваджувати контроль доступу в межах Spring Security.

```

CREATE TABLE roles (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE
);
INSERT INTO roles (name) VALUES ('ROLE_USER');
INSERT INTO roles (name) VALUES ('ROLE_ADMIN');

```

На рисунку 2.2 наведено схему бази даних.

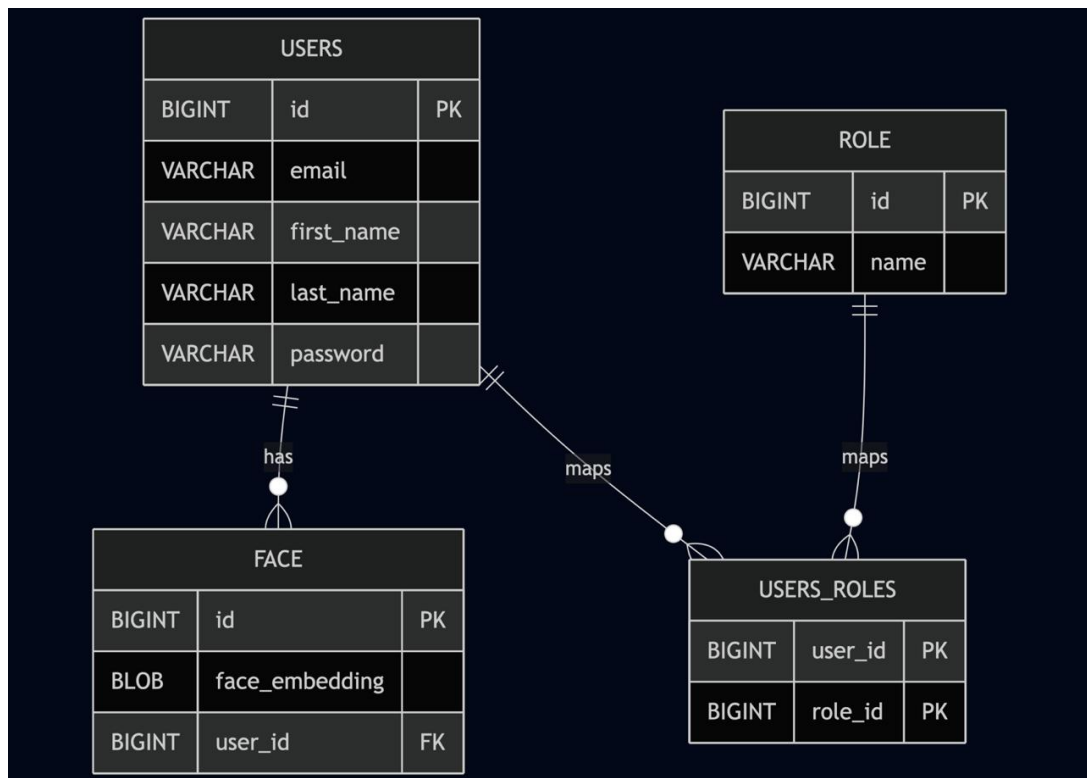


Рис 2.2 – Схема БД

У таблиці 2.2 подано розгорнутий та узагальнений опис основних сутностей, що входять до структури розроблюваної бази даних, а саме таблиць users, roles, faces та users_roles.

Схема БД

Таблиця	Основні поля	Призначення
users	id (PK), email (UNIQUE), first_name, last_name, password	Зберігає дані користувачів, які мають доступ до системи
roles	id (PK), name (UNIQUE)	Містить ролі доступу (наприклад, ROLE_USER, ROLE_ADMIN)
faces	id (PK), user_id (FK), face_embedding	Біометричні ембеддинги для аутентифікації користувачів за обличчям
users_roles	user_id (PK, FK), role_id (PK, FK)	Зв'язувальна таблиця для реалізації відношення «багато-до-багатьох» між користувачами та ролями

Висновки до другого розділу

У другому розділі було проведено аналіз основних варіантів використання застосунку та реалізовано практичні приклади їх технічного втілення. Визначено, що ключовими сценаріями взаємодії користувача із системою є реєстрація облікового запису, автентифікація за біометричними ознаками, створення, редагування, видалення та перегляд нотаток, а також додаткові функції синхронізації між пристроями, керування сесіями та відновлення доступу. Усі вони базуються на принципі поєднання зручності використання з високим рівнем інформаційної безпеки.

У практичній частині продемонстровано реалізацію основних сутностей системи (користувач, обличчя, роль), доступ до них через репозиторії Spring Data JPA та обробку біометричних даних за допомогою OpenCV і TensorFlow. Окрему увагу приділено сервісам реєстрації й аутентифікації, де класична модель із

паролем доповнена біометричною перевіркою обличчя, що забезпечує надійний і сучасний механізм ідентифікації.

Також у розділі описано архітектуру REST API, яка надає зовнішнім клієнтам зручні інтерфейси для взаємодії з системою, і розкрито механізми безпеки, побудовані на Spring Security та JWT. Вони дозволяють реалізувати stateless-модель захисту, коли сервер не зберігає стану користувача, а кожен запит підтверджується криптографічно захищеним токеном. Завдяки цьому досягається баланс між продуктивністю, масштабованістю й конфіденційністю.

Отже, результати другого розділу засвідчують практичну реалізованість і ефективність запропонованого підходу. Система демонструє інтеграцію сучасних технологій у сфері мобільної безпеки, що робить її конкурентоспроможною серед аналогів і закладає підґрунтя для подальшого вдосконалення функціональності та масштабування.

РОЗДІЛ 3. ТЕСТУВАННЯ ЗАСТОСУНКУ ТА РОЗГОРТАННЯ

3.1 Опис тест кейсів

В даному підрозділі будуть описані тест кейси для розробленого застосунку та результати перевірки.

Даний тест перевіряє коректність обробки сценарію, коли користувач намагається зареєструватися з електронною адресою, яка вже існує у базі даних. Він емулює поведінку репозиторію користувачів, повертаючи вже наявний запис, після чого очікує виникнення винятку `RegistrationException`. Тест також перевіряє відповідність тексту помилки очікуваному повідомленню, що підвищує надійність перевірки. Крім того, перевіряється, що метод збереження нового користувача не викликається, якщо валідація не пройдена. Таким чином, забезпечується коректне функціонування бізнес-логіки реєстрації та запобігання дублюванню облікових записів у системі.

@Test

```
void register_WhenUserAlreadyExists_ShouldThrowRegistrationException() {  
  
    when(userRepository.findByEmail(registerRequest.getEmail())).thenReturn(Optional.of  
(user));  
  
    RegistrationException exception = assertThrows(RegistrationException.class,  
        () -> authenticationService.register(registerRequest, multipartFile));  
    assertEquals("User with this email already exists: test@example.com",  
        exception.getMessage());  
  
    verify(userRepository).findByEmail(registerRequest.getEmail());  
    verify(userRepository, never()).save(any(User.class));  
}
```

Цей тест перевіряє алгоритм порівняння ембеддингів обличчя, що використовується для визначення подібності між двома біометричними зразками. Для перевірки беруться два ідентичні вектори, які представляють ту саму особу.

