ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка мобільного застосунку для полегшення підбору одягу в залежності від погоди та настрою»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Михайло СІКАЛЮК
(підпис)

Виконав: здобувач вищої освіти групи ПД-44
Михайло СІКАЛЮК

Керівник: Микита ТРЕНЬОВ
асистент кафедри
Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Сікалюку Михайлу Васильовичу

1. Тема кваліфікаційної роботи: «Розробка мобільного застосунку для полегшення підбору одягу в залежності від погоди та настрою»
керівник кваліфікаційної роботи асистент кафедри Микита ТРЕНЬОВ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: Теоретичні відомості про мобільну розробку у середовищі Flutter, особливості побудови інтерфейсів та роботи з базами даних. Опис методів інтеграції зовнішніх API (погодного, рекомендаційного, обробки зображень), формування структури даних для збереження галереї одягу та образів. Технічна документація з використанням Firebase, Cloudinary та інших сервісів, що забезпечують роботу застосунку.

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

1. Огляд та аналіз існуючих мобільних застосунків і технологій, що використовуються для персоналізації образів та адаптації до погодних умов.

2. Проєктування мобільного застосунку для підбору одягу відповідно до погоди та настрою користувача.
3. Програмна реалізація та опис функціонування застосунку WeatherMood з інтеграцією Firebase, OpenWeather API, Cloudinary та рекомендаційної системи.
4. Тестування функціональності застосунку.

5. Перелік графічного матеріалу: *презентація*
1. Вимоги до програмного забезпечення.
2. Аналіз аналогів.
3. Програмні засоби реалізації.
4. Перелік функцій програми
5. Діаграма класів.
6. Діаграма прецендентів
7. Діаграма активностей
8. Діаграма послідовності
9. Екранні форми.
10. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд існуючих рішень для персоналізації одягу, обліку погоди та настрою користувача	14.03-17.03.2025	
4	Проєктування архітектури мобільного застосунку WeatherMood	18.03-19.04.2025	
5	Тестування застосунку WeatherMood: логіка, інтерфейс, інтеграція сервісів (Firebase, API)	20.04-28.04.2025	
6	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
7	Розробка демонстраційних матеріалів	12.05-18.05.2025	
8	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Михайло СІКАЛЮК
(підпис)

Керівник
кваліфікаційної роботи _____ Микита ТРЕНЬОВ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: __58_ стор., __0__ табл., __12_ рис., _20_ джерел.

Мета роботи — підвищення зручності користувачів у виборі щоденного одягу за допомогою мобільного застосунку, який враховує погодні умови, настрій і вміст гардеробу.

Об'єкт розробки — процес підбору одягу з урахуванням змінного настрою та погодних умов.

Предмет розробки — мобільний застосунок для формування персоналізованого образу на день на основі аналізу погоди, настрою користувача та наявних речей.

Короткий зміст роботи: Розроблено мобільний застосунок, що автоматизує процес підбору одягу з урахуванням погодних умов, настрою користувача та вмісту гардеробу. Програмно реалізовані основні функції системи, зокрема: авторизація, додавання фотографій речей із видаленням фону, визначення домінантного кольору, класифікація одягу за категоріями, формування образу на день із поясненням вибору. Застосунок інтегрує штучний інтелект для персоналізованих рекомендацій, а також API для визначення погоди та обробки зображень. У роботі використано Flutter для розробки інтерфейсу, Dart як основну мову програмування, Firebase для авторизації та зберігання даних, OpenWeather API для отримання погодних умов, Remove.bg і Cloudinary для обробки та збереження зображень.

Сферою використання застосунку є підтримка щоденного вибору вбрання користувачами, які прагнуть виглядати доречно відповідно до настрою та погоди, із можливістю редагування та пояснення кожного образу.

КЛЮЧОВІ СЛОВА: МОБІЛЬНИЙ ЗАСТОСУНОК, ПІДБІР ОДЯГУ, FLUTTER, FIREBASE, ПОГОДА, НАСТРІЙ, ГАРДЕРОБ.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми та її актуальність. У наш час, коли доступ до інформації та сервісів через мобільні застосунки став звичним елементом повсякденного життя, питання швидкого та зручного підбору одягу набуло особливого значення. Темп сучасного життя, змінні погодні умови та різноманіття речей у гардеробі створюють щоденні труднощі у виборі образу. Часто користувач не має часу або натхнення для того, щоб ретельно обирати одяг відповідно до настрою чи зовнішніх умов. У таких випадках було б доцільно мати персонального цифрового помічника, який врахує всі ці фактори та запропонує готове рішення. Саме з цією метою був створений мобільний застосунок, розроблений у межах цієї дипломної роботи.

Використання штучного інтелекту дозволяє автоматизувати процес аналізу вмісту гардеробу, визначати погодні умови, враховувати настрій користувача та пропонувати релевантні комбінації одягу. Такий підхід підвищує ефективність, економить час та усуває щоденні труднощі, пов’язані з вибором вбрання.

Ступінь вивчення проблеми. На сьогодні існує чимало сайтів і мобільних сервісів, які дозволяють користувачам вести облік власного гардеробу, створювати образи або зберігати улюблені луки. Проте більшість із них обмежується ручною фіксацією речей без врахування зовнішніх умов або емоційного стану. Типовий застосунок працює як візуальна шафа — зручно, але без інтелектуальної підтримки. Лише поодинокі сервіси намагаються враховувати погоду, і майже жоден не пропонує інтеграцію настрою користувача у процес вибору образу. Крім того, наявність реклами, складні інтерфейси або платний доступ обмежують функціональність таких сервісів.

Водночас користувачі прагнуть мінімізувати час на рутинні завдання, мати просте та адаптивне рішення, яке допомагає виглядати доречно за будь-яких умов. Це створює запит на новий тип мобільного застосунку — персонального стиліста, який працює у режимі реального часу.

Об'єкт розробки — процес підбору одягу в умовах змінного настрою та погодних умов.

Предмет розробки — мобільний застосунок, який формує персоналізований образ, базуючись на погоді, настрої та наявному гардеробі користувача.

Мета роботи — підвищення зручності користувачів у виборі щоденного одягу за допомогою інтелектуального мобільного інструменту, що поєднує інформацію про погоду, настрій та стильові уподобання.

Методика розробки включає елементи UI/UX-дизайну, використання Flutter як фреймворку для створення мобільного застосунку, застосування Firebase для авторизації та зберігання даних, інтеграцію з погодним API, а також генерацію рекомендацій з поясненням вибору за допомогою штучного інтелекту.

На основі аналізу потреб користувачів і технічних можливостей було обрано шлях створення простого, адаптивного, інтуїтивного застосунку, який не потребує багато ресурсів та здатен швидко формувати образи з урахуванням актуальних параметрів.

Враховуючи завдання дипломної роботи, найбільш оптимальним рішенням стало використання штучного інтелекту, здатного визначати зв'язки між кольором, стилем, настроєм та погодою, і на цій основі пропонувати логічно обґрунтовані варіанти одягу.

Наукова новизна полягає у поєднанні кількох чинників — настрою, погодних умов, вмісту гардеробу та стилістичних уподобань — для генерації щоденного образу. Застосунок використовує простий інтерфейс, не містить рекламних модулів та забезпечує логічне пояснення вибору кожної речі.

Практична значущість полягає у створенні готового рішення, яке може бути використане широким колом користувачів. Застосунок допомагає організувати гардероб, формувати стиль, зменшити кількість непотрібних покупок і впевнено почуватися у будь-якій ситуації. Реалізація виконана на мові програмування Dart з використанням фреймворку Flutter та хмарної платформи Firebase, що забезпечує високу швидкість, масштабованість і стабільність роботи програм.

1 АНАЛІЗ ТА ХАРАКТЕРИСТИКА МОБІЛЬНИХ ЗАСТОСУНКІВ

1.1 Історія створення мобільних застосунків та їх характеристика

Поява мобільних застосунків — це один із найважливіших етапів розвитку цифрових технологій. Їх історія нерозривно пов’язана з еволюцією мобільних телефонів, а згодом і смартфонів. Ще у 1990-х роках з’явились перші програмні продукти, які можна умовно вважати прообразами сучасних застосунків. У ті часи вони були елементарними — календарі, калькулятори, будильники, прості ігри на кшталт "Змійки", вбудовані у телефони компаній Nokia, Siemens, Ericsson.

З переходом від кнопкових телефонів до смартфонів змінилася й сама концепція застосунків. Якщо раніше програми були заздалегідь встановлені виробником і не підлягали зміні, то з розвитком мобільних операційних систем з’явилась можливість встановлення стороннього програмного забезпечення. Одним із перших проривів у цьому напрямку стало створення платформи Symbian… У 2008 році був запущений App Store… На відміну від закритої моделі Apple, Android надавав більшу свободу розробникам.

Справжній прорив стався у 2007 році з презентацією першого iPhone. Компанія Apple не лише запропонувала новий формат пристрою з сенсорним екраном, а й заклала підґрунтя для екосистеми мобільних застосунків. У 2008 році був запущений **App Store** — онлайн-магазин, де користувачі могли легко завантажувати програми сторонніх розробників. Це стало початком нової ери в індустрії. Вже за перший рік існування App Store у ньому було представлено понад 50 000 застосунків. Розробка програм стала доступною для незалежних студій і навіть окремих ентузіастів [□1].

У відповідь на успіх Apple, компанія Google запустила свою платформу — **Android Market**, яка згодом перетворилась у **Google Play**. На відміну від закритої моделі Apple, Android надавав більшу свободу розробникам, що сприяло швидкому зростанню кількості застосунків. На початку 2010-х років мобільні застосунки

активно розвивались у таких категоріях, як комунікації (Viber, WhatsApp), розваги (Instagram, YouTube), навігація (Google Maps), банкінг, покупки, замовлення послуг [□2].

Відтоді мобільні застосунки стали важливою частиною повсякденного життя. Їх кількість невпинно зростала, і вже у 2020-х роках більшість бізнесів мають мобільні застосунки нарівні з вебсайтами. Багато сервісів узагалі функціонують лише у форматі застосунку, наприклад, TikTok або Uber.

Сучасні мобільні застосунки мають низку характерних особливостей:

- **інтеграція з апаратними функціями смартфона** — камера, GPS, акселерометр, Bluetooth;
- **високий рівень персоналізації** — збереження даних користувача, налаштування тем, мов, уподобань;
- **робота з хмарними сервісами** — для зберігання фото, відео, резервного копіювання, синхронізації;
- **підтримка push-сповіщень** — дозволяє оперативно інформувати користувача про важливі події;
- **офлайн-доступ** — частина даних може зберігатись локально для використання без інтернету;
- **використання штучного інтелекту** — наприклад, у персональних рекомендаціях, чат-ботах, розпізнаванні зображень;
- **інтерфейс, адаптований під сенсорне керування** — інтуїтивний дизайн, швидка навігація, великі елементи управління.

Історично застосунки поділяються на три основні типи: **нативні**, **веб-застосунки** та **гібридні**. Нативні створюються окремо для кожної операційної системи (наприклад, Swift для iOS, Kotlin/Java для Android). Вони забезпечують найвищу швидкість і повноцінний доступ до функцій пристрою. Веб-застосунки — це адаптовані сайти, які відкриваються через браузер, а гібридні поєднують переваги обох типів і часто створюються з використанням фреймворків, таких як **Flutter**, **React Native**, **Xamarin** [□3].

Важливо також відзначити зміну підходів до дизайну та взаємодії. Якщо раніше достатньо було простої функціональності, то сьогодні інтерфейс має бути мінімалістичним, зручним, швидким і адаптивним. Користувачі не готові витрачати час на навчання роботі із застосунком — усе має бути інтуїтивно зрозумілим з перших секунд.

На даний момент розвиток мобільних застосунків продовжується в напрямку підвищення автоматизації, персоналізації та глибшої інтеграції з іншими цифровими середовищами: розумними годинниками, голосовими помічниками, інтернетом речей. Усе це відкриває нові можливості для створення застосунків, які не просто виконують одну функцію, а стають повноцінними помічниками у повсякденному житті.

1.2 Переваги мобільних застосунків

Мобільні застосунки стали звичною частиною повсякденного життя. Їх використовують у транспорті, на роботі, вдома, у супермаркеті, у черзі до лікаря — у будь-який момент, коли потрібно щось дізнатися, оформити чи просто зекономити час. Це перетворило смартфон на багатофункціональний інструмент, а самі застосунки — на ключову форму взаємодії між людиною та цифровим світом.

Основною перевагою таких рішень є швидкий доступ до функціоналу. Після встановлення застосунок запускається за секунду і не потребує повторного введення адреси сайту, логіну чи пароля (якщо передбачено збереження сесії). Це дозволяє оперативно реагувати на потребу — наприклад, підібрати образ перед виходом із дому, ще не встигнувши навіть прокинутись до кінця.

Наступною важливою перевагою є персоналізація. Сучасні застосунки здатні запам’ятовувати вибір, реакції та вподобання користувача. У контексті розробленого застосунку це проявляється в тому, що речі сортуються за категоріями, кольором, стилем, а образи підбираються не випадково, а з урахуванням попередніх вподобань і настрійового фону. Це дозволяє зробити

взаємодію не просто технічною, а людською — коли програма ніби розуміє, чого хочеться саме сьогодні.

Інтеграція з іншими функціями пристрою — ще один ключовий плюс. Мобільний застосунок може отримувати дані про погоду за геолокацією, відкривати камеру, надсилати push-сповіщення, зчитувати зображення або синхронізуватись із календарем. Це розширює межі функціоналу далеко за рамки простого «перегляду даних». У підсумку — формується досвід, коли програма дійсно працює "разом з тобою", а не замість тебе.

Окремо варто згадати автономність. У багатьох випадках мобільні застосунки здатні працювати частково або повністю без інтернету. Якщо інформація кешується або зберігається у базі даних на пристрої, користувач має доступ до ключового функціоналу навіть у зоні з поганим покриттям. Це особливо зручно в транспорті, за містом, або у ситуаціях, коли з’єднання тимчасово відсутнє.

Ще однією перевагою є інтерфейс. У порівнянні з сайтами, де елементи можуть виглядати по-різному залежно від браузера, у мобільних застосунках усе виглядає стабільно й однаково — дизайн контролюється розробником до кожного пікселя. Це спрощує навігацію, дозволяє уникнути візуального шуму, зробити акценти на важливому. У розробленому застосунку, наприклад, кожна категорія одягу має власне вікно перегляду, а користувач одразу розуміє, що і де знаходиться.