Метод `compareEmbedding` обчислює косинусну подібність між ними, і очікується, що результат буде істинним. У такий спосіб тест підтверджує правильність реалізації математичного алгоритму, який лежить в основі системи розпізнавання облич. Успішне проходження цього тесту гарантує, що система здатна стабільно ідентифікувати одного користувача за різними, але еквівалентними біометричними даними.

@Test

```
void compareEmbedding_WithSimilarEmbeddings_ShouldReturnTrue() {  
    byte[] embedding1 = testEmbedding.clone();  
    byte[] embedding2 = testEmbedding.clone();  
    boolean result = faceService.compareEmbedding(embedding1, embedding2);  
    assertTrue(result);  
}
```

У цьому тесті здійснюється перевірка інтеграційного рівня контролера, що обробляє запит на авторизацію користувача. Метод імітує HTTP-запит до ендпоінта `/auth/login` із коректними обліковими даними та очікує відповідь із кодом стану 200 OK. Тест підтверджує, що у відповіді присутній JWT-токен, який генерується у випадку успішної автентифікації. Таким чином, він перевіряє повний цикл взаємодії між клієнтом і сервером, включаючи маршалізацію даних у форматі JSON, обробку запиту контролером, виклик бізнес-логіки сервісу та формування коректної відповіді API.

@Test

```
void login_WithValidCredentials_ShouldReturnToken() throws Exception {  
  
    when(authenticationService.login(any(UserLoginRequestDto.class))).thenReturn(login  
Response);  
    mockMvc.perform(post("/auth/login")  
        .contentType(MediaType.APPLICATION_JSON)
```

```
.content(objectMapper.writeValueAsString(loginRequest)))  
.andExpect(status().isOk())  
.andExpect(jsonPath("$.token").value("jwt-token"));  
}
```

Цей тест призначений для перевірки функціональності генерації та валідації JWT-токенів у системі безпеки. Він створює токен на основі дійсної електронної адреси користувача та перевіряє, що отримане значення не є порожнім, має коректну структуру та містить три частини, розділені крапками. Додатково тест перевіряє, чи можна з токена коректно вилучити електронну адресу користувача, що підтверджує цілісність підпису й правильність алгоритму кодування. Таким чином, тест забезпечує надійність роботи механізму аутентифікації, що ґрунтується на JWT.

@Test

```
void generateToken_WithValidEmail_ShouldReturnValidToken() {  
    String email = "test@example.com";  
    String token = jwtTokenProvider.generateToken(email);  
    assertNotNull(token);  
    assertFalse(token.isEmpty());  
    assertTrue(token.contains("."));  
    String extractedEmail = jwtTokenProvider.getEmail(token);  
    assertEquals(email, extractedEmail);  
}
```

Даний тест перевіряє правильність обробки виключних ситуацій, які виникають під час біометричної автентифікації. У тесті штучно створюється об'єкт винятку `FaceLoginException` із повідомленням про відсутність збігу обличчя. Обробник помилок `GlobalExceptionHandler` має повернути відповідь HTTP 401 `Unauthorized` із тілом, що містить структурований JSON-опис помилки. Тест підтверджує, що повідомлення у відповіді містить як заголовок помилки, так і конкретну причину, що дає змогу клієнтському застосунку відобразити її

користувачу. Перевірка цілісності відповіді забезпечує стабільність і передбачуваність поведінки системи у разі збоїв.

```
@Test
```

```
void
```

```
handleFaceLoginException_WithSimpleMessage_ShouldReturnUnauthorizedWithJson(
) {
```

```
    FaceLoginException exception = new FaceLoginException("No matching face
found");
```

```
    ResponseEntity<String> response =
globalExceptionHandler.handleFaceLoginException(exception);
```

```
    assertNotNull(response);
```

```
    assertEquals(HttpStatus.UNAUTHORIZED, response.getStatusCode());
```

```
    assertTrue(response.getBody().contains("Face login failed"));
```

```
    assertTrue(response.getBody().contains("No matching face found"));
```

```
}
```

Тест перевіряє поведінку сервісу користувачів під час пошуку облікового запису за неіснуючою електронною адресою. Імітується ситуація, коли репозиторій не повертає жодного результату, що має спричинити виняток типу `NoSuchElementException`. Перевіряється, що текст повідомлення відповідає очікуваному шаблону, а метод пошуку у репозиторії викликається саме з переданою адресою. Цей тест підтверджує надійність обробки виняткових ситуацій, запобігає некоректному доступу до нульових об'єктів і гарантує передбачувану реакцію системи на відсутність даних у базі користувачів.

```
@Test
```

```
void findByEmail_WithNonExistingEmail_ShouldThrowNoSuchElementException() {
```

```
    String email = "nonexistent@example.com";
```

```
    when(userRepository.findByEmail(email)).thenReturn(Optional.empty());
```

```
    NoSuchElementException exception = assertThrows(NoSuchElementException.class,
        () -> userService.findByEmail(email));
```

```

    assertEquals("Cannot find user with email nonexistent@example.com",
exception.getMessage());
    verify(userRepository).findByEmail(email);
}

```

Цей тест перевіряє роботу механізму аутентифікації за фотографією в ситуації, коли система не знаходить відповідного біометричного шаблону в базі даних. Метод `loginWithFace` повертає `null`, і очікується, що сервіс автентифікації згенерує виняток `FaceLoginException` із відповідним повідомленням. Додатково тест перевіряє, що при помилці не відбувається звернення до репозиторію користувачів, що підтверджує правильність логічного ланцюга операцій. Така перевірка гарантує безпечну обробку невдалих спроб входу та правильне сповіщення користувача про причину відмови.

@Test

```

void loginByPhoto_WithNoMatchingFace_ShouldThrowFaceLoginException() throws
IOException {
    when(faceService.loginWithFace(multipartFile)).thenReturn(null);
    FaceLoginException exception = assertThrows(FaceLoginException.class,
        () -> authenticationService.loginByPhoto(multipartFile));
    assertEquals("No matching face found for login", exception.getMessage());
    verify(faceService).loginWithFace(multipartFile);
    verify(userRepository, never()).findById(anyLong());
}

```

Даний тест призначено для перевірки правильності реалізації сервісу користувацьких даних Spring Security. Він моделює ситуацію, коли в системі існує користувач із заданою електронною адресою, і перевіряє, що метод `loadUserByUsername` повертає коректний об'єкт `UserDetails`. Тест підтверджує, що всі властивості безпеки (незаблокований обліковий запис, дійсні облікові дані, активний статус) мають правильні значення, а також що ролі користувача належним чином завантажуються. Завдяки цьому забезпечується цілісність

взаємодії між сервісом авторизації та модулем безпеки Spring, що є ключовим аспектом для стабільної роботи системи.