Не можна оминути увагою й те, що сучасні мобільні застосунки все частіше інтегрують штучний інтелект. І це не футуристика — це вже реальність. Алгоритми аналізують не лише зображення чи текст, а й настрій, кольорові поєднання, звички. У застосунку, створеному в межах дипломної роботи, якраз реалізовано можливість формування образу з поясненням: чому вибрано саме цю кофту, чому саме ці штани пасують до погоди й настрою. Така функція не просто допомагає — вона навчає, формує стиль, робить людину впевненішою у власному вигляді.

У результаті можна сказати, що мобільні застосунки стали універсальним інструментом: вони швидкі, адаптивні, розумні. Їхня сила — у гнучкості, можливості працювати під конкретного користувача, автоматично адаптуючись до змін навколишнього середовища та поведінки. Саме ці властивості зробили

мобільні застосунки ключовим середовищем для реалізації нових сервісів — від комунікацій до персонального підбору одягу.

1.3 Види мобільних застосунків

У межах розвитку мобільних технологій сформувалося кілька основних типів застосунків, які відрізняються способом розробки, архітектурою, рівнем доступу до ресурсів пристрою та можливостями масштабування. Кожен вид має свої переваги й обмеження, які враховуються під час вибору платформи для створення нового продукту.

Першу групу становлять **нативні мобільні застосунки**. Їх створюють спеціально для конкретної операційної системи — Android або iOS. Для розробки під Android використовуються мови Java або Kotlin, а під iOS — Swift або Objective-C. Нативні застосунки мають найкращу продуктивність, повноцінно використовують ресурси пристрою, мають доступ до всіх API, сенсорів та системних функцій. Завдяки цьому вони найчастіше використовуються для складних, інтерактивних або ресурсоємних проєктів — мобільні ігри, банківські застосунки, соціальні мережі.

До переваг нативних застосунків належать висока швидкість роботи, стабільність, можливість реалізувати унікальний дизайн під стандарти конкретної платформи. Але їх розробка вимагає окремої команди для кожної ОС, що значно підвищує витрати. Крім того, необхідно підтримувати дві різні кодові бази, що ускладнює оновлення і тестування.

Інший тип — **вебзастосунки**. Це програми, які працюють у браузері мобільного пристрою, не потребують встановлення, але при цьому мають обмежений функціонал. Вони створюються за допомогою HTML, CSS, JavaScript і зазвичай використовуються для спрощених сервісів або інформаційних порталів. Їхня головна перевага — кросплатформеність: одна версія працює на всіх пристроях.

Попри зручність, вебзастосунки мають низку обмежень: немає повного доступу до ресурсів пристрою, обмежена взаємодія з камерами, мікрофонами, Bluetooth, а також менша швидкість роботи. Тому вони підходять лише для проєктів, де не потрібно глибокої інтеграції з функціями смартфона.

Між цими двома крайнощами знаходяться **гібридні застосунки**. Це продукти, які створюються одразу для кількох платформ, використовуючи єдину кодову базу. Найпопулярніші фреймворки для такого підходу — **Flutter**, **React Native**, **Xamarin**. Вони дозволяють писати код один раз, а потім збирати проєкт для Android і iOS. У більшості випадків це оптимальний вибір для стартапів або середніх застосунків, де важлива економія часу та бюджету.

Гібридні застосунки поєднують у собі переваги обох типів: вони мають майже нативну швидкодію, мають змогу інтегруватися з більшістю системних функцій, а також значно швидше оновлюються і тестуються. Саме цей підхід було обрано при створенні застосунку для підбору одягу. Було використано фреймворк **Flutter**, що дозволило зосередитись на логіці застосунку та дизайні, не витрачаючи ресурси на дублювання коду для кожної платформи.

Окремо варто виділити **прогресивні вебзастосунки** (PWA — Progressive Web App). Це сучасний тип вебрішень, який частково імітує мобільні застосунки: вони можуть запускатись у повноекранному режимі, працювати офлайн, надсилати push-сповіщення. PWA дозволяє створити застосунок без публікації в App Store чи Google Play, але все ще має обмеження у швидкодії та взаємодії з деякими API.

Кожен із розглянутих видів має свої сфери застосування. Вибір між ними залежить від цілей, ресурсів, очікуваної аудиторії та необхідної глибини функціоналу. У випадку проєкту, присвяченого підбору одягу за погодою та настроєм, важливо було забезпечити кросплатформенність, швидкість розробки, доступ до камери, локальне зберігання даних та інтеграцію з Firebase. Усі ці критерії найкраще виконував саме гібридний підхід із використанням Flutter.

У підсумку можна сказати, що чітке розуміння типів мобільних застосунків дозволяє обрати технічно й логічно обґрунтовану модель розробки. Від цього

залежить не лише функціональність, а й швидкість створення, зручність користування, масштабованість і подальший розвиток продукту.

1.4 Аналіз існуючих аналогів

На сьогодні у цифровому просторі представлено велику кількість мобільних застосунків, які тією чи іншою мірою пов’язані з гардеробом, стилем та створенням образів. Частина з них працює як особистий каталог речей, інші — надають рекомендації, а деякі — виконують функцію онлайн-шопінгу з елементами стилістики. Проте більшість цих застосунків орієнтовані на одне конкретне завдання і не поєднують погодні умови, настрій користувача та логіку підбору образу в єдину систему.

Одним із найбільш відомих аналогів є застосунок **Smart Closet**. Його функціонал включає можливість додавання фотографій одягу, створення лукбуків, планування образів на тиждень. Є підтримка фільтрації за кольором, сезоном, брендом. Проте відсутній автоматичний підбір одягу, немає врахування емоційного стану користувача, а рекомендації створюються вручну.

Застосунок **Stylebook** також дозволяє завантажувати речі, створювати капсульний гардероб і фіксувати, що було вдягнено в який день. Програма має зручний календар, лічильник носінь, статистику за сезонами. Але всі дії виконуються вручну, автоматизація відсутня, а інтерфейс вже виглядає застарілим. Крім того, додаток орієнтований переважно на користувачів iOS, без версії для Android [□4].

Ще один приклад — **Pureple**. Це безкоштовний застосунок для керування гардеробом. Він має функцію автосортування, може створювати образи випадковим чином або за певними тегами. Проте відсутній зв'язок з погодними умовами, не реалізовано аналіз настрою, а пояснення вибору образу не надається.

Інший приклад — **Acloset**. Цей застосунок має сучасний дизайн, підтримує автоматичне розпізнавання типу одягу з фото, веде статистику носіння та дає рекомендації на основі минулих виборів. Водночас логіка рекомендацій

незрозуміла, і в інтерфейсі відсутні пояснення: чому запропоновано саме цей образ. Крім того, Acloset не враховує погодні умови та не дозволяє обирати настрій, що є ключовими параметрами в рамках поставленої мети дипломної роботи.

Частково функції персоналізації реалізовано в таких застосунках, як **Lookiero** або **Cladwell**. У них враховуються уподобання користувача, і навіть пропонуються добірки речей на день. Але їх логіка орієнтована на онлайн-шопінг — мета в тому, щоби продати новий одяг, а не використати вже наявний у гардеробі. Такі сервіси вимагають щомісячної підписки, а без інтернету взагалі не працюють. Це знижує зручність використання і створює бар'єр для широкої аудиторії.

У процесі аналізу було виявлено спільну рису — більшість аналогів не надають глибокої персоналізації образу. Вони або фіксують наявний одяг, або випадковим чином поєднують речі без пояснень, або фокусуються лише на шопінгу. Водночас можливість враховувати зовнішні (погода) та внутрішні (настрій) чинники, пояснювати підбір образу та зберігати весь гардероб у цифровому вигляді — залишається практично невикористаною.

На основі порівняння стало очевидним, що на ринку відсутній застосунок, який об'єднує:

- автоматичний підбір образу;
- погодні умови за геолокацією;
- настрій користувача;
- логічне пояснення вибору;
- можливість редагування та гортання елементів вручну;
- повну незалежність від онлайн-магазинів.

Це створює простір для реалізації нового підходу — застосунку, який дійсно «думає» разом із користувачем, формує лук на основі кількох параметрів, пояснює свій вибір і дозволяє редагувати результат. Саме така логіка й була закладена у проєкт дипломної роботи.

1.5 Цільова аудиторія

Визначення цільової аудиторії є одним із ключових етапів під час проєктування будь-якого цифрового продукту. Це дозволяє краще зрозуміти потреби майбутніх користувачів, сформувати зручний інтерфейс, підібрати відповідні функції та логіку взаємодії. У випадку застосунку для підбору одягу за погодою та настроєм важливо не лише з’ясувати вік чи соціальний статус аудиторії, а й розуміти її поведінкові та емоційні особливості.

У першу чергу до цільової аудиторії належать **молоді люди віком від 16 до 35 років**. Це покоління, яке звикло до швидких рішень, цифрових сервісів і мобільності. Вони щоденно користуються смартфонами, цінують персоналізацію, прагнуть виглядати стильно та не витрачати час на рутину. Саме в цій категорії найчастіше виникає потреба у зручному, візуальному, інтуїтивно зрозумілому інструменті, який допоможе вирішити питання "що вдягнути" — швидко, красиво й доречно.

До цієї групи також відносяться **студенти та молоді спеціалісти**, які часто змінюють локації, відвідують навчальні заклади, офіси, неформальні зустрічі. У них, як правило, вже є сформований гардероб, але відсутній час або бажання щоразу вручну комбінувати речі. Застосунок, який бере на себе завдання формування образу, стає цінним помічником, що не тільки заощаджує час, а й допомагає краще організувати свій одяг.

Окрема частина потенційної аудиторії — це **ті, хто захоплюється модою**, але не завжди впевнено почувається у власному стилі. У цій категорії особливо важливою є функція пояснення вибору — чому підібрано саме ці речі, як вони поєднуються, чим пасують до погоди або настрою. Застосунок у такому випадку виконує ще й освітню роль, допомагаючи розвивати власний стиль і навички комбінування одягу.

Варто також виділити **користувачів, які прагнуть раціоналізувати гардероб**. Це люди, які намагаються уникати зайвих покупок, підтримують ідею «капсульного підходу», хочуть максимально ефективно використовувати вже наявні

речі. Вони часто стикаються з проблемою великої кількості одягу, який "не носиться", бо складно згадати, з чим його поєднати. У цьому випадку цифрова галерея одягу й автоматизований підбір образів допомагає оптимізувати використання гардеробу.

Додатково передбачено охоплення **користувачів із непостійним настроєм або емоційною чутливістю**, яким складно приймати побутові рішення. У таких випадках застосунок бере на себе ще й частину психологічного навантаження — вибір одягу перестає бути проблемою, натомість перетворюється на приємний ритуал із елементом персоналізованої підтримки.

Усі перераховані групи мають спільні характеристики: постійне використання смартфону, позитивне ставлення до мобільних застосунків, інтерес до персоналізованих сервісів, потребу в економії часу, бажання виглядати добре у будь-якій ситуації.

У результаті формування цільової аудиторії стало можливим точніше визначити функціонал застосунку: просте додавання одягу, автоматичне формування образу, врахування настрою, пояснення вибору, зрозумілий інтерфейс, відсутність реклами, кросплатформенність. Усе це забезпечує комфортний досвід для основної групи користувачів і підвищує шанси застосунку на успішне впровадження в реальне використання.

1.6 Визначення вимог до застосунку

Після аналізу аналогів, вивчення поведінки потенційної аудиторії та визначення загальної ідеї застосунку, сформульовано основні функціональні та нефункціональні вимоги до системи. Ці вимоги є основою для подальшої реалізації архітектури, інтерфейсу, логіки взаємодії та вибору інструментів розробки.

Перш за все, застосунок має забезпечувати **автоматичне формування образу на день**, використовуючи дані про:

- погодні умови в місці перебування користувача;
- настрій, обраний вручну при вході;

- наявні речі у гардеробі, які вже були додані користувачем раніше.

Це вимагає інтеграції з **геолокаційним API** (для визначення координат) та **погодним API** (наприклад, OpenWeather), а також реалізації інтерфейсу для вибору емоційного стану. Образ формується на основі поєднання щонайменше одного елемента з кожної категорії: аксесуари, голова, шия, тіло, руки, ноги, ступні. Важливо, щоб результат не був випадковим — підбір має враховувати кольорову гармонію, стиль і доцільність речей з точки зору функціональності (наприклад, не пропонувати легку футболку при мінус п’яти).

Передбачено реалізацію функції пояснення вибору образу. На поточному етапі застосунок формує текстове пояснення за допомогою умовної логіки на основі кольору, категорії та настрою. В подальшому заплановано інтеграцію зі штучним інтелектом для генерації більш точного та персоналізованого опису.

Ще одним обов’язковим компонентом є **галерея речей користувача**. Має бути реалізована можливість додавання фотографій одягу через камеру або з галереї пристрою. При завантаженні зображення повинно бути автоматично очищене від фону (за допомогою інструменту видалення фону або спеціального API), після чого зберігається у базі даних із зазначенням категорії, кольору (визначеного автоматично) та стилю (може задаватися вручну).

Усі дані зберігаються у **хмарній базі Firebase**, що дозволяє синхронізувати гардероб між пристроями, не втрачаючи речі при перевстановленні або переході на новий смартфон.

Функціональні вимоги:

- додавання одягу з фото та автоматичним видаленням фону;
- збереження речей у категоріях, з тегами та кольором;
- автоматичне визначення погоди за місцем перебування користувача;
- ручний вибір настрою з емоджі-панелі;
- формування образу за допомогою RecommendationService;
- пояснення вибору одягу на основі параметрів;
- можливість вручну змінити окремі елементи образу після генерації;

- доступ до профілю користувача з даними акаунту, аватаром і лічильником збережених образів.

Нефункціональні вимоги:

- стабільна робота як на Android, так і на iOS (через Flutter);
- привабливий, інтуїтивно зрозумілий інтерфейс;
- локалізація інтерфейсу (мінімум українська мова);
- відсутність реклами;
- офлайн-доступ до завантажених речей і останнього збереженого образу;
- захист даних користувача та авторизація через Firebase Authentication.

В решті решт було сформовано повноцінне технічне бачення, що дозволяє реалізувати застосунок, який не лише зберігає одяг, а й «думає» разом із користувачем. Акумуляція функцій підбору, пояснення, настроєвого впливу та кліматичних умов забезпечує комплексний підхід до вирішення щоденної задачі — що вдягнути сьогодні.

2 АНАЛІЗ ТЕХНОЛОГІЙ ТА ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Засоби розробки системи

Для реалізації мобільного застосунку, який автоматизує підбір одягу на основі погоди та настрою, було обрано набір сучасних інструментів, що дозволяють забезпечити кросплатформенність, швидку розробку, зручний інтерфейс і гнучкість у майбутньому масштабуванні. Усі засоби були підібрані з урахуванням вимог до продуктивності, зручності підтримки, стабільності та наявності спільноти.