@Test

```
void loadUserByUsername_WithExistingEmail_ShouldReturnUserDetails() {  
    String email = "test@example.com";  
    when(userRepository.findByEmail(email)).thenReturn(Optional.of(testUser));  
    UserDetails userDetails = customUserDetailsService.loadUserByUsername(email);  
    assertNotNull(userDetails);  
    assertEquals(testUser.getEmail(), userDetails.getUsername());  
    assertEquals(testUser.getPassword(), userDetails.getPassword());  
    assertTrue(userDetails.isAccountNonExpired());  
    assertTrue(userDetails.isAccountNonLocked());  
    assertTrue(userDetails.isCredentialsNonExpired());  
    assertTrue(userDetails.isEnabled());  
    assertNotNull(userDetails.getAuthorities());  
}
```

В таблиці 3.1 наведено результати автоматичної перевірки тесткейсів.

Таблиця 3.1

Результати автоматичної перевірки тесткейсів

№ тест кейсу	Назва тест кейсу	Результат
1	AuthenticationService Test — реєстрація існуючого користувача	Успішно пройдено
2	FaceService Test — порівняння ідентичних ембеддингів	Успішно пройдено
3	Controller Integration Test — авторизація з валідними даними	Успішно пройдено
4	JWT Security Test — генерація та перевірка JWT токена	Успішно пройдено

5	Exception Handling Test — обробка FaceLoginException	Успішно пройдено
6	UserService Test — пошук неіснуючого користувача	Успішно пройдено
7	Face Login Test — вхід без збігу обличчя	Успішно пройдено
8	Security User Details Test — завантаження користувача за email	Успішно пройдено

3.2 Розгортання бекенд частини застосунку

Розгортання серверної складової побудовано на контейнеризації з використанням Docker та організоване як двостадійна збірка. На етапі «builder» застосовується образ eclipse-temurin:17-jdk-jammy, у якому встановлюється Maven і виконується попереднє завантаження залежностей через команду `mvn dependency:go-offline`. Така стратегія мінімізує час повторних збірок завдяки кешуванню шару з артефактами Maven, оскільки `pom.xml` копіюється до контейнера раніше за вихідні коди. Після копіювання каталогу `src` виконується повний цикл складання `mvn clean package -DskipTests`, унаслідок чого формується виконуваний JAR-файл Spring Boot. Другий, «runtime»-етап використовує легший базовий образ eclipse-temurin:17-jre-jammy, що зменшує підсумковий розмір контейнера та скорочує поверхню атаки завдяки відсутності компіляторів і інструментів розробника.

Оскільки застосунок містить модулі комп'ютерного зору й інференсу нейронної мережі, у середовищі виконання встановлюються нативні залежності OpenCV та TensorFlow разом з бібліотеками лінійної алгебри та системними компонентами (зокрема `libopencv-dev`, `libopenblas0`, `liblapack3`, `libatlas3-base`, `libgomp1`, `libgcc-s1`, `libgfortran5`, а також `libglib2.0-0`, `libsm6`, `libxext6`, `libxrender-dev`). Це гарантує коректне завантаження нативних бінарів під час викликів з Java-

біндінгів. Зібраний JAR копіюється у каталог /app, а ресурси, необхідні для виконання (LBP-каскад, модельні файли, конфігурації), додатково розміщуються у файловій системі. Спеціальний крок розпаковує вміст app.jar у директорію /app/extracted-resources, щоб надати доступ до каскадів та інших артефактів поза клас-путями JAR'a під час рантайму. Така екстракція усуває залежність від внутрішньої структури Spring Boot-архіву та спрощує адресацію ресурсів нативними API.

З міркувань безпеки застосунок виконується не від імені привілейованого користувача: створюється система облікових записів appuser:appuser, власником якої призначається робочий каталог. Додатково виділяється записуваний тимчасовий каталог /app/tmp, а змінна JVM -Djava.io.tmpdir=/app/tmp вмикає ізольоване використання тимчасових файлів усередині контейнера. На рівні продуктивності JVM налаштовано -Xmx512m -Xms256m та збирач сміття G1 (-XX:+UseG1GC) із підтримкою контейнерних лімітів (-XX:+UseContainerSupport), що забезпечує передбачуване споживання пам'яті й стабільну латентність у середовищі з обмеженими ресурсами. Для зниження шуму логування TensorFlow використовується змінна середовища TF_CPP_MIN_LOG_LEVEL=2, а TF_ENABLE_ONEDNN_OPTS=0 вимикає специфічні оптимізації, які на деяких конфігураціях CPU можуть спричиняти варіативність продуктивності та нетривіальні залежності від наборів інструкцій [28].

Життєздатність контейнера контролюється через механізм перевірок стану: у Dockerfile та docker-compose.yml визначено HEALTHCHECK, який опитує ендпойнт Spring Boot Actuator /actuator/health. Це дозволяє оркестратору автоматично визначати готовність сервісу до приймання трафіку й перезавантажувати контейнер у разі деградації або зависання. Експонування порту 8081 узгоджується з порт-бриджингом у Compose-файлі, де сервіс публікується на хості під тими самими номерами портів для спрощення локального тестування.

Оркестрація розгортання виконується за допомогою docker-compose версії 3.8. У складі сервісу ai-face-authentication явно фіксується платформа linux/amd64,

що усуває неоднозначність у середовищах з кількома архітектурами (наприклад, на Apple Silicon із емуляцією). Профіль Spring Boot визначається змінною `SPRING_PROFILES_ACTIVE=dev`, що активує відповідні конфігурації зберігання, логування й безпеки для розробницьких стендів. Додаткові параметри середовища прокидаються через `JAVA_OPTS`, завдяки чому управління налаштуваннями JVM централізоване та не вимагає перекомпіляції. Монтування тому `./src/main/resources/etc:/app/etc:ro` забезпечує детерміноване підкладання файлів каскадів і модельних артефактів у контейнері з правами лише для читання, що підвищує безпекову ізоляцію і спрощує оновлення без перебудови образу. Параметри `healthcheck` у Compose дублюють перевірку з Dockerfile, але дозволяють гнучко налаштовувати інтервал, таймаут і період стартової затримки відповідно до характеристик цільової інфраструктури.

Послідовність запуску зводиться до стандартних кроків: збірка багатостадійного образу з використанням `docker compose build`, підняття сервісу `docker compose up -d`, очікування переходу стану здоров'я в `healthy` за логами або через `docker inspect`, після чого сервіс доступний на `http://localhost:8081`. Завдяки відокремленню стадій збірки та рантайму, фіксації платформи, роботі під некореневим користувачем і вбудованим перевіркам стану, процес розгортання є відтворюваним, безпечним і придатним до перенесення між локальним середовищем розробника та серверними інстансами. Таке компонування контейнера також спрощує подальше CI/CD: у конвеєрі достатньо виконати кроки складання образу, завантаження в реєстр і запуску на цільовому середовищі, не вносячи змін до прикладного коду чи конфігурацій Spring Boot.

3.3 Розгортання фронтенд частини застосунку

Фронтенд частина системи реалізована у вигляді нативного мобільного застосунку для операційної системи iOS, створеного мовою програмування Swift з використанням фреймворку SwiftUI. Розгортання iOS-застосунку має свою специфіку, оскільки потребує використання екосистеми Apple Developer, яка

забезпечує контроль за підписом, сертифікацією та розповсюдженням програмного забезпечення.

Процес починається із збирання проєкту в середовищі Xcode, яке є офіційним інструментом для розробки iOS-додатків. Після завершення етапів компіляції та перевірки застосунків трансформується у виконуваний пакет формату .ipa (iOS App Archive), який містить усі необхідні ресурси, залежності та цифрові сертифікати. Для цього Xcode використовує інструмент Xcodebuild і систему керування залежностями Swift Package Manager, яка автоматично завантажує зовнішні бібліотеки (наприклад, Combine, Firebase або локальні AI SDK, якщо вони використовуються).

Наступним етапом є цифрове підписування застосунку. Кожен iOS-додаток повинен бути підписаний дійсним сертифікатом розробника, виданим Apple, та пов'язаним із Provisioning Profile. Цей профіль містить інформацію про Bundle ID, цільові пристрої (якщо це тестова збірка) і права доступу до функцій системи — наприклад, до камери, Face ID, хмарного сховища або біометричних API. Без належного підпису застосунків не може бути запущений на фізичних пристроях або розповсюджений через App Store.

Для локального тестування використовуються сервіси TestFlight або безпосереднє встановлення на підключений iPhone/iPad через Xcode. TestFlight — офіційна платформа Apple для бета-тестування, яка дозволяє розробнику поширювати збірки серед тестувальників через запрошення або публічні посилання. Це дає змогу виконувати контроль якості, збирати зворотний зв'язок і виявляти можливі помилки ще до публічного релізу.

У разі публікації застосунку у App Store розгортання здійснюється через App Store Connect, де розробник завантажує підписаний .ipa-файл, вказує метадані (опис, категорію, рейтинг контенту, ключові слова, зображення), після чого застосунків проходить автоматичну та ручну перевірку з боку Apple. Після затвердження користувачі можуть завантажити додаток безпосередньо через магазин.

У випадку внутрішнього корпоративного розгортання використовується Apple Developer Enterprise Program, що дозволяє поширювати застосунок всередині організації без публікації в App Store. Такий підхід актуальний для тестування або внутрішнього користування.

Під час виконання на пристрої iOS застосунок використовує власне контейнеризоване середовище, яке ізолює його дані від системи. Усі файли, нотатки та біометричні шаблони зберігаються у локальному сховищі App Sandbox, що захищає інформацію користувача. Для інтеграції з бекендом застосунок встановлює HTTPS-з'єднання із сервером Spring Boot, автентифікуючись через JWT-токен.

Перед публікацією застосунок проходить процедуру оптимізації збірки. Виконується зменшення розміру пакета за допомогою Swift compiler optimizations, очищення невикористаних ресурсів (Assets trimming) і ввімкнення Bitcode, який дає Apple змогу автоматично перекомпілювати застосунок для різних архітектур (наприклад, ARM64). Це забезпечує стабільну роботу додатка на різних моделях iPhone та iPad.

Отже, процес розгортання фронтенд-частини охоплює повний цикл — від компіляції в Xcode та підпису сертифікатом розробника до тестування через TestFlight і публікації в App Store. Така процедура гарантує відповідність стандартам безпеки Apple, стабільність роботи застосунку та зручність користувачів при його встановленні та оновленні. На рисунку 3.1 зображено повний цикл розгортання бекенд та фронтенд частин

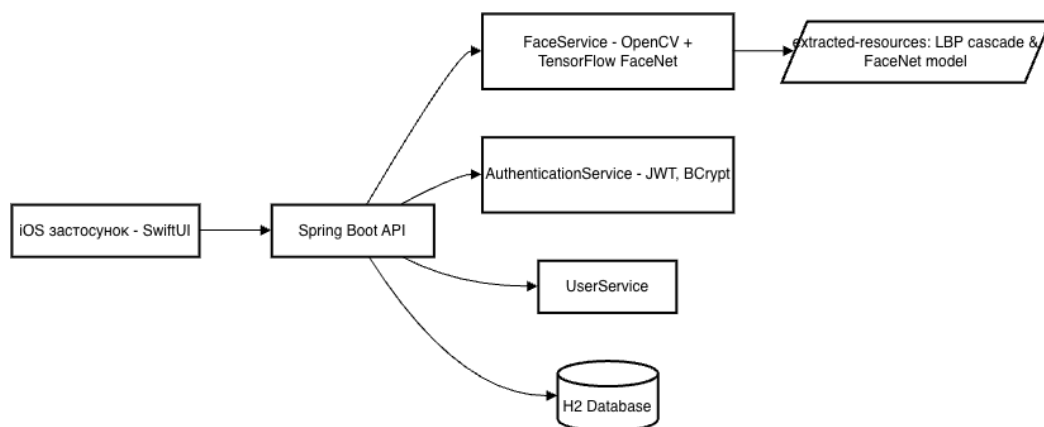


Рис 3.1 – Діаграма розгортання застосунку

На діаграмі показано, що користувач взаємодіє з мобільним додатком на iOS, який реалізує інтерфейс користувача засобами SwiftUI та забезпечує автентифікацію через Face ID або відправку зображення обличчя на сервер. Комунікація із сервером здійснюється через захищене HTTPS-з'єднання з використанням JWT-токенів для підтвердження автентичності.

На стороні сервера, розгорнутого у Docker-контейнері, функціонує Spring Boot-додаток, який містить сервіси:

AuthenticationService — відповідає за автентифікацію, генерацію токенів та обробку запитів логіну/реєстрації.

FaceService — реалізує детекцію обличчя за допомогою OpenCV та створення ембеддингів через TensorFlow FaceNet.

UserService — виконує управління користувачами та доступ до бази даних.

У середині контейнера розміщена H2 Database, що зберігає облікові записи, ембеддинги облич і токени. Модельні файли (lbpcascade_frontalface_improved.xml та 20180402-114759.pb) розпаковуюються у директорію /app/extracted-resources.

Таким чином, діаграма демонструє двошарову архітектуру: мобільний клієнт (SwiftUI) ↔ сервер (Spring Boot + Docker + TensorFlow), що взаємодіють через стандартизований REST API з використанням безпечних механізмів автентифікації.

3.4 Робота з застосунком

Робота з мобільним застосунком починається з головного екрана, який відображає перелік створених користувачем нотаток. На цьому етапі користувачу надається можливість переглядати наявні записи, а також переходити до створення або редагування нотаток. У списку відображаються основні атрибути кожної нотатки: заголовок, фрагмент тексту та дата останнього редагування. Такий підхід забезпечує швидку навігацію та зручний доступ до інформації. На рисунку 3.1 зображено екран перегляду списку нотаток користувача.