Основною технологією розробки було обрано **Flutter** — відкритий фреймворк від компанії Google для створення мобільних, веб- та десктоп-застосунків з єдиною кодовою базою [□5]. Його основною перевагою є можливість створення кросплатформенних інтерфейсів, які виглядають однаково на Android та iOS, без необхідності писати окремий код під кожну систему. Flutter використовує мову **Dart**, яка поєднує простоту синтаксису з високою продуктивністю виконання.

Однією з ключових переваг Flutter стало наявність великої кількості готових бібліотек, зокрема для роботи з Firebase, HTTP-запитами, анімаціями, формами, а також UI-компонентами, які можна легко кастомізувати [□6]. Використання hot reload значно пришвидшило процес розробки, дозволяючи одразу бачити результат змін без перезапуску застосунку [□7].

Для зберігання даних, авторизації користувача, обробки зображень і хостингу було використано **Firebase** — набір хмарних сервісів, також розроблений Google.

Зокрема, у межах даного проєкту застосовано такі компоненти Firebase:

- **Firebase Authentication** — для реєстрації, логіну та керування акаунтом користувача;
- **Firebase Storage** — для зберігання зображень одягу у хмарі після їх обробки.

Для визначення погодних умов використано зовнішній API — **OpenWeatherMap**. Це дозволило отримувати актуальні дані про температуру, стан неба, вітер тощо, враховуючи геолокацію користувача. Інформація

використовується при формуванні образу — наприклад, у разі дощу система не пропонуватиме легке взуття або відкрите вбрання. Дані запитуються через HTTP-запит, а результат обробляється асинхронно у Flutter.

Для роботи з геолокацією використано бібліотеку **geolocator**, яка дозволяє отримати координати користувача у режимі реального часу. Ця інформація автоматично підставляється у погодний запит, що знімає необхідність ручного введення міста або адреси.

Обробка зображень одягу реалізується із застосуванням сторонніх бібліотек для видалення фону — зокрема, може бути використано **remove.bg API** або інші інструменти комп'ютерного зору. На етапі написання дипломної роботи частину функцій реалізовано у вигляді заглушок із можливістю подальшого оновлення. Зображення обрізаються, центровані та зберігаються у Firebase Storage.

Окремо передбачено формування текстових пояснень до згенерованого образу. Наразі ця функція реалізована у вигляді умовної логіки: пояснення створюється на основі категорій речей, кольору, імені настрою та погодних тегів. У подальшому планується інтеграція з моделлю OpenAI (через GPT API) для генерації більш персоналізованих, стилістично адаптованих пояснень.

Для керування станом застосунку використовується підхід **Stateful Widgets** із розбиттям логіки за екранами. У майбутньому можливе впровадження більш масштабованого підходу, наприклад, за допомогою **provider** або **bloc**, якщо обсяг даних зростатиме.

Розробка ведеться у середовищі **Android Studio** з повною підтримкою Flutter SDK, інтеграцією емуляторів та можливістю тестування на фізичному пристрої.

У підсумку, обрані засоби забезпечили стабільність, швидкість та гнучкість розробки, дозволили реалізувати основний функціонал у межах дипломної роботи, а також створили фундамент для подальшого вдосконалення та масштабування проєкту.

2.2 Flutter

2015 рік ознаменував початок нової ери в мобільній розробці з появою Flutter — інструментарію від Google, який кинув виклик традиційним підходам і монополії Java та Swift. Flutter обіцяв простоту, швидкість та нативну візуалізацію, пропонуючи альтернативу для створення мобільних застосунків з єдиною кодовою базою.

Після двох років активної розробки, у 2017 році було представлено Flutter 1.0. Цей реліз став сигналом для індустрії: фреймворк готовий до реальних проєктів. У 2018 році з'явилася щорічна конференція Flutter Engage, яка об'єднала спільноту навколо інструменту. Уже в 2019 році Flutter отримав масштабне оновлення, що значно покращило продуктивність, розширило UI-бібліотеки та зробило платформу ще більш зручною для розробників.

2020 рік став переломним — було додано підтримку вебзастосунків, що зробило Flutter не просто мобільним фреймворком, а універсальним інструментом. У 2021 році з виходом Flutter 2.0 було розширено можливості доступності, стабільності та інтеграції, а вже у 2022 — Flutter 3.0 відкрив двері для десктопних застосунків. З кожним роком зростали як функціональність, так і довіра з боку спільноти.

У 2023–2024 роках Flutter досяг піку стабільності, масштабованості та швидкодії. Його позиція як лідера у сфері кросплатформеної розробки вже не викликала сумнівів. Спільнота зросла до мільйонів розробників, а екосистема — до тисяч плагінів, рішень та навчальних матеріалів.

У межах дипломної роботи Flutter було обрано як основну платформу для створення мобільного застосунку, що автоматизує підбір одягу за погодними умовами та настроєм користувача. Головною причиною вибору стала можливість написати один код для двох платформ — Android і iOS, що значно пришвидшило розробку і спростило підтримку.

Flutter побудований на мові Dart, яка дозволила легко організувати логіку програми та асинхронну взаємодію з API. Фреймворк надає готові UI-компоненти

— **віджети**, які легко налаштовуються. Було використано як базові елементи (Text, Icon, ElevatedButton, Column, Row), так і складніші — ListView, Stack, PageView.

Серед основних можливостей Flutter, які активно застосовувалися у проєкті:

- **Hot Reload** — дозволив пришвидшити розробку завдяки миттєвому оновленню інтерфейсу;
- **Stateful Widgets** — забезпечили динамічність таких екранів, як головна сторінка, форма настрою та рекомендаційний блок;
- **Гнучке компонування** — через Stack, Positioned, GridView вдалося побудувати складні елементи, наприклад, галерею речей або кастомні діалоги;
- **Адаптація під роздільну здатність екрану** — реалізовано через MediaQuery та LayoutBuilder;
- **Інтеграція з HTTP-запитами** — для отримання даних про погоду, звернень до OpenAI, обміну з Firebase;
- **Обробка станів** — реалізовано через setState, а в перспективі можливе впровадження Provider або Riverpod для масштабування.

Flutter також забезпечив просту інтеграцію з Firebase — для авторизації, зберігання зображень та синхронізації даних гардеробу. Інтерфейс логіну, профілю, форм реєстрації та завантаження одягу було створено саме через стандартні компоненти Flutter, доповнені анімаціями для кращого UX.

Застосування Flutter дозволило реалізувати повноцінний застосунок зі складною логікою, інтерактивним інтерфейсом та перспективою масштабування. Платформа виявилась стабільною, швидкою і при цьому достатньо гнучкою для реалізації індивідуального підходу до користувача.

2.3 Dart

Мова програмування Dart стала основною мовою при розробці застосунку для підбору одягу в межах дипломної роботи. Її було створено компанією Google у 2011 році як універсальну, сучасну мову з орієнтацією на зручність розробника,

швидкість виконання та підтримку сучасних підходів у розробці інтерфейсів користувача.

Спочатку Dart позиціонувався як заміна JavaScript для браузерів, але згодом фокус змістився на створення високопродуктивних інтерфейсів, що призвело до його інтеграції у фреймворк Flutter. Саме завдяки цій інтеграції мова отримала новий виток популярності, ставши основою для створення кросплатформених мобільних, веб- і десктоп-застосунків.

Dart поєднує об’єктно-орієнтовану модель із підтримкою сучасних синтаксичних конструкцій, таких як лямбда-функції, null safety, розширення, міксини, асинхронність через async/await та стріми [8]. Код виглядає компактно і водночас читабельно, що значно спрощує підтримку й масштабування проєкту.

У межах розробки застосунку Dart дозволив реалізувати:

- структуру екрану як класу (через Stateful або Stateless Widget);
- логіку роботи з API (наприклад, отримання погоди або виклик функцій генерації рекомендацій);
- обробку асинхронних операцій: запит до Firebase, збереження зображень, завантаження галереї;
- динамічне оновлення інтерфейсу при зміні стану користувача чи результатів генерації образу;
- обробку даних (фільтрацію, сортування речей, визначення активних категорій).

Особливо важливою стала підтримка **null safety** — системи контролю, яка запобігає виклику властивостей чи методів у випадках, коли змінна не ініціалізована. Це дозволило уникнути багатьох помилок, що типові для мобільної розробки, особливо на ранніх етапах.

Для реалізації структури проєкту було застосовано принципи **розділення відповідальностей**: логіка для отримання даних винесена у окремі класи-сервіси, компоненти інтерфейсу згруповані у власні файли, що спрощує навігацію та масштабування застосунку.

Асинхронна модель Dart (Future, async/await, Stream) дозволила ефективно працювати з Firebase, OpenWeather API, а також формувати взаємодію з

користувачем у реальному часі. Наприклад, під час очікування відповіді від API застосунок не зависає, а плавно оновлює вміст або відображає індикатор завантаження.

Крім того, Dart має зручні конструкції для **роботи зі списками** (методи map, where, reduce), що активно застосовувалися під час обробки масиву речей користувача, генерації варіантів образів та фільтрації за категоріями. Це дозволило зменшити обсяг коду та зробити його більш гнучким до змін у майбутньому.

Серед переваг Dart, які виявились особливо корисними у проєкті, можна назвати:

- **швидка компіляція** — як у режимі JIT (під час розробки), так і AOT (на етапі релізу);
- **висока продуктивність** — код виконується швидко, а інтерфейс працює плавно;
- **зрозумілий синтаксис** — легко читати, писати та підтримувати;
- **глибока інтеграція з Flutter** — відсутність необхідності в конфігураціях або сторонніх обгортках;
- **велика кількість документації та прикладів** — що спростило адаптацію й пошук рішень.

Мову Dart було обґрунтовано обрано для реалізації застосунку, оскільки вона забезпечила стабільну, зручну та масштабовану основу для всієї логіки програми. Її поєднання з Flutter дало змогу зосередитися на реалізації функціональності без зайвих ускладнень, а можливості мови покрили всі потреби розробки на рівні як інтерфейсу, так і бізнес-логіки.

2.4 Firebase

Firebase — це набір хмарних сервісів від Google, який дозволяє розробникам створювати повноцінні клієнт-серверні застосунки без потреби у власному бекенді. У межах дипломної роботи було використано два основні компоненти платформи — **Firebase Authentication** та **Cloud Firestore** [□9]. Ці інструменти дозволили реалізувати реєстрацію, авторизацію, збереження даних користувача та гардеробу.

Firebase Authentication забезпечує механізм створення облікових записів користувачів із використанням електронної пошти та паролю. Платформа бере на себе всю обробку: зберігання зашифрованих паролів, перевірку логіну, захист від зловживань, скидання пароля, підтвердження пошти. Це значно зменшує ризик помилок у безпеці та дозволяє зосередитися на реалізації основного функціоналу.

У застосунку реалізовано можливість реєстрації нового користувача з введенням пошти, пароля та додаткових полів — наприклад, нікнейму. Після реєстрації користувач автоматично потрапляє на головну сторінку, де зберігається його сеанс. При повторному вході достатньо лише авторизації — доступ до гардеробу та налаштувань відбувається автоматично. Для цього Firebase надає токени доступу та слугує провайдером сесій.

Для збереження одягу, категорій, кольору, обраного настрою, згенерованих луків та іншої структурованої інформації використано **Cloud Firestore** — масштабовану NoSQL-базу даних. Вона дозволяє зберігати дані у форматі колекцій і документів. Такий підхід ідеально підходить для мобільних застосунків, де потрібна гнучкість, реальний час і зручна структура [10].

У проєкті були створені окремі колекції для користувачів та їхніх речей. Кожен користувач отримує унікальний uid, що дозволяє ідентифікувати його в базі. У колекції users зберігається загальна інформація: email, нікнейм, дата створення акаунту. У колекції clothes містяться документи з параметрами кожної одиниці одягу: назва, категорія, колір, стиль, тег настрою, дата додавання.

Firestore надає можливість виконувати фільтрацію, сортування, оновлення даних у реальному часі, а також синхронізувати їх із локальним кешем, що дозволяє швидко відображати інформацію навіть при слабкому інтернет-з’єднанні.

Інтеграція Firebase у Flutter відбувається через офіційні пакети: firebase_auth для авторизації та cloud_firestore для взаємодії з базою даних. Було реалізовано обробку виключень, сповіщення про помилки входу (наприклад, "невірний пароль" або "користувача не існує"), а також контроль доступу до контенту після авторизації.

Переваги використання Firebase у цьому проєкті:

- **швидка інтеграція** — налаштування займає мінімум часу;

- **високий рівень безпеки** — Google забезпечує захист автентифікації та доступу;
- **відсутність потреби у власному сервері** — уся логіка бекенду реалізується через API;
- **масштабованість** — при зростанні кількості користувачів система зберігає стабільність;
- **підтримка Flutter на офіційному рівні** — усі пакети добре документовані та підтримуються спільнотою.

Використання Firebase дозволило зменшити кількість ручної роботи, пов'язаної з безпекою та базами даних, а також забезпечити надійне зберігання користувацьких даних і швидкий доступ до них у межах мобільного застосунку. Це створило надійний фундамент для подальшого розвитку та масштабування проєкту.

2.5 Cloudinary API

Для реалізації зручного зберігання фотографій одягу в межах дипломного проєкту було обрано **Cloudinary API** — зовнішній хмарний сервіс, що спеціалізується на обробці, оптимізації та доставці медіаконтенту. Його перевага полягає в повній готовності до роботи з візуальними ресурсами без потреби самостійно розгортати систему зберігання, обробки та масштабування зображень [□11].

Cloudinary забезпечує швидке завантаження, зберігання та доступ до зображень за допомогою REST API. Це дає змогу не залежати від власного сервера або Firebase Storage, а використовувати спеціалізовану платформу, орієнтовану саме на графічний контент.

У межах проєкту Cloudinary було використано для збереження фотографій одягу, які користувач завантажує під час формування власного гардеробу. Кожне зображення проходить етапи:

1. Завантаження через інтерфейс застосунку (з камери або галереї);
2. Видалення фону або кадрування (локально або через інший сервіс);
3. Надсилання готового зображення на сервер Cloudinary;

4. Отримання посилання на файл, яке зберігається у Firestore разом з іншими параметрами одягу.

Сервіс дозволяє не лише зберігати зображення, а й автоматично виконувати низку трансформацій: зменшення розміру, обрізання, конвертація у WebP або інші формати, оптимізація за якістю. Ці функції можна задавати як під час завантаження, так і при формуванні URL-адреси. Наприклад, у дипломному застосунку використовувалась трансформація зменшеного розміру з автоматичним центрованим обрізанням, що дозволяло уніфікувати вигляд усіх речей у галереї.

Для інтеграції з Flutter було використано HTTP-запит до API Cloudinary. Дані зображення передавались у форматі multipart/form-data, а у відповіді поверталась структура з URL до зображення, його ID та технічними параметрами. Посилання одразу зберігалось у Firestore разом із категорією, кольором та іншими тегами речі.

Серед переваг використання Cloudinary:

- **висока швидкість доставки** контенту через вбудовану CDN (Content Delivery Network);
- **автоматичне масштабування** — не потрібно піклуватися про об’єм сховища;
- **гнучкість обробки** — можливість налаштувати розмір, якість, обрізку прямо через URL;
- **детальна документація** та приклади для інтеграції;
- **надійність** — стабільна робота навіть при великій кількості зображень.

Також сервіс має функціонал видалення зображень за ID, що дозволяє реалізувати повне керування гардеробом користувача: у разі видалення речі з бази, відповідне зображення також може бути видалено з хмарного сховища.

Cloudinary не потребує складних налаштувань — для старту достатньо створити акаунт, отримати API-ключі та інтегрувати їх у застосунок. У проєкті було використано безплатний тариф, якого вистачає для зберігання тисяч зображень — цього цілком достатньо для MVP.

У результаті використання Cloudinary дозволило винести всю роботу з візуальним контентом на зовнішній сервіс, зменшивши навантаження на застосунок і спростивши розгортання. Це дало змогу сфокусуватись на основній

логіці — підборі образу — та забезпечити стабільне, швидке й масштабоване зберігання зображень без складної серверної частини.

2.6 Remove.bg API

Одним із ключових викликів при реалізації застосунку стало забезпечення охайного вигляду візуального контенту. Речі в гардеробі мають відображатися без зайвого фону, що дозволяє фокусувати увагу користувача на самому одязі та формувати естетичний інтерфейс. Щоб досягти цього, було інтегровано **Remove.bg API** — зовнішній сервіс, який автоматично видаляє фон із фотографій за допомогою комп’ютерного зору та глибинного навчання [12].

Remove.bg — один із найпопулярніших інструментів для обробки зображень, який не потребує ручного редагування. Його API дозволяє надіслати файл або посилання на зображення, а у відповідь отримати PNG із прозорим фоном. Сервіс підтримує автоматичне вирізання об’єктів будь-якої складності: одяг, люди, аксесуари, тварини.

У межах дипломного проєкту Remove.bg використовується на етапі додавання нової речі до гардеробу. Після того, як користувач обирає зображення (з камери або галереї) та задає всі необхідні параметри — категорію, стиль, назву, колір — зображення надсилається на обробку до Remove.bg. Лише після завершення видалення фону, оброблене PNG зберігається в Cloudinary, а відповідне посилання — разом з метаданими речі — додається до бази даних Firestore.

Інтеграція відбувається через REST API, де в заголовку запиту вказується ключ авторизації, а тіло запиту формується у вигляді multipart/form-data. Після обробки зображення результат зчитується як Uint8List, і може бути збережений локально або переданий до хмарного сховища.

Такий підхід дозволяє повністю автоматизувати процес: користувач бачить лише результат — готову річ без фону у своїй цифровій шафі. Ручна обрізка, використання фоторедакторів чи спеціальних навичок не потрібні.

Серед переваг використання Remove.bg у застосунку:

- **висока точність вирізання об'єкта**, збереження дрібних деталей, текстур;

- **швидкість** — у середньому обробка займає до 2 секунд;
- **простота інтеграції через HTTP-запит**;
- **можливість попереднього перегляду результату перед збереженням**;
- **відсутність необхідності у власному сервері обробки зображень**.

У випадку, якщо зображення не вдається обробити (через помилку мережі або API), застосунок повідомляє користувача про проблему, і повторна спроба стає доступною. Усі ці дії відбуваються у фоні, без необхідності взаємодії з технічними налаштуваннями.

Важливою особливістю є те, що Remove.bg API працює без потреби в складних SDK або сторонніх бібліотеках — достатньо звичайного HTTP-запиту. Це дозволило легко інтегрувати сервіс у Flutter-застосунок, використовуючи стандартний пакет http.

У межах дипломної було використано безплатний тариф, що дозволяє обробляти обмежену кількість зображень на місяць. Цього вистачило для тестування функціоналу. У перспективі, за потреби масштабування, передбачено перехід на платну підписку або реалізацію обробки через локальні алгоритми сегментації фону.

Завдяки Remove.bg вдалось реалізувати простий, але ефективний механізм, який забезпечує чистий вигляд галереї та сприяє точнішому розпізнаванню речей. Це стало важливою частиною візуального стилю застосунку й підвищило загальну якість користувацького досвіду.

2.7 OpenWeather API

Для реалізації функції підбору одягу з урахуванням погодних умов у застосунку було використано сервіс **OpenWeather API**. Це популярна платформа, яка надає доступ до погодних даних у реальному часі, а також прогнозів, кліматичних статистик та історичних значень. Сервіс широко застосовується в мобільних застосунках, розумних пристроях та автоматизованих системах прийняття рішень [□13].

У контексті дипломного проєкту OpenWeather використовується для отримання поточних погодних умов у місці перебування користувача. Дані використовуються під час генерації образу дня: якщо на вулиці дощ, обирається верхній одяг із захистом, якщо сонячно — підбирається легший варіант, а при зниженні температури пріоритет надається речам із довгими рукавами та утепленими тканинами.

Сервіс надає погодні дані через REST API. Для виконання запиту потрібні географічні координати (широта і довгота), які автоматично визначаються за допомогою GPS-модуля смартфона. Застосунок використовує Flutter-бібліотеку geolocator, яка запитує дозвіл у користувача та передає координати у запит до OpenWeather.

У відповіді від OpenWeather API повертається JSON-структура з температурою, типом погоди (ясно, хмарно, дощ, сніг), вологістю, вітром та іншими параметрами. У межах цього проєкту використовуються лише ключові показники:

- **температура повітря** (у градусах Цельсія);
- **тип погоди** (у вигляді опису та ID іконки);
- **стан неба** — як фактор для вибору кольорів та стилів образу.

На основі отриманої інформації формується текстовий опис погодних умов, який відображається на головній сторінці застосунку та одночасно використовується під час формування образу. Наприклад, у випадку фрази "легкий дощ і +14°C", система не запропонує відкрите взуття або футболку, а натомість згенерує поєднання з курткою, водостійким взуттям та шарфом.

Інтеграція OpenWeather API у Flutter-застосунок реалізована через стандартні HTTP-запити (http.get), а отримані дані обробляються асинхронно за допомогою Future та async/await. Це дозволяє працювати з API у фоновому режимі, не блокуючи інтерфейс користувача. У разі відсутності з’єднання або помилки API застосунок інформує користувача відповідним повідомленням.

OpenWeather API має безкоштовний тарифний план, якого було достатньо для реалізації дипломного проєкту. Він передбачає певну кількість запитів на добу, доступ до основних погодних параметрів і високу стабільність відповіді.

Платформа також підтримує багатомовність — у разі потреби можна отримувати описи погоди українською або англійською мовою.

Переваги використання OpenWeather API у застосунку:

- **актуальність даних у режимі реального часу**;
- **зручність інтеграції з Flutter**;
- **висока точність навіть при мінімальному тарифі**;
- **гнучка структура відповіді** — з можливістю вибирати лише потрібні параметри;
- **можливість масштабування** — перехід на тариф з прогнозом, історією, аналітикою.

OpenWeather API дозволив зробити підбір образу більш персоналізованим, релевантним до поточних умов, а також підвищити довіру користувача до застосунку. Цей компонент став одним із основних у реалізації концепції — одяг, що підбирається не випадково, а за реальними факторами, які відображають поточний стан навколишнього середовища.

2.8 Геолокація

Під час проєктування функціоналу, що враховує погодні умови, виникла потреба у визначенні точного місця перебування користувача. Це дозволяє автоматично формувати запити до погодного сервісу без ручного введення координат чи населеного пункту. Для реалізації цієї задачі було застосовано Flutter-бібліотеку **geolocator**, яка забезпечує зручний доступ до географічного розташування пристрою [□14].

Geolocator є стабільним пакетом, що підтримується спільнотою Flutter і надає функції отримання координат у режимі реального часу. Він дозволяє визначити широту та довготу з різною точністю — від приблизної (через Wi-Fi) до максимально точної (через GPS). Отримані координати автоматично передаються у запит до OpenWeather API, що дозволяє динамічно визначати актуальні погодні умови у момент генерації образу.

У процесі реалізації функції геолокації спочатку здійснюється запит дозволу в користувача. Якщо дозвіл надано, пристрій визначає поточне розташування з використанням методу getCurrentPosition() з параметром LocationAccuracy.high. У випадку, коли доступ заборонено, система реагує відповідним повідомленням та може запропонувати використати останні збережені координати або відкрити додаткові налаштування.

Для уникнення блокування інтерфейсу запити до GPS-модуля виконуються асинхронно. Дані передаються у вигляді Position, після чого викликається функція завантаження погодних умов за допомогою зовнішнього API. Геолокація активується один раз під час запуску застосунку або оновлюється вручну за потреби, що дозволяє зберігати баланс між точністю та енергоефективністю пристрою.

З метою підвищення зручності користувача, увесь процес відбувається у фоні — без потреби взаємодії з технічними елементами. Після авторизації автоматично визначається локація, і на головній сторінці відображається погодна інформація, актуальна для цього місця. Це забезпечує персоналізацію досвіду та дозволяє системі формувати лук з урахуванням кліматичних умов, що відповідають реальній ситуації.

У разі виникнення виняткових ситуацій — недоступність сервісу, відмова у доступі до GPS, помилки на стороні платформи — передбачено обробку помилок із відповідним інформуванням. Це дозволяє уникати аварійних завершень роботи застосунку та зберігати контрольовану поведінку.

Застосування геолокації в поєднанні з погодним API значно розширює можливості персоналізації. Система автоматично підлаштовується під зміну кліматичних умов, не потребуючи додаткових дій зі сторони користувача. У підсумку це підвищує точність рекомендацій, актуальність запропонованих образів і загальний рівень зручності.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОДУКТУ

3.1 Проєктування архітектури застосунку

Архітектура мобільного застосунку є ключовим етапом у процесі розробки, що визначає логіку взаємодії між його компонентами. Застосунок, метою якого є полегшення вибору одягу з урахуванням настрою користувача та погодних умов, був спроєктований відповідно до принципів модульності, масштабованості та повторного використання коду [15].

Було обрано клієнт-серверну архітектуру, у якій мобільний застосунок (клієнтська частина) взаємодіє з віддаленим сервером (базою даних та сторонніми API). Такий підхід дає змогу реалізувати централізоване зберігання даних користувача, забезпечити синхронізацію між пристроями та зменшити навантаження на клієнтську частину.

Основні компоненти архітектури включають:

- Клієнтська частина (Flutter-додаток): реалізує інтерфейс користувача, логіку взаємодії, обробку геолокації, зображень, авторизацію та відображення рекомендованих луків;
- Серверна частина: відповідає за зберігання та обробку даних у Firebase Firestore, а також за автентифікацію користувачів через Firebase Auth;
- Зовнішні сервіси: інтегровані для обробки зображень (Remove.bg API), зберігання фотографій (Cloudinary) та отримання погодних даних (OpenWeatherMap API) ;
- Сервіс рекомендацій: окремий модуль, відповідальний за підбір образу на основі введених параметрів. У майбутньому передбачено розширення з використанням нейронної мережі або GPT API для генерації пояснень.

Уся архітектура спроєктована з урахуванням можливості масштабування — як за кількістю користувачів, так і за функціональністю.

На рисунку 3.1 зображена схема взаємодії компонентів системи.

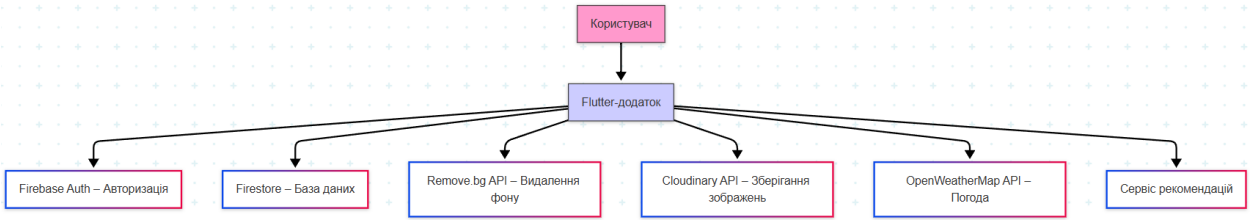


Рис. 3.1 Структурна схема взаємодії Flutter-додатку з сервісами Firebase, API та системою рекомендацій.

3.2 Архітектура клієнтської частини

Клієнтська частина мобільного застосунку реалізована з використанням фреймворку **Flutter**, що дозволяє створювати нативні інтерфейси для платформ Android та iOS з єдиною кодовою базою на мові **Dart**. Завдяки цьому досягається висока швидкість розробки, уніфікованість інтерфейсів, легкість у тестуванні та розгортанні застосунку. Крім того, Flutter надає широкі можливості для створення адаптивного дизайну, що враховує різні розміри екранів і щільність пікселів [□16].

У межах реалізації застосунку була обрана **модульна архітектура**, в основі якої лежить принцип розділення обов'язків (**Separation of Concerns**). Кожен компонент відповідає за окрему частину логіки або інтерфейсу, що значно спрощує масштабування проєкту, внесення змін і підтримку коду в майбутньому [□17].

Основні структурні компоненти клієнтської частини:

- **Екрани (Screens)** — це основні сторінки застосунку, які відповідають за візуальне відображення інформації та взаємодію з користувачем. До них належать: головний екран, галерея одягу, реєстрація/вхід, особистий кабінет, генерація луків, екран з обраними речами тощо. Кожен екран містить специфічну логіку відображення відповідного контенту;
- **Виджети (Widgets)** — повторно використовувані елементи інтерфейсу, такі як кнопки, поля введення, картки одягу, списки, модальні вікна тощо. Вони створюються у вигляді окремих Dart-файлів і забезпечують чистоту основного коду екранів;

- **Контролери (Controllers)** — компоненти, що відповідають за логіку переходів між екранами, обробку натискань, виклик функцій завантаження/збереження даних, а також зв’язок з сервісами. Наприклад, AddClothingController обробляє додавання нового предмета одягу, виклик Remove.bg API, завантаження на Cloudinary та збереження метаданих у Firestore;
- **Сервіси (Services)** — спеціальні класи, що взаємодіють з Firebase (авторизація та база даних), зовнішніми API (Cloudinary, OpenWeatherMap, Remove.bg) та логікою підбору одягу. Усі запити винесено в окремі сервіси, що дає змогу уникнути дублювання коду та спростити тестування функціоналу;
- **Моделі (Models)** — структури даних, що описують об’єкти, з якими працює застосунок. Наприклад, модель ClothingItem включає поля id, назва, категорія, колір, посилання на зображення та використовується для збереження та відображення елементів гардеробу;

Особливості логіки та керування станом:

Керування станом реалізовано вбудованими інструментами Flutter. Для простих сценаріїв використано **Stateful Widgets** із локальним станом. У більш складних випадках — наприклад, під час генерації луків або збереження інформації користувача — застосовується поділ логіки між контролерами та сервісами, які передають оновлення UI через setState() або callback-функції.