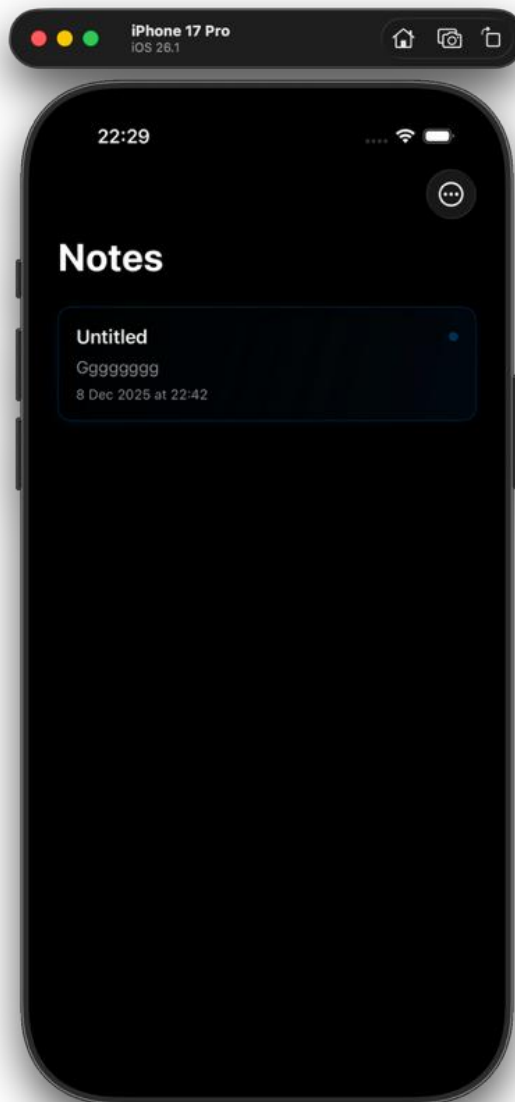


Рис 3.1 - Екран перегляду списку нотаток користувача.

Під час вибору окремої нотатки здійснюється перехід до режиму редагування. На відповідному екрані користувач може змінювати заголовок і основний текст нотатки, користуватися елементами форматування (напівжирний текст, курсив, підкреслення, зміна розміру шрифту), а також зберігати внесені зміни. Інтерфейс редагування реалізований з урахуванням принципів мінімалізму й ергономіки, що сприяє комфортній роботі з текстом навіть на мобільних пристроях із невеликим екраном.

На рисунку 3.2 зображено екран редагування нотатки з панеллю форматування тексту.

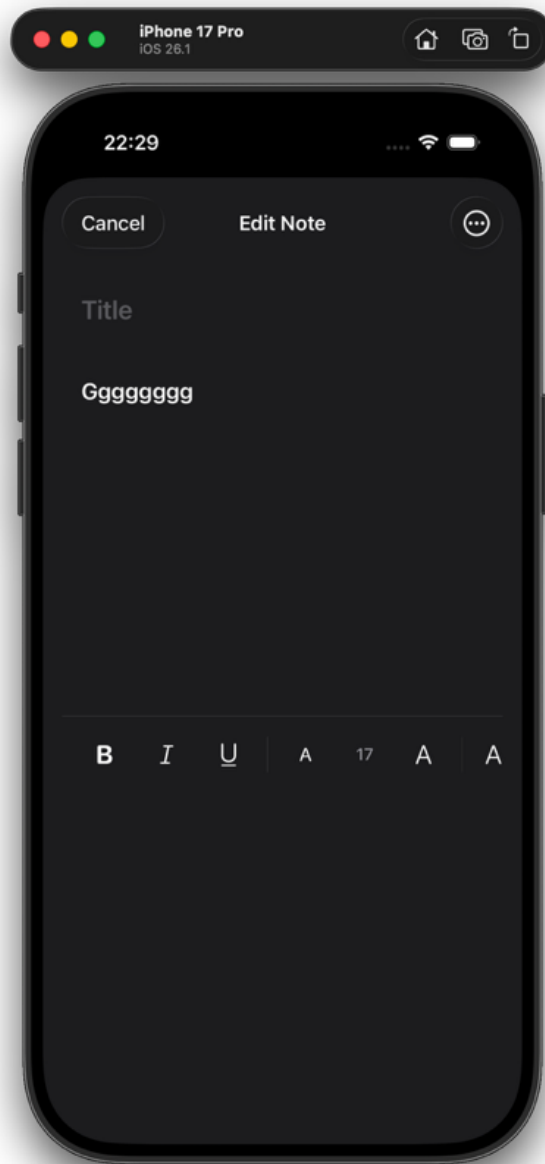


Рис 3.2 - Екран редагування нотатки з панеллю форматування тексту.

Доступ до функціональності застосунку можливий лише після успішної автентифікації користувача. Для цього реалізовано екран авторизації, який підтримує кілька режимів входу: класичну авторизацію за електронною поштою та паролем, реєстрацію нового користувача, а також вхід за допомогою біометричної аутентифікації на основі розпізнавання обличчя. Користувач може обрати відповідний режим, перемикаючись між вкладками інтерфейсу. На рисунку 3.3 зображено стартовий екран автентифікації з вибором режиму входу до системи.

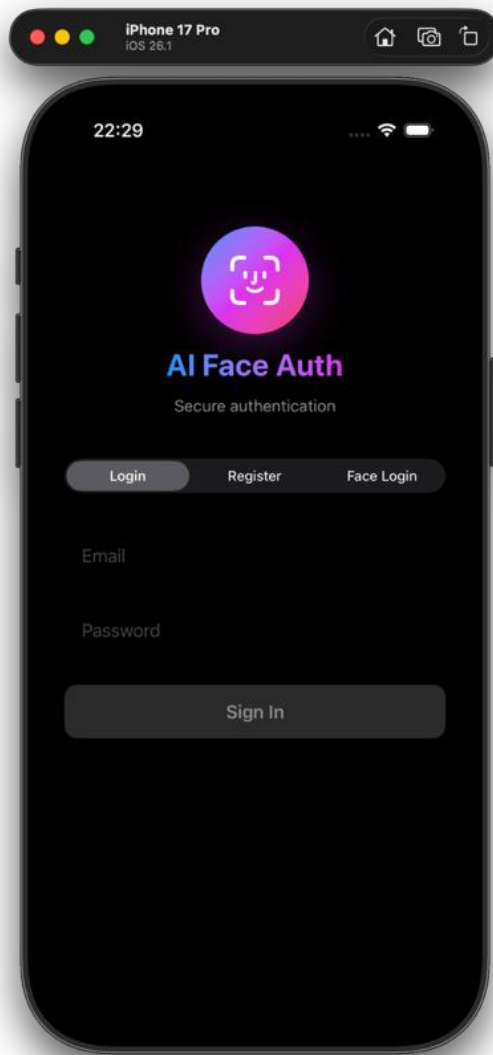


Рис 3.3 - Стартовий екран автентифікації з вибором режиму входу до системи.

У процесі реєстрації нового користувача передбачено введення персональних даних (ім'я, прізвище, електронна адреса, пароль і підтвердження пароля), а також етап захоплення зображення обличчя за допомогою камери мобільного пристрою. Отримане зображення передається до серверної частини застосунку для формування біометричного ембеддингу, який надалі використовується для автентифікації. Такий підхід дозволяє прив'язати обліковий запис користувача до його біометричних характеристик. На рисунку 3.4 зображено екран реєстрації користувача з можливістю захоплення фотографії обличчя.