Інтерфейс побудовано за принципами **реактивності**: будь-які зміни у вхідних даних автоматично відображаються на екрані, без необхідності повторного створення сторінки або оновлення вручну. Наприклад, після додавання нового предмета одягу до галереї, відповідна частина інтерфейсу миттєво оновлюється.

Додаток також враховує обмеження мобільних пристроїв — мінімізовано кількість запитів до сервера, забезпечено попереднє кешування зображень і використано асинхронні методи (async/await) для уникнення блокування UI.

Завдяки модульній побудові клієнтська частина є легко масштабованою. У разі потреби можуть бути додані нові екрани, нові категорії або фільтри одягу, без суттєвих змін у вже реалізованій структурі.

На рисунку 3.2 зображена діаграма класів.

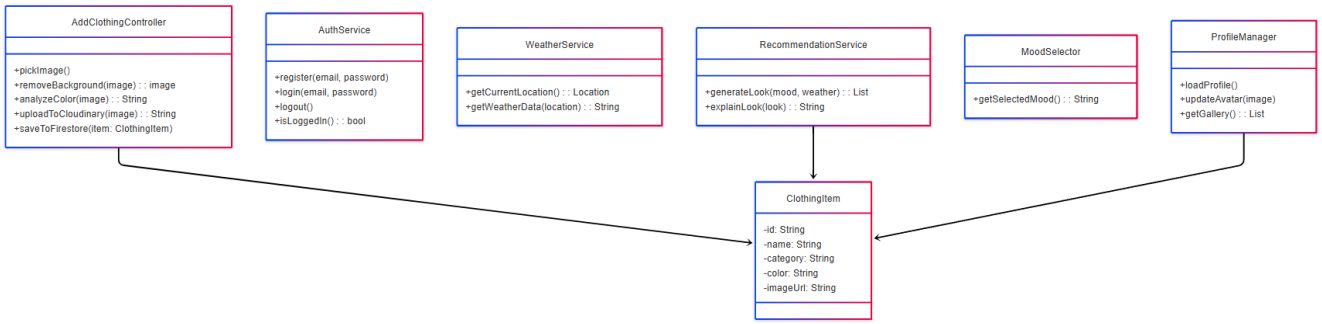


Рис. 3.2 Діаграма класів основних компонентів мобільного застосунку

3.3 Архітектура серверної частини

Серверна частина застосунку реалізована з використанням хмарної платформи **Firebase**, яка надає готові рішення для автентифікації користувачів, зберігання даних, а також базову обробку запитів у режимі реального часу. Обрана архітектура дозволяє зосередитися на логіці взаємодії з даними без необхідності розгортання окремого фізичного сервера чи ручного налаштування бекенду.

У межах реалізації проєкту використовуються два основні сервіси Firebase:

- **Firebase Authentication** — забезпечує авторизацію та реєстрацію користувачів за допомогою електронної пошти та пароля. Після реєстрації кожному користувачу присвоюється унікальний UID, який надалі використовується для зв’язку з відповідними даними в базі;
- **Cloud Firestore** — NoSQL база даних, що використовується для зберігання інформації про користувачів, їхні предмети одягу, категорії, налаштування, а також результати генерації луків. Структура бази є ієрархічною та гнучкою, що дозволяє легко масштабувати систему й додавати нові колекції або поля.

Логіка взаємодії між клієнтом і сервером:

1. **Реєстрація та вхід**: користувач створює обліковий запис, який зберігається у Firebase Auth. UID цього акаунту використовується як ключ до збереження персональних даних у Firestore;
2. **Завантаження одягу**: після додавання нового предмета одягу на клієнтському боці, застосунок надсилає запит до Firestore із даними про назву, категорію, колір, посилання на зображення та UID користувача;
3. **Зчитування галереї**: на головній або профільній сторінці застосунку відбувається запит до колекції Firestore, яка містить предмети одягу, прив'язані до UID поточного користувача;
4. **Генерація луків**: на основі поточної погоди, настрою та доступних речей, логіка генерації формує набір одягу та зберігає його в окремій колекції generated_looks;
5. **Збереження додаткової інформації**: додаткові поля, такі як обране, історія генерацій або колекції настрою, також зберігаються в базі як вкладені документи або додаткові колекції.

Переваги обраної архітектури:

- **Швидке розгортання** — немає потреби створювати окремий сервер або API;
- **Масштабованість** — Firebase автоматично обробляє зростання кількості користувачів та запитів;
- **Безпека** — конфігуровані правила доступу дозволяють обмежити читання/запис даних лише авторизованим користувачам;
- **Реактивність** — застосунок може миттєво отримувати оновлення з Firestore без необхідності перезапуску чи повторного завантаження.

У подальшому до серверної частини може бути підключено окремий хостинг для більш складної логіки обробки даних, наприклад, генерації рекомендацій за допомогою Python або Node.js сервісу.

3.4 Архітектура бази даних

У рамках реалізації застосунку для зберігання даних обрано **Cloud Firestore** — гнучку, масштабовану хмарну NoSQL-базу даних, що працює з ієрархічною структурою: **колекції → документи → підколекції**. Такий формат дозволяє ефективно зберігати структуровану інформацію, пов'язану з конкретними користувачами.

Основні колекції бази даних:

1. **users**
    - Кожен документ у цій колекції має ID, що відповідає UID користувача з Firebase Auth;
    - Основні поля:
        - email — електронна пошта користувача;
        - nickname — нікнейм;
        - avatarUrl — посилання на аватарку;
        - isFirstTime — булевий прапорець для перевірки новачків;
        - moodHistory — масив або вкладена підколекція зі збереженими настроями.
2. **clothing_items**
    - Зберігає предмети одягу, додані користувачами;
    - Основні поля:
        - userId — UID користувача, власника речі;
        - name — назва речі;
        - category — категорія (аксесуари, голова, тіло тощо) ;
        - color — домінантний колір (hex або назва) ;
        - imageUrl — посилання на зображення в Cloudinary;
        - styleTags — список тегів або стиль (опціонально) .
3. **generated_looks**
    - Містить лукі, згенеровані системою за погодою та настроєм;
    - Основні поля:

- userId — UID користувача;
- mood — обраний настрій;
- weather — опис погоди;
- lookItems — масив об’єктів типу ClothingItem;
- explanation — пояснення, чому саме цей лук рекомендовано.

На рисунку 3.3 зображена архітектура бази даних у вигляді діаграми.

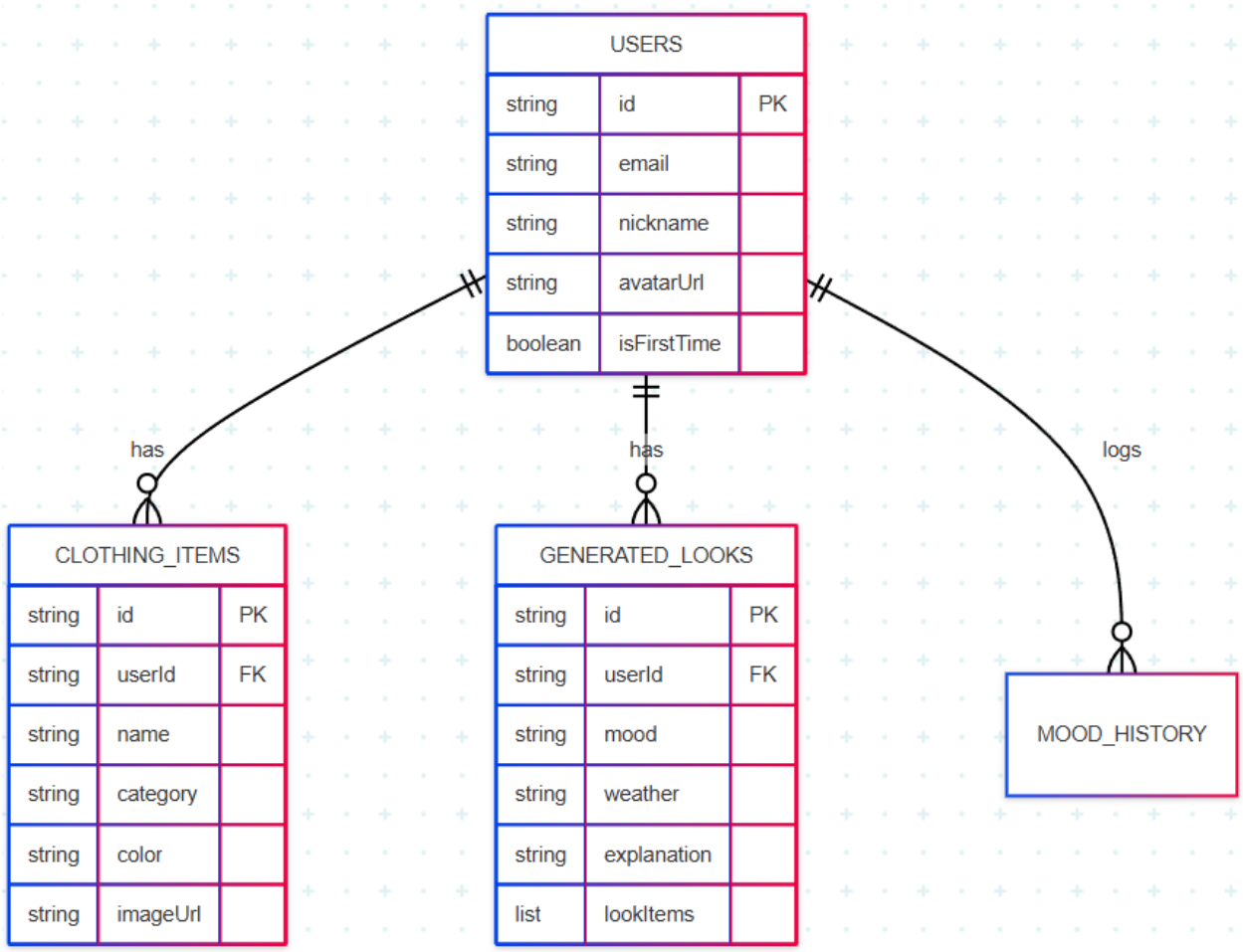


Рис.3.3 ER-діаграма структури даних мобільного застосунку

3.5 Екран входу

На рисунку 3.4 зображено екран входу користувача до мобільного застосунку. Він виконаний у світлому мінімалістичному стилі, без зайвих візуальних елементів, що забезпечує чітке фокусування користувача на основних діях: введенні облікових даних та навігації до реєстрації. Загальна структура екрану реалізована за

допомогою базових віджетів Flutter, зокрема Scaffold, Column, TextFormField, ElevatedButton та GestureDetector.

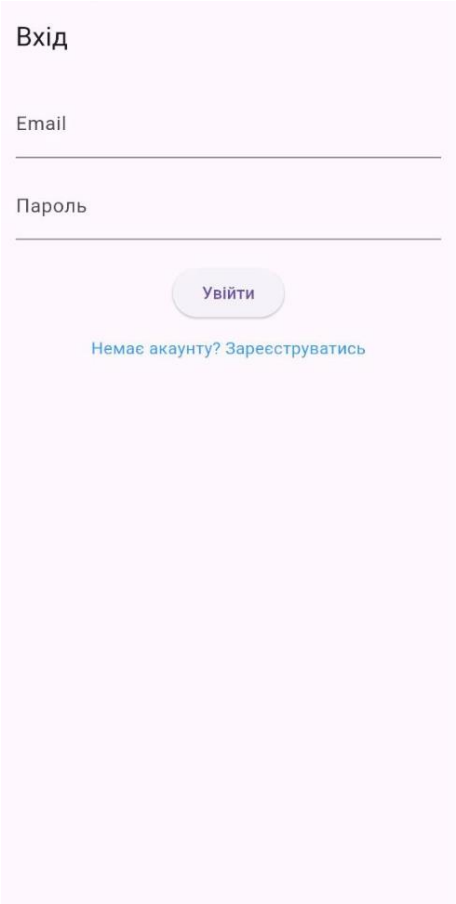


Рис.3.4 Екран входу в застосунок

У верхній частині екрана розташовано заголовок **«Вхід»**, виконаний у стилі Material Design із використанням шрифту підвищеної ваги (fontWeight: FontWeight.bold). Він логічно структурує сторінку та чітко позначає призначення даного екрана.

Далі розміщено два поля вводу:

- **Email** — текстове поле з базовою валідацією та підказкою у вигляді лейблу;
- **Пароль** — поле з прихованим введенням символів для безпеки, реалізоване з можливістю перемикання видимості (опціонально, залежно від конфігурації).

Ці поля оформлені через TextFormField зі стандартним підкресленням (underline border), що відповідає загальному стилю застосунку та не відволікає увагу користувача.

Центральне місце займає кнопка **«Увійти»**, стилізована з м’яким заокругленням країв і ледь помітною тінню. Колір кнопки (світло-фіолетовий) контрастує з білим фоном, але при цьому гармонійно вписується у візуальний стиль програми. При натисканні викликається функція входу, що надсилає запит до **Firebase Authentication** для перевірки правильності введених облікових даних.

У нижній частині сторінки розташовано текстове посилання **«Немає акаунту? Зареєструватись»**, яке веде до сторінки реєстрації. Його реалізовано через Text.rich() із TextSpan та вбудованим GestureDetector для обробки кліку. Акцентний синій колір виділяє цей елемент як інтерактивний.

У випадку помилки (наприклад, невірна пошта або пароль) застосунок відображає відповідне повідомлення про помилку. Усі повідомлення інтегровано в інтерфейс без використання зовнішніх бібліотек, що спрощує підтримку коду та забезпечує стабільність роботи.

Візуальне вирівнювання, пропорції та відступи налаштовано так, щоб екран виглядав збалансовано на більшості розмірів екранів, зокрема смартфонів середнього класу. Загальний вигляд екрана повністю відповідає вимогам до зручності використання та UX-дизайну мобільних застосунків.

3.6 Екран реєстрації

На рисунку 3.5 зображено екран реєстрації нового користувача. Він оформлений у тому ж світлому мінімалістичному стилі, що й екран входу, з дотриманням єдиної візуальної концепції застосунку. Основна мета цього інтерфейсу — забезпечити максимально простий і швидкий процес створення облікового запису з мінімальною кількістю полів.

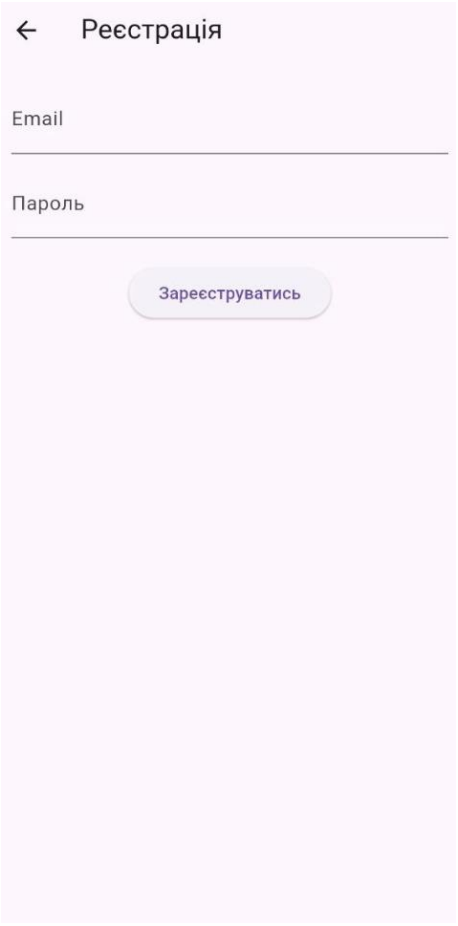


Рис.3.5 Екран “Реєстрація”

У верхній частині екрана розміщено заголовок **«Реєстрація»**, зліва від якого присутня кнопка повернення назад у вигляді стрілки. Її реалізовано за допомогою IconButton у AppBar, що дозволяє повернутися на попередній екран (вхід).

Нижче розташовано два основні поля для введення:

- **Email** — поле для введення електронної адреси користувача. Містить вбудовану валідацію формату email;
- **Пароль** — поле з прихованим введенням символів. Підтримується обмеження на мінімальну довжину пароля, що перевіряється ще до відправки даних на сервер;

Кожне поле реалізовано з використанням стандартного TextFormField, зі збереженням стилістики застосунку: підкреслення замість повної рамки, нейтральний шрифт, світлий фон. Поля мають достатній міжрядковий відступ, що покращує зчитування та зменшує ймовірність помилок при введенні.

Центральним елементом є кнопка **«Зареєструватись»**, стилізована відповідно до загального дизайну (округлі краї, фіолетовий шрифт, світлий фон, тінь для глибини). При натисканні на кнопку відбувається валідація введених даних та викликається функція створення акаунта через **Firebase Authentication**. У разі помилки (наприклад, вже існує обліковий запис із такою поштою), система повертає відповідне повідомлення.

Успішна реєстрація призводить до автоматичного переходу на головну сторінку застосунку. У той же момент система оновлює базу даних (Firestore), додаючи нового користувача в колекцію users з UID, поштою, нікнеймом (якщо реалізовано), аватаром (опціонально) та прапорцем isFirstTime = true.

Всі масштабуються під розмір екрана, що забезпечує коректне відображення як на малих, так і на великих пристроях.

3.7 Головний екран застосунку

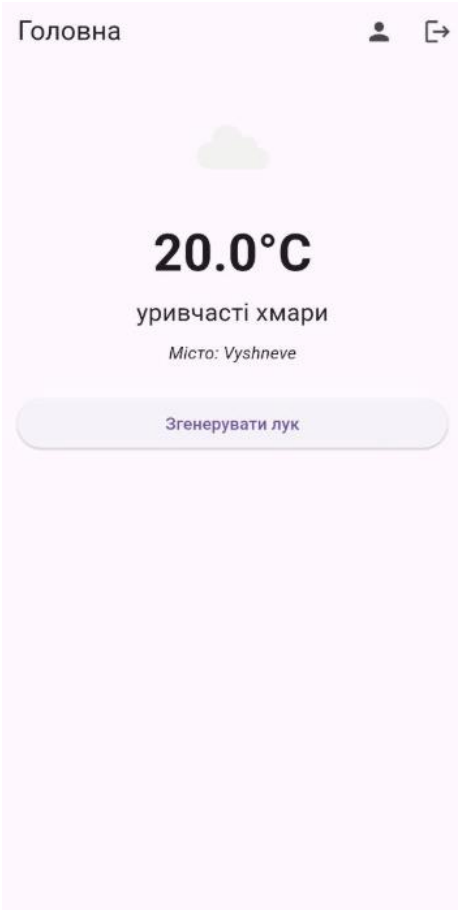


Рис. 3.6 Екран “Головна”

На рисунку 3.6 зображено головний екран застосунку, який є стартовою точкою після входу до системи. Інтерфейс оформлено у світлому стилі з мінімалістичним підходом, де акцент зроблено на найважливішій інформації — поточній погоді, настрої користувача та кнопці генерації луків.

У верхній частині екрана розташовано **AppBar** із назвою розділу «Головна», а також двома іконками праворуч:

- іконка профілю, яка дозволяє перейти до персонального кабінету користувача;
- іконка виходу із застосунку.

Центральну частину екрана займає **блок з погодою**, який динамічно оновлюється на основі геолокації користувача. Для отримання координат використовується Flutter-пакет geolocator, а для підвантаження погодних умов — API сервісу **OpenWeatherMap**.

На основі отриманої відповіді на екрані виводиться:

- температура у градусах Цельсія (наприклад, 20.0°C) ;
- короткий опис стану атмосфери (наприклад, «уривчасті хмари»);
- назва населеного пункту (наприклад, Vyshneve) ;
- а також піктограма погоди, яка динамічно змінюється залежно від погодного коду.

Однією з важливих особливостей головного екрана є можливість **оновлення даних про погоду свайпом вниз**. Для цього використовується віджет RefreshIndicator, який дозволяє вручну оновити стан сторінки без перезапуску застосунку.

Після завантаження погодних даних застосунок перевіряє, чи збережено у Firestore останній обраний **настрій користувача**. Якщо ні, відображається **діалогове вікно з вибором настрою**, яке зображено на рисунку 3.7.

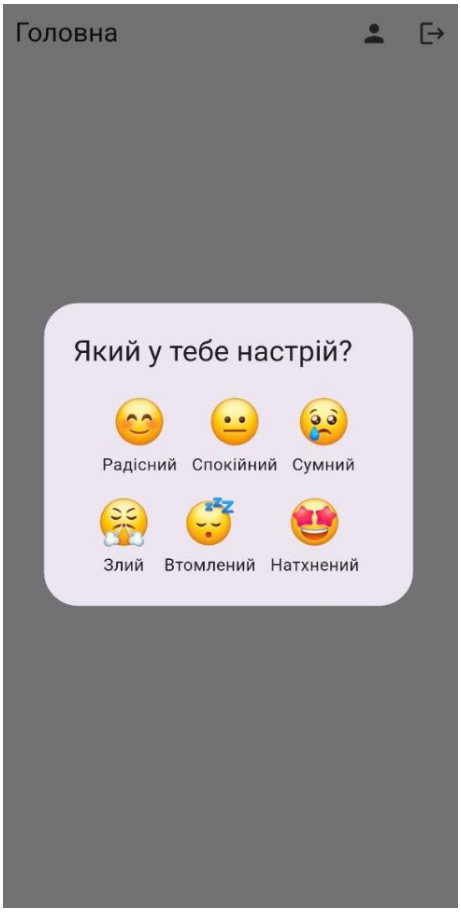


Рис.3.7 Екран “Головна” з діалоговим вікном для вибору настрою

У ньому користувач може обрати свій емоційний стан, натиснувши на одну з шести кнопок з emoji:

- Радісний ;
- Спокійний ;
- Сумний ;
- Злий ;
- Втомлений ;
- Натхненний .

Кожен із цих варіантів впливає на логіку генерації луків у RecommendationService, змінюючи пріоритет стилів та кольорів обраного одягу. Вибір настрою зберігається до Firestore і повторно не запитується, доки користувач самостійно не захоче змінити свій стан.

Під блоком настрою або погоди розміщено кнопку **«Згенерувати лук»**, яка є основною дією цього екрана. Вона активує сервіс рекомендацій, який враховує

погоду, настрій та доступні елементи гардеробу з бази даних. У результаті користувач отримує підбір щоденного образу, який може зберегти або оновити.

Візуальний стиль сторінки витримано у світлій палітрі зі спокійними кольорами, що сприяє комфортній взаємодії. Усі елементи інтерфейсу адаптивні, використовують MediaQuery для коректного масштабування, а структура компонувань побудована на базі Column, Padding та кастомних виджетів.

3.8 Екран профілю користувача

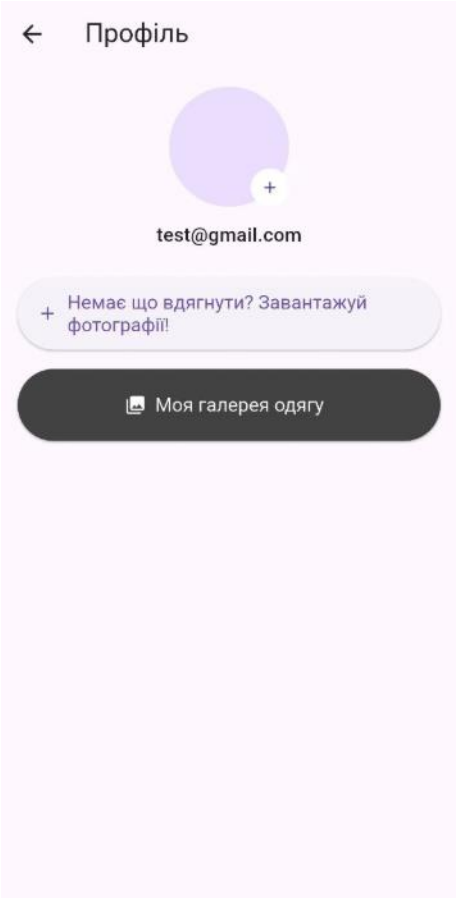


Рис.3.8 Екран “Профіль”

На рисунку 3.8 зображено екран профілю, який надає користувачу доступ до основних функцій керування акаунтом, перегляду особистої інформації та завантаження власного гардеробу.

У верхній частині інтерфейсу розташовано **AppBar** з кнопкою повернення назад (у вигляді стрілки), заголовком **«Профіль»**, вирівняним ліворуч, та мінімалістичним оформленням, що не відволікає увагу від основного контенту.

Центральний елемент екрана — **аватар користувача**, реалізований у вигляді кола (CircleAvatar). За замовчуванням відображається стилізована фіолетова заливка, яка може бути замінена на фото користувача. У нижній частині аватарки знаходиться кнопка з піктограмою «+», яка дозволяє змінити або завантажити нове зображення. Натискання на цю кнопку викликає функцію вибору зображення з галереї або камери, а після вибору — автоматичне збереження у Firestore та оновлення інтерфейсу.

Під аватаром виводиться **email користувача**, отриманий через Firebase Authentication. Цей елемент є лише для перегляду і не підлягає редагуванню без додаткової автентифікації.

Нижче розташована кнопка у вигляді промо-блоку з текстом **«Немає що вдягнути? Завантажуй фотографії!»**, що заохочує користувача поповнити свій гардероб. Кнопка стилізована у вигляді блоку з заокругленими краями та легкою тінню, реалізована за допомогою InkWell або ElevatedButton з кастомним оформленням. Вона викликає функцію переходу до форми додавання нового предмета одягу.

Основна функціональна кнопка **«Моя галерея одягу»** знаходиться в нижній частині екрана. Вона дозволяє користувачу переглянути збережені речі, які надалі можуть бути використані для генерації луків. Кнопка темного кольору з іконкою зображення, розміщеною ліворуч, має чіткий візуальний акцент і логічно завершує структуру сторінки. Перехід здійснюється до окремого екрана галереї, де відображаються категоризовані речі.

Інтерфейс екрану є адаптивним, усі елементи рівномірно розміщені, а кольорова схема відповідає загальному стилю застосунку.

3.9 Екран додавання одягу

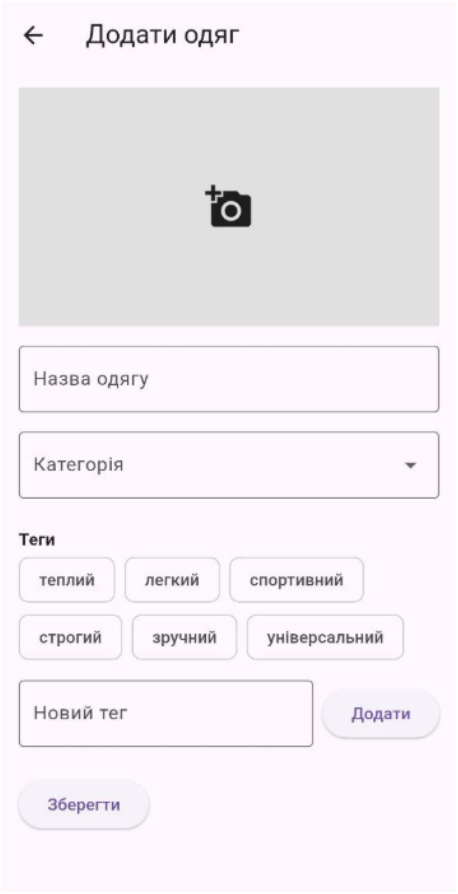


Рис. 3.9 Екран “Додати одяг”

На рисунку 3.9 представлено екран, який дозволяє користувачу поповнювати свій цифровий гардероб, додаючи фотографії елементів одягу. Інтерфейс цієї сторінки побудовано у звичному для всього застосунку світлому стилі з інтуїтивно зрозумілими полями та кнопками.

У верхній частині розташовано **AppBar** з назвою «Додати одяг» та кнопкою повернення до попереднього екрана. Основний вміст сторінки структуровано вертикально з логічним розташуванням елементів — від зображення до підтвердження.

Центральне місце займає **порожній контейнер із піктограмою камери**, яка сигналізує про можливість додати фото. При натисканні на цей блок користувачеві пропонується обрати зображення з галереї або зробити фото камерою.

Після вибору зображення відбувається:

1. **Видалення фону** через API Remove.bg;
2. **Збереження очищеного фото** на хмарному сховищі Cloudinary;
3. **Додавання метаданих** до Firestore.