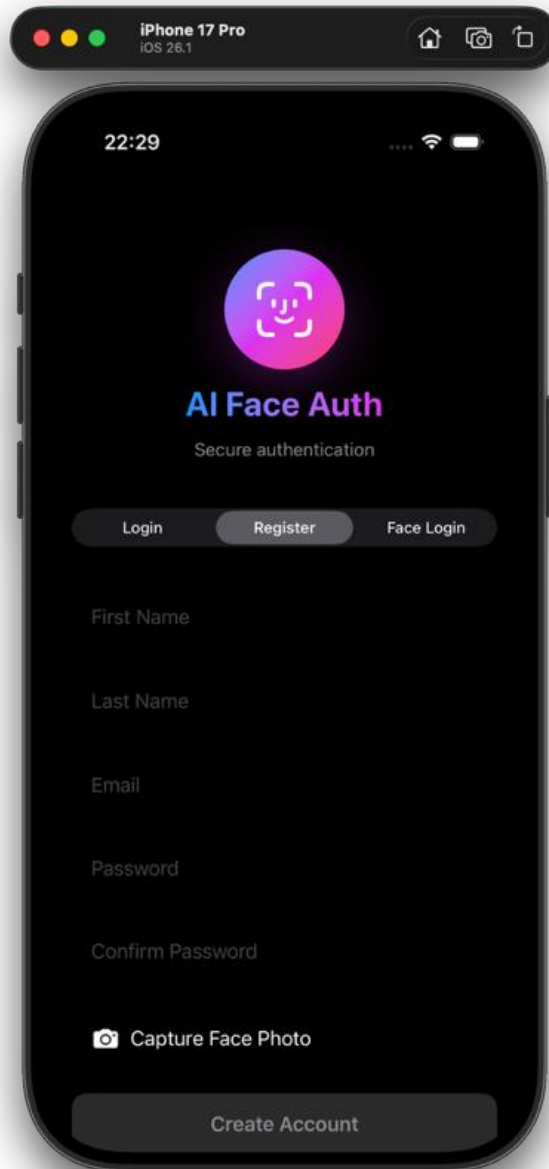


Рис 3.4 - Екран біометричної реєстрації з використанням розпізнавання

Окремий режим роботи застосунку присвячений біометричному входу. У цьому випадку користувач активує функцію розпізнавання обличчя, після чого камера фіксує зображення, яке порівнюється з раніше збереженими біометричними даними. У разі досягнення заданого порогового значення подібності користувач отримує доступ до своїх нотаток без необхідності введення пароля. Це суттєво підвищує зручність використання застосунку та рівень безпеки доступу до персональної інформації. На рисунку 3.5 зображено екран біометричної автентифікації з використанням розпізнавання обличчя.

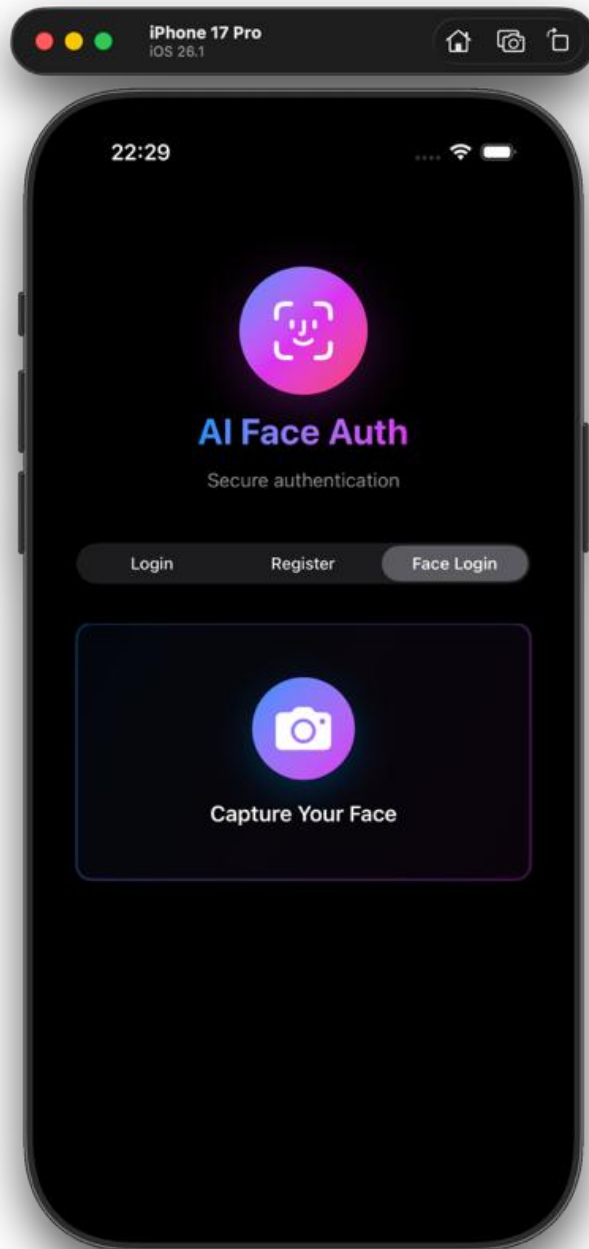


Рис 3.5 - Екран біометричної автентифікації з використанням розпізнавання обличчя.

Таким чином, розроблений мобільний застосунок забезпечує повний цикл роботи користувача — від безпечної автентифікації до створення, редагування та зберігання нотаток. Поєднання класичних механізмів доступу та біометричної автентифікації дозволяє досягти балансу між зручністю використання та високим рівнем захисту персональних даних.

Висновки до третього розділу

Проведене тестування підтвердило правильність реалізації ключових функцій системи та відповідність розробленого програмного забезпечення функціональним вимогам. Усі модульні тести пройшли успішно, що свідчить про стабільність компонентів аутентифікації, безпеки, управління користувачами та розпізнавання облич. Інтеграційні перевірки довели коректність взаємодії між REST-контролерами, сервісами та рівнем доступу до даних. Реалізоване контейнерне розгортання бекенд-частини за допомогою Docker забезпечує відтворюваність середовища, ізоляцію залежностей та спрощене масштабування. Фронтенд-застосунок, розроблений для iOS на основі SwiftUI, успішно проходить усі етапи підпису, тестування і публікації в екосистемі Apple, що гарантує безпечне використання технологій Face ID і стабільну інтеграцію з сервером. Таким чином, створений застосунок є повністю готовим до практичного впровадження, забезпечуючи користувачам високий рівень безпеки, продуктивності та зручності у користуванні.

ВИСНОВКИ

У результаті виконання наукової практики було досягнуто поставленої мети – розроблено прототип мобільного застосунку для створення та безпечного зберігання нотаток із використанням технології біометричної аутентифікації на основі розпізнавання обличчя. Дослідження поєднало теоретичні та практичні аспекти сучасних підходів до забезпечення інформаційної безпеки, побудови архітектурних рішень і впровадження алгоритмів штучного інтелекту в мобільні системи. Проведений аналіз показав, що існуючі застосунки для нотаток не забезпечують комплексного підходу до захисту персональних даних, тоді як використання розпізнавання обличчя дає змогу створити зручний і водночас надійний механізм доступу до інформації.