Нижче розміщені два поля введення:

- **назва одягу** — просте текстове поле (TextFormField), де користувач вводить умовну назву речі;
- **Категорія** — випадаючий список (DropdownButtonFormField), який містить попередньо визначені варіанти: голова, тіло, ноги, аксесуари тощо.

Далі йде блок із тегами, які допомагають краще описати стиль чи призначення речі. Частину тегів запропоновано за замовчуванням (наприклад, «теплий», «зручний», «універсальний»). Також користувач може **додати власний тег**, ввівши його в текстове поле та натиснувши кнопку **«Додати»**. Усі вибрані теги зберігаються разом із речами у Firebase.

На завершення сторінки розташована кнопка **«Зберегти»**, яка активує процес перевірки заповнених даних і надсилає інформацію до бази даних. Якщо користувач не заповнив обов’язкові поля або не додав фото, виводиться відповідне повідомлення про помилку.

Інтерфейс реалізовано з використанням віджетів Form, TextFormField, DropdownButton, Wrap (для тегів) та ElevatedButton. Весь процес додавання — від вибору зображення до збереження у Firestore — відбувається в одному екрані, без додаткових переходів, що спрощує взаємодію.

3.10 Екран галереї одягу

На рисунку 3.10 зображено екран «Моя галерея одягу», який дозволяє користувачу переглядати, систематизувати та взаємодіяти з усіма збереженими елементами гардеробу. Інтерфейс цієї сторінки побудовано у вигляді сітки, де кожен елемент представлений у формі картки з візуальним та текстовим описом.

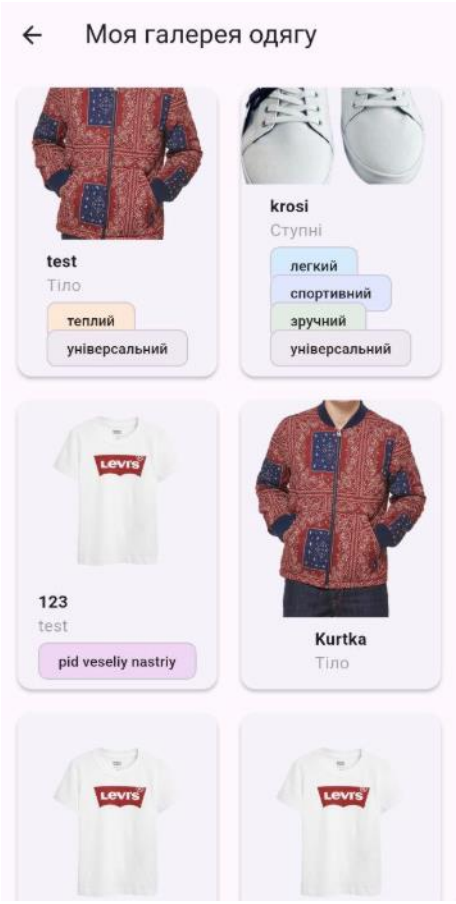


Рис.3.10 Екран “Моя галерея одягу”

У верхній частині розміщується **AppBar** із кнопкою повернення назад, заголовком сторінки та витриманим стилем оформлення, який відповідає загальній тематиці застосунку.

Основний вміст екрану реалізовано у вигляді **сітки (GridView.builder)**, яка адаптивно підлаштовується під розміри екрана та кількість елементів.

Кожна картка одягу містить:

- зображення речі (із вже видаленим фоном) ;
- назву речі;
- категорію (наприклад, «Тіло», «Ступні»);
- список тегів, які характеризують стиль або особливість одягу;

Усі дані завантажуються з бази Firestore за UID поточного користувача. Після завантаження список сортується за категоріями для зручності навігації.

Картка одягу реалізована як кастомний виджет, що включає в себе:

- Container із тінню та заокругленням кутів;

- Image.network() для зображення;
- Text() для виводу назви та категорії;
- Wrap для відображення тегів у вигляді кольорових блоків (Chip або Container зі стилізацією).

Користувач може скролити галерею, щоб переглядати весь вміст, а в майбутньому — натискати на картки для редагування або видалення речей.

Інтерфейс галереї плавно працює на різних розмірах екранів. У разі відсутності одягу на сторінці відображається повідомлення з пропозицією додати новий елемент через відповідну кнопку на екрані профілю.

3.11 Екран згенерованого луку

На рисунку 3.11 представлено екран, на якому користувач бачить сформований образ (лук) на основі поточної погоди, настрою та наявних у гардеробі речей. Цей екран є логічним продовженням головної сторінки, яка ініціює процес генерації.

Інтерфейс побудовано за принципом вертикального списку категорій одягу. Для кожної з них відображається підібраний елемент (за наявності) або повідомлення **«Немає»** у разі відсутності відповідної речі в гардеробі користувача.

Категорії представлені у фіксованому порядку:

- Голова;
- Тіло;
- Ноги;
- Ступні;
- Аксесуари.

Кожен блок містить:

- Назву категорії;
- Зображення елемента одягу або хрестик із написом **«Немає»**;
- Кнопки навігації вліво/вправо (IconButton) для ручного переключення елементів в межах категорії.

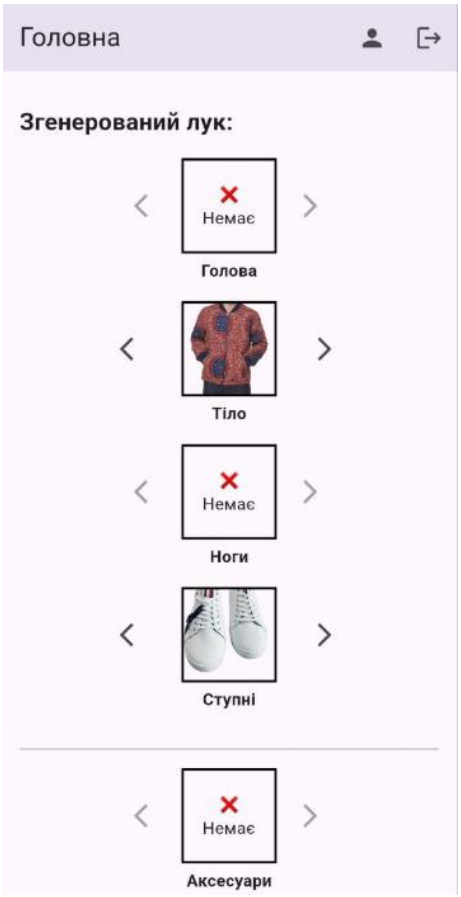


Рис.3.11 Екран “Головна” зі згенерованим “луком”

Функціональність прокрутки реалізована через **горизонтальний список речей у кожній категорії**, що дозволяє користувачеві **самостійно змінити окремі елементи образу** — наприклад, вибрати іншу куртку чи взуття. Зміни не потребують повторного запуску генерації, що забезпечує гнучкість і зручність редагування згенерованого луку.

Кожна картка одягу має рамку (Container з border), що підкреслює її активний стан. У разі відсутності речей у категорії, відображається квадрат із червоним хрестиком і написом «Немає» — такий візуальний підхід чітко сигналізує про нестачу елементів у гардеробі.

Цей екран також підтримує **реактивне оновлення**: якщо користувач переходить до профілю, додає нову річ, а потім повертається — інформація про гардероб оновлюється автоматично завдяки зверненню до Firestore.

4 ТЕСТУВАННЯ

4.1 Тестування

Щоб застосунок працював так, як задумано, важливо переконатися, що всі його частини — як окремо, так і разом — функціонують стабільно, без помилок і не викликають труднощів у користувача. Саме тому після реалізації основної логіки було проведено повноцінне тестування. Воно охопило всі ключові функції: від взаємодії з користувачем до звернень до зовнішніх сервісів, обробки виняткових ситуацій та коректного збереження даних.

У процесі тестування було використано кілька підходів — кожен з них мав свою роль і дозволяв оцінити систему з різних сторін. Це дало змогу виявити як логічні помилки в окремих функціях, так і проблеми у взаємодії між компонентами [□18].

Системне тестування

Цей тип перевірки охоплює **повну функціональність застосунку** — від моменту його запуску до закриття. У цьому випадку йдеться не про окремі частини, а про все як єдине ціле. Перевірялися всі ключові дії, які виконує звичайний користувач: реєстрація, вхід, додавання одягу до гардеробу, взаємодія з головною сторінкою, вибір настрою, генерація луку, перегляд згенерованого образу

Системне тестування охоплює повну функціональність застосунку — від моменту його запуску до закриття. Такий тип перевірки відповідає міжнародному підходу до валідації згідно зі стандартами якості [□19].

Окремо тестувалась поведінка застосунку при відсутності інтернету, при нестачі обов’язкових даних (наприклад, якщо у гардеробі не вистачає речей), а також реакція на некоректні дії користувача (наприклад, збереження без фото чи без назви).

Також було протестовано поведінку застосунку на різних розмірах екранів та пристроях — щоб переконатись, що інтерфейс не “ламається”, а всі функції залишаються доступними та зручними.

Інтеграційне тестування

Коли окремі частини працюють правильно, важливо перевірити, **як вони взаємодіють між собою**. Наприклад, навіть якщо модуль генерації образу працює окремо, а погода відображається правильно — це не гарантує, що при їх поєднанні результат буде коректним.

Тому були протестовані всі ключові зв’язки:

- генерація луку на основі погоди та настрою;
- взаємодія між базою даних Firestore і галереєю одягу;
- обробка API-відповідей сервісів OpenWeatherMap, Remove.bg та Cloudinary;
- збереження та відображення результатів генерації у відповідному UI-блоці.

Таке тестування дозволило виявити, наприклад, що при повільному інтернет-з’єднанні завантаження погоди може не встигати до генерації луку — і це вимагало асинхронного контролю станів.

Юніт-тестування

Юніт-тести — це перевірка **окремих функцій або методів**, ізольовано від решти застосунку. У рамках цього проєкту перевірялися, наприклад:

- функція генерації луку з певного набору речей;
- логіка сортування одягу за категоріями;
- фільтрація гардеробу за користувачем;
- обробка даних погоди (перетворення коду в опис чи іконку).

Юніт-тести реалізовувались у Flutter за допомогою пакета test. Для підміни зовнішніх залежностей застосовувались моки, щоб не прив’язуватись до реальної бази чи API.

Перевага таких тестів у тому, що їх можна запускати часто, і вони швидко вказують на помилки при зміні окремих частин логіки. Це особливо важливо при розширенні функціоналу або оновленні структури даних.

Приймальне тестування

На фінальному етапі, коли функціонал уже стабільний, було проведено **імітацію поведінки реального користувача**.

Перевірялись типові сценарії:

- новий користувач заходить і не має жодного одягу;
- користувач одразу намагається згенерувати лук, не вибравши настрій;
- частина речей відсутня, і генерація повинна обрати хоча б те, що є;
- користувач додає аватар і редагує свій гардероб.

Приймальне тестування дозволило подивитись на застосунок очима користувача, не як розробника, і виявити речі, які раніше не помічались. Наприклад, зручність підказок, швидкість відгуку інтерфейсу, логічність розміщення кнопок.

4.2 Тестування в застосунку

Клієнтська частина застосунку реалізована на Flutter, що дозволяє зручно проводити модульне тестування інтерфейсів за допомогою flutter_test. Було протестовано екран авторизації, який відповідає за вхід користувача до системи. Метою тестування є перевірка наявності ключових елементів інтерфейсу та їх правильне відображення.

Наведений нижче тест перевіряє:

- заголовок сторінки;
- поля введення email і пароля;
- кнопку входу;
- посилання на сторінку реєстрації.

Клієнтська частина застосунку реалізована на Flutter, що дозволяє зручно проводити модульне тестування інтерфейсів за допомогою flutter_test. Було протестовано екран авторизації, який відповідає за вхід користувача до системи. Метою тестування є перевірка наявності ключових елементів інтерфейсу та їх правильне відображення.

Наведений нижче тест перевіряє:

- заголовок сторінки;

- поля введення email і пароля;
- кнопку входу;
- посилання на сторінку реєстрації.

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:weather_mood/features/auth/presentation/pages/login_page.dart';
void main() {
  testWidgets('LoginPage shows all expected UI elements', (WidgetTester tester)
async {
    await tester.pumpWidget(
const MaterialApp(
  home: LoginPage(),
),
    );
    // Перевірка заголовка
    expect(find.text('Вхід'), findsOneWidget);
    // Поля вводу
    expect(find.byType(TextField), findsNWidgets(2));
    expect(find.widgetWithText(TextField, 'Email'), findsOneWidget);
    expect(find.widgetWithText(TextField, 'Пароль'), findsOneWidget);
    // Кнопка входу
    expect(find.widgetWithText(ElevatedButton, 'Увійти'), findsOneWidget);
    // Посилання на реєстрацію
    expect(find.text('Немає акаунту? Зареєструватись'), findsOneWidget);
  });
}
```

Пояснення до тесту:

– *pumpWidget* ініціалізує сторінку LoginPage у тестовому середовищі.

– *expect(find.text('Вхід')...)* перевіряє, чи відображається заголовок.

– *find.byType(TextField)* перевіряє наявність двох текстових полів.

– *widgetWithText(...)* переконується, що кожне поле має відповідну мітку. – Перевірка кнопки *Увійти* і посилання *Немає акаунту? Зареєструватись* гарантує, що інтерфейс містить усі важливі елементи.

На рисунку 4.1 зображений результат тестування.