Розроблена архітектура системи на основі Spring Boot і SwiftUI продемонструвала ефективність поєднання серверних і клієнтських технологій для досягнення високого рівня продуктивності, безпеки та масштабованості. Використання бібліотек OpenCV і TensorFlow забезпечило інтеграцію алгоритмів машинного навчання у процес обробки зображень, що дозволило реалізувати біометричну автентифікацію з високим рівнем точності. На рівні мобільного клієнта SwiftUI та архітектура MVVM створили гнучку модель інтерфейсу, здатну до реактивного оновлення станів і безпечної взаємодії із сервером через RESTful API.

Практична реалізація системи підтвердила її функціональну завершеність і відповідність вимогам інформаційної безпеки. Було реалізовано основні сценарії користування – реєстрацію, автентифікацію за обличчям, створення, редагування, перегляд і пошук нотаток, а також синхронізацію між пристроями. Особливу увагу приділено організації бази даних, збереженню ембеддингів у зашифрованому вигляді, використанню JWT-токенів і для контролю доступу.

Наукова новизна полягає у поєднанні класичних архітектурних підходів із сучасними технологіями біометрії, що створює передумови для розвитку нових типів інтелектуальних мобільних застосунків. Отримані результати мають значну

практичну цінність і можуть бути використані у процесі підготовки фахівців з інженерії програмного забезпечення, а також у подальших дослідженнях, спрямованих на вдосконалення систем автентифікації, зокрема у сфері фінансових, освітніх та корпоративних сервісів.

Таким чином, під час проходження наукової практики здобуто комплексні знання та практичні навички з проєктування, програмної реалізації, тестування й розгортання мобільних систем із підвищеними вимогами до безпеки, що підтверджує успішність виконання поставлених завдань і досягнення мети дослідження

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Biometric Security: Enhancing Mobile App Protection Nordstone. URL: <https://nordstone.co.uk/blog/integrating-biometricsecurity-features-into-mobile-apps> (дата звернення: 05.03.2025).
2. How to Create a REST API using Java Spring Boot GeeksforGeeks. URL: <https://www.geeksforgeeks.org/java/how-to-create-a-rest-api-using-java-spring-boot/> (дата звернення: 11.04.2025).
3. What is OpenCV Library – GeeksforGeeks. URL: <https://www.geeksforgeeks.org/computer-vision/opencv-overview/> (дата звернення: 22.05.2025).
4. Secure Mobile App | Biometric Authentication Addicta. URL: <https://addictaco.com/secure-your-business-app-biometric-authentication-in-mobile-apps/> (дата звернення: 28.06.2025).
5. Building REST APIs in Java with Spring Boot Foojay. URL: <https://foojay.io/today/building-rest-apis-in-java-with-spring-boot/> (дата звернення: 14.03.2025).
6. Haar Cascades vs. LBP Cascades in Face Detection StackOverflow. URL: <https://stackoverflow.com/questions/8791178/haar-cascades-vs-lbp-cascades-in-face-detection> (дата звернення: 19.09.2025).
7. How to Test Biometric Authentication on Mobile Apps Dev.to. URL: <https://dev.to/misterankit/how-to-test-biometric-authentication-on-mobile-apps-1o1p> (дата звернення: 03.07.2025).
8. How to Build a REST API with Spring Boot Camunda. URL: <https://camunda.com/blog/2025/05/how-to-build-a-rest-api-with-spring-boot-a-step-by-step-guide/> (дата звернення: 27.04.2025).
9. Face detection using cascade classifier Scikit-image. URL: https://scikit-image.org/docs/0.25.x/auto_examples/applications/plot_face_detection.html (дата звернення: 15.05.2025).

10. The Next Step in Mobile App Security LinkedIn. URL: <https://www.linkedin.com/pulse/rise-biometric-authentication-next-step-mobile-app-security-op0vc> (дата звернення: 09.09.2025).
11. FaceNet: A Unified Embedding for Face Recognition arXiv. URL: <https://arxiv.org/abs/1503.03832> (дата звернення: 11.02.2025).
12. JSON Web Token (JWT) Authentication Tyk. URL: <https://tyk.io/docs/basic-config-and-security/security/authentication-authorization/json-web-tokens> (дата звернення: 06.06.2025).
13. How to Containerize an Application Using Docker DataCamp. URL: <https://www.datacamp.com/tutorial/how-to-containerize-application-using-docker> (дата звернення: 22.03.2025).
14. Face Recognition using FaceNet – GitHub Repository GitHub. URL: <https://github.com/rehanfazalkhan/Face-Recognition-using-FaceNET-Deep-learning-Opencv> (дата звернення: 17.04.2025).
15. Using JSON Web Tokens for API Authentication Trebllle. URL: <https://trebllle.com/blog/using-json-web-tokens-for-api-authentication-beginners-guide> (дата звернення: 10.07.2025).
16. Containerize an Application Docker Docs. URL: https://docs.docker.com/get-started/workshop/02_our_app/ (дата звернення: 08.02.2025).
17. How to Build a Face Recognition System Using FaceNet in Python Dev.to. URL: <https://dev.to/abhinowww/how-to-build-a-face-recognition-system-using-facenet-in-python-27kh> (дата звернення: 12.05.2025).
18. JWT Authentication API Platform. URL: <https://api-platform.com/docs/core/jwt/> (дата звернення: 16.10.2025).
19. FaceNet – Using Facial Recognition System GeeksforGeeks. URL: <https://www.geeksforgeeks.org/machine-learning/facenet-using-facial-recognition-system/> (дата звернення: 04.11.2025).

20. Modern MVVM iOS App Architecture with Combine and SwiftUI Vadim Bulavin. URL: <https://www.vadimbulavin.com/modern-mvvm-ios-app-architecture-with-combine-and-swiftui/> (дата звернення: 13.06.2025).
21. Spring Security: Authentication and Authorization In-Depth Marco Behler. URL: <https://www.marcobehler.com/guides/spring-security> (дата звернення: 21.07.2025).
22. Training a Deep Network in TensorFlow SuperML. URL: <https://superml.org/tutorials/bdl-training-deep-network-tensorflow> (дата звернення: 09.02.2025).
23. Introducing MVVM into your SwiftUI project Hackingwithswift. URL: <https://www.hackingwithswift.com/books/ios-swiftui/introducing-mvvm-into-your-swiftui-project> (дата звернення: 26.09.2025).
24. Mastering Authorization and Authentication with Spring Security LinkedIn. URL: <https://www.linkedin.com/pulse/mastering-authorization-authentication-springsecurity-harsh-mishra-dpqwc> (дата звернення: 10.03.2025).
25. Introduction to TensorFlow GeeksforGeeks. URL: <https://www.geeksforgeeks.org/python/introduction-to-tensorflow/> (дата звернення: 18.05.2025).
26. MVVM in SwiftUI Explained: Build Scalable & Testable Apps Swift-pal. URL: <https://swift-pal.com/mvvm-in-swiftui-explained-build-scalable-testable-apps-with-clean-architecture-2b36443ebfca> (дата звернення: 24.10.2025).
27. Spring Security Authentication DZone. URL: <https://dzone.com/articles/spring-security-authentication> (дата звернення: 30.04.2025).
28. TensorFlow Google Open Source. URL: <https://opensource.google/projects/tensorflow> (дата звернення: 11.11.2025).

29. Understanding MVVM Architecture with SwiftUI Dev.to. URL: https://dev.to/tech_tales_daa8a7eab515b3/understanding-mvvm-architecture-with-swiftui-4380 (дата звернення: 03.03.2025).
30. How to map JPA/Hibernate entities on the same database table StackOverflow. URL: <https://stackoverflow.com/questions/29007676/how-to-map-two-jpa-or-hibernate-entities-on-the-same-database-table> (дата звернення: 06.05.2025).
31. Learn to Protect Passwords with Bcrypt Hash in a Few Minutes Hashnode. URL: <https://dev.to/stephenodogwu/learn-to-protect-passwords-with-bcrypt-hash-in-a-few-minutes-c82> (дата звернення: 17.08.2025).
32. REST API – Why use PUT DELETE POST GET StackOverflow. URL: <https://stackoverflow.com/questions/4573305/rest-api-why-use-put-delete-post-get> (дата звернення: 19.07.2025).
33. Hibernate: Mapping Entities Hibernate Docs. URL: <https://docs.hibernate.org/annotations/3.5/reference/en/html/entity.html> (дата звернення: 08.06.2025).
34. The Bcrypt Algorithm for Secure Password Hashing Hashnode. URL: <https://itsvinayak.hashnode.dev/the-bcrypt-algorithm-for-secure-password-hashing> (дата звернення: 04.02.2025).
35. Registration with Spring Security – Password Encoding with BCrypt Baeldung. URL: <https://www.baeldung.com/spring-security-registration-password-encoding-bcrypt> (дата звернення: 29.03.2025).
36. HTTP Methods in APIs: GET, POST, PUT, DELETE Explained API7. URL: <https://api7.ai/learning-center/api-101/http-methods-in-apis> (дата звернення: 01.08.2025).
37. Hibernate ORM – Your relational data. Objectively. Hibernate.org. URL: <https://hibernate.org/orm/> (дата звернення: 16.11.2025).
38. Triplet Loss: Intro, Implementation, Use Cases V7Labs. URL: <https://www.v7labs.com/blog/triplet-loss> (дата звернення: 14.04.2025).

39. Implementing Face Detection Using Haar Cascades and AdaBoost Algorithm PacktPub. URL: <https://www.packtpub.com/de-br/learning/how-to-tutorials/implementing-face-detection-using-haar-cascades-adaboost-algorithm?srsltid=AfmBOoq7H11rq4hX9ou6vOmDU2JM73tUJIvIHU3ZVJUfSBSroaWeWkJv> (дата звернення: 22.02.2025).
40. Facial Recognition with Eigenfaces and Fisherfaces GitHub Pages. URL: [https://anyacakes.github.io/images/facial-recognition-eigenfaces%20\(1\).pdf](https://anyacakes.github.io/images/facial-recognition-eigenfaces%20(1).pdf) (дата звернення: 30.06.2025).
41. Triplet Loss – Illustration of triplet loss. URL: https://www.researchgate.net/figure/Illustration-of-triplet-loss-N-pair-loss-and-the-proposed-ranking-loss-For-triplet-loss_fig2_334997522 (дата звернення: 12.07.2025).
42. Face Recognition and Detection Using Eigenface Algorithm SSE Capstone Repository. URL: <https://cdn.aui.ma/sse-capstone-repository/pdf/spring-2023/Face%20Recognitionand%20Detection%20Using%20Eigenface%20Algorithm%20Final%20Report.pdf> (дата звернення: 09.09.2025).
43. Triplet Loss Definition Encord. URL: <https://encord.com/glossary/triplet-loss-definition/> (дата звернення: 03.02.2025).

ДОДАТКИ

Код перевірки JWT токена

```
package com.example.aifaceauthentication.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.nd4j.common.io.StringUtils;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private static final String AUTH_HEADER = "Authorization";
    private static final String TOKEN_START = "Bearer ";
```

```

private final JwtTokenProvider jwtTokenProvider;
private final UserDetailsService userDetailsService;

@Override
protected void doFilterInternal(
    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain)
    throws ServletException, IOException {
    String token = getToken(request);

    if (token != null && jwtTokenProvider.isTokenValid(token)) {
        String email = jwtTokenProvider.getEmail(token);
        UserDetails userDetails =
            userDetailsService.loadUserByUsername(email);
        Authentication auth = new UsernamePasswordAuthenticationToken(
            userDetails, null, userDetails.getAuthorities()
        );
        SecurityContextHolder.getContext().setAuthentication(auth);
    }

    filterChain.doFilter(request, response);
}

private String getToken(HttpServletRequest request) {
    String bearerToken = request.getHeader(AUTH_HEADER);
    if (StringUtils.hasText(bearerToken) &&
        bearerToken.startsWith(TOKEN_START)) {
        return bearerToken.substring(TOKEN_START.length());
    }
}

```

```
}
```

```
return null;
```

```
}
```

```
}
```

Код автентифікації користувача за допомогою фотографії

```
package com.example.aifaceauthentication.controller;

import com.example.aifaceauthentication.dto.user.UserLoginRequestDto;
import com.example.aifaceauthentication.dto.user.UserLoginResponseDto;
import com.example.aifaceauthentication.dto.user.UserRegisterRequestDto;
import com.example.aifaceauthentication.dto.user.UserResponseDto;
import com.example.aifaceauthentication.service.AuthenticationService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.IOException;

@RestController
@RequiredArgsConstructor
@RequestMapping("/auth")
public class UserController {

    private final AuthenticationService authenticationService;

    @PostMapping("/register")
    public ResponseEntity<UserResponseDto> register(
```

```

        @RequestParam("user") @Validated UserRegisterRequestDto requestDto,
        @RequestParam("photo") MultipartFile photo) throws IOException {
            UserResponseDto userResponseDto =
authenticationService.register(requestDto, photo);
            return ResponseEntity.ok(userResponseDto);
        }

```

```

        @PostMapping("/login")
        public ResponseEntity<UserLoginResponseDto> authenticate(@Validated
        @RequestBody UserLoginRequestDto requestDto) {
            UserLoginResponseDto userLoginResponseDto =
authenticationService.login(requestDto);
            return ResponseEntity.ok(userLoginResponseDto);
        }

```

```

        @PostMapping("/login/photo")
        public ResponseEntity<UserLoginResponseDto>
authenticateByPhoto(@RequestParam("photo") MultipartFile photo) {
            UserLoginResponseDto userLoginResponseDto =
authenticationService.loginByPhoto(photo);
            return ResponseEntity.ok()

```

```

                .contentType(org.springframework.http.MediaType.APPLICATION_JSON)
                .body(userLoginResponseDto);
        }

```

```

        @GetMapping("/info")
        public ResponseEntity<UserResponseDto> getUserDetails() {

```

```
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

        String email = authentication.getName();

        UserResponseDto userResponseDto =
authenticationService.getUserByEmail(email);

        return ResponseEntity.ok(userResponseDto);
    }
}
```