```
PS D:\weather_mood> flutter test test/login_page_test.dart
00:03 +1: All tests passed!
```

Рис.4.1 Успішне проходження тестів

Як видно з результату, тест успішно виконується, що свідчить про коректну побудову інтерфейсу логіну. Такий підхід дозволяє вчасно виявляти помилки при зміні UI та підтримувати високу якість взаємодії з користувачем.

4.3 Завантаження проєкту на GitHub

Для зберігання мобільного застосунку у віддаленому сховищі та забезпечення зручної співпраці та резервного копіювання, проєкт було завантажено на платформу GitHub [20]. Репозиторій містить весь код застосунку, включаючи Flutter-інтерфейс, логіку взаємодії з Firebase та систему генерації образів (луків) на основі погоди й настрою.

Щоб додати проєкт до репозиторію GitHub, було виконано стандартні кроки:

1. **Ініціалізація Git-репозиторію**

   У кореневій папці проєкту було виконано команду:

   git init

2. **Додавання віддаленого репозиторію**

   Було вказано URL-адресу репозиторію, створеного на GitHub:

   git remote add origin https://github.com/FoxyArt1/weather_mood.git

3. **Індексація файлів**

   Всі файли проєкту додано до індексу:

git add .

4. **Створення першого коміту**

Із коротким описом основної функціональності:

git commit -m "first commit"

5. **Завантаження проєкту у віддалений репозиторій**

Відправлення локального репозиторію на GitHub:

git push -u origin master

У результаті виконання цих дій, весь код проєкту став доступним у віддаленому репозиторії GitHub, що дозволяє зручно переглядати, змінювати та документувати історію змін. Окрім основної гілки, у проєкті можуть бути створені додаткові гілки для розробки нових функцій, виправлення помилок або експериментів

ВИСНОВКИ

Результатом виконаної роботи стала розробка мобільного застосунку для підбору одягу відповідно до поточної погоди та настрою користувача. Реалізація проєкту здійснювалась за допомогою мови Dart та фреймворку Flutter, з використанням сервісів Firebase і сторонніх API.

У ході виконання проєкту було реалізовано такі завдання:

– **Проаналізовано предметну область**: визначено проблему щоденного вибору одягу, охарактеризовано поведінку потенційних користувачів та виділено ключові критерії впливу (погода, настрій, стиль одягу). Проведено аналіз аналогічних застосунків та сформульовано функціональні вимоги, серед яких: авторизація, створення віртуального гардеробу, категоризація одягу, збереження зображень, видалення фону, отримання погоди та генерація луку на основі настрою і погодних умов.

– **Обрано інструменти розробки**: Flutter як основна технологія для створення застосунку, Firebase Authentication та Cloud Firestore як засоби автентифікації та зберігання даних, Cloudinary для зберігання зображень, Remove.bg API для обробки фото, OpenWeatherMap API для отримання погодних даних, GitHub для контролю версій, Android Studio як середовище розробки.

– **Розроблено архітектуру застосунку**: створено класову модель, визначено основні екрани, структуру каталогів, API-сервіси, а також реалізовано систему рекомендацій з підтримкою обробки кольору, категорій та настрою.

– **Створено функціональний застосунок**: реалізовано інтерфейс з підтримкою авторизації, додаванням одягу до галереї, автоматичним визначенням категорії, відображенням поточної погоди, генерацією луків, а також можливістю редагування образу вручну.

– **Проєкт протестовано та завантажено на GitHub**: виконано тестування основних екранів і логіки авторизації, а також налаштовано систему контролю версій з доступом до всієї історії змін у відкритому репозиторії.

Робота пройшла апробацію.

За її результатами було опубліковано наступні тези доповідей:

1. Сікалюк М.В. Треньов М.Г. Апаратне і програмне забезпечення інформаційних технологій. Інформаційні технології – 2025: зб. тез XII Всеукраїнської науково-практичної конференції молодих учених, 15 трав. 2025р., м. Київ / Київський столичний університет імені Бориса Грінченка, С. 195-196
2. Сікалюк М.В. Треньов М.Г. Визначення вимог до мобыльного додатку для підбору одягу в залежності від погоди та настрою. Зб. тез VI Міжнародна науково-технічна “Сучасний стан та перспективи розвитку IoT”, 15 квітня 2025р., м. Київ / Державний університет інформаційно-комунікаційних технологій, С. (подано до друку)

ПЕРЕЛІК ПОСИЛАНЬ

1. Rouse, M. (2021). App Store (Apple App Store). TechTarget. Retrieved from: https://www.techtarget.com/whatis/definition/App-Store-Apple-App-Store

2. Statista. (2023). Most popular global mobile messenger apps as of January 2023, based on number of monthly active users. Retrieved from: https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

3. Sannacode. (2021). Flutter vs React Native vs Xamarin: what to choose in 2021? Retrieved from: https://sannacode.com/blog/flutter-vs-react-native-vs-xamarin/

4. Smith, J. (2021). Top 10 Closet Apps to Try in 2021. Stylecraze. Retrieved from: https://www.stylecraze.com/articles/best-closet-apps/

5. Raluca Budiu & Kate Moran. (2020). Mobile UX: Best Practices. Nielsen Norman Group. Retrieved from: https://www.nngroup.com/articles/mobile-ux/

6. Singhal, A. (2022). MVVM Architecture in Flutter. Medium. Retrieved from: https://medium.com/flutter-community/mvvm-architecture-in-flutter-31d27b1142c4

7. Raouf, A. (2022). Why Flutter Is the Future of Mobile App Development. Toptal. Retrieved from: https://www.toptal.com/flutter/flutter-future-mobile-apps

8. Hiller, J. (2023). Programming with Dart: 2023 Edition. O'Reilly Media.

9. Moroney, L. (2020). The Firebase Book: The official guide. Google Cloud Press.

10. Firebase. (2023). Get started with Cloud Firestore. Retrieved from: https://firebase.google.com/docs/firestore/quickstart

11. Cloudinary. (2023). Image and Video Management API Documentation. Retrieved from: https://cloudinary.com/documentation/image_upload_api_reference

12. Remove.bg. (2024). Remove Background API Documentation. Retrieved from: https://www.remove.bg/api

13. OpenWeather. (2023). Weather API – OpenWeatherMap. Retrieved from: https://openweathermap.org/api

14. Google Developers. (2023). Geolocation API overview. Retrieved from: https://developers.google.com/maps/documentation/geolocation/overview

15. Nielsen, J. (2021). Usability Heuristics for User Interface Design. Nielsen Norman Group. Retrieved from: https://www.nngroup.com/articles/ten-usability-heuristics/

16. Norman, D. (2020). The Design of Everyday Things: Revised and Expanded Edition. Basic Books.

17. Кожем'яка О. А. (2021). Інтерфейс користувача: підхід до проектування та тестування. Київ: Вид-во НТУУ «КПІ».

18. Канєвський В. М. (2022). Методи тестування програмного забезпечення: Навчальний посібник. Львів: ЛНУ імені Івана Франка.

19. ISO/IEC/IEEE 29119-1:2021. Software and systems engineering – Software testing – Part 1: Concepts and definitions. International Organization for Standardization.

20. GitHub. (2025). GitHub REST API documentation. Retrieved from: https://docs.github.com/rest

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка мобільного застосунку для полегшення підбору одягу в залежності від погоди та настрою

Виконав студент 4 курсу
групи ПД-44
Сікалюк Михайло Васильович
Керівник роботи
Асистент кафедри ІПЗ Треньов Микита Георгійович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета - спростити процес підбору одягу шляхом створення мобільного застосунку
Об'єкт - процес вибору повсякденного або тематичного одягу.
Предмет - програмне забезпечення для підбору одягу на основі погодних умов та емоційного стану користувача.

2

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Проаналізувати існуючі рішення у сфері персоналізованих рекомендацій одягу.
2. Визначити вимоги до мобільного застосунку.
3. Дослідити доступні API для отримання даних про погоду
4. Вибрати відповідні інструменти для реалізації застосунку.
5. Програмно реалізувати функціональний прототип мобільного застосунку.
6. Тестування розробленного ПЗ

3

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Реєстрація та автентифікація користувачів
- Обробка погодних даних та геолокації
- Генерація рекомендацій з одягу на основі оброблених даних
- Інтеграція з галереєю для створення власного віртуального гардеробу

Нефункціональні вимоги:

- Сумісність із операційною системою Android
- Захист персональних даних користувача

АНАЛІЗ АНАЛОГІВ

Характеристика	Pinterest	Cladwell	Smart Closet	Zalando	WeatherMood
Адаптація до погоди	-	+	-	+	+
Урахування настрою	-	-	-	-	+
Рекомендація одягу на основі галереї	-	+	+	+	+
Автоматичне збереження образів	-	+	+	-	+
Мінімалістичний інтерфейс	+	+	+	+	+
Персоналізація (настрій + лук + погода)	-	-	-	-	+

5

ПРОГРАМНІ ЗАСОБИ ДЛЯ РЕАЛІЗАЦІЇ

Flutter

Dart

AndroidStudio

Firebase

OpenWeatherMap

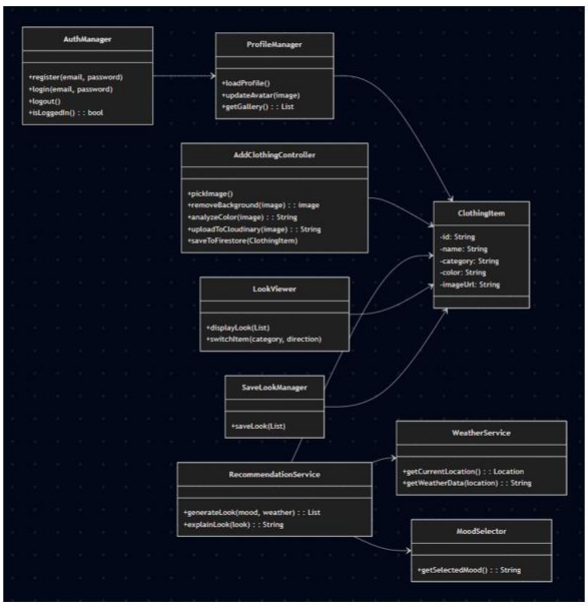
RemoveBg

6

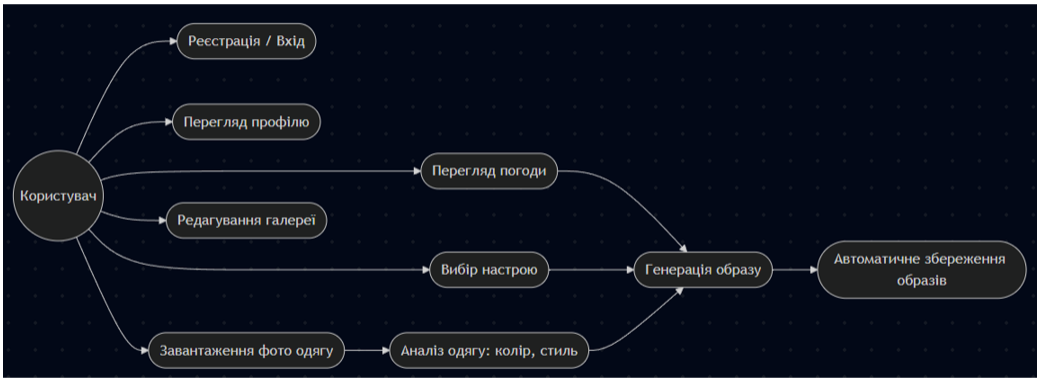
ПЕРЕЛІК ФУНКЦІЙ ПРОГРАМИ

1. AuthManager — керує реєстрацією, входом користувача та перевіркою сесії.
2. ProfileManager — завантажує/редагує аватар, нікнейм та показує галерею речей.
3. AddClothingController — обробляє додавання нового одягу
4. WeatherService — отримує геолокацію та погодні дані через OpenWeatherMap API.
5. MoodSelector — фіксує настрій користувача для підбору луку.
6. RecommendationService — формує образ.
7. LookViewer — виводить згенерований лук і дає змогу гортати елементи.
8. SaveLookManager — дозволяє зберегти образ у галерею.
9. NavigationManager — керує переходами між екранами

ДІАГРАМА КЛАСІВ



ДІАГРАМА ПРЕЦЕНДЕНТІВ

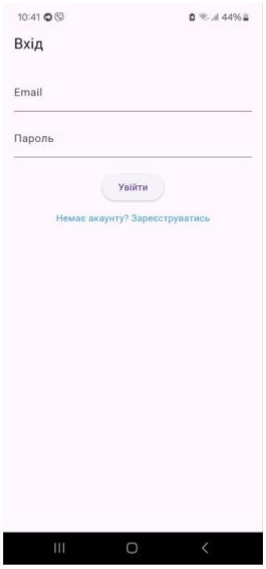


9

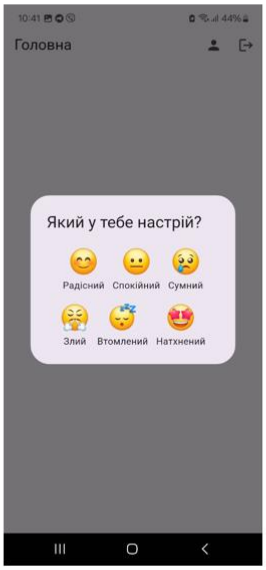
ДІАГРАМА АКТИВНОСТЕЙ

10

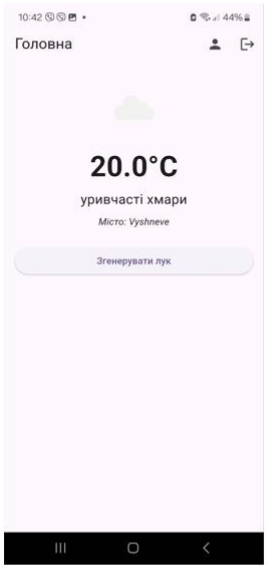
ЕКРАННІ ФОРМИ



Сторінка логіну

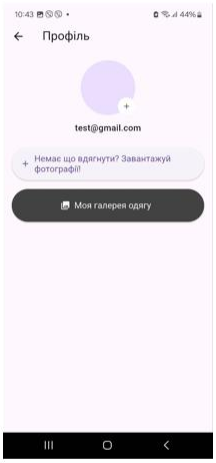


Головна: вибір настрою

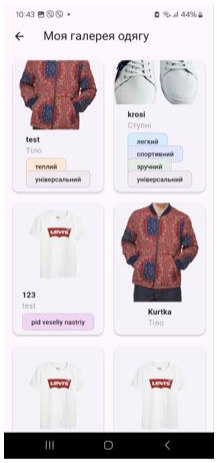


Головна: погода та генерація луку

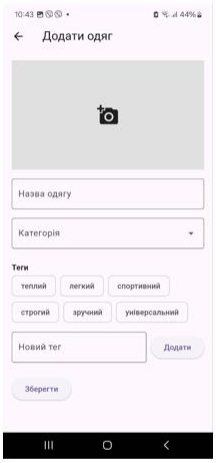
ЕКРАННІ ФОРМИ



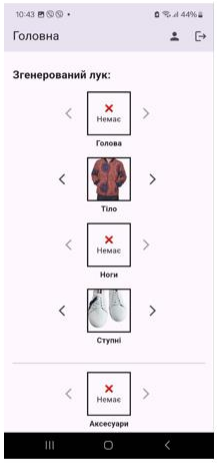
Сторінка профілю



Галерея одягу



Сторінка додавання одягу



Головна: згенерований лук

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Сікалюк М.В. Треньов М.Г. Апаратне і програмне забезпечення інформаційних технологій. Інформаційні технології – 2025: зб. тез XII Всеукраїнської науково-практичної конференції молодих учених, 15 трав. 2025р., м. Київ / Київський столичний університет імені Бориса Грінченка, с. 195-196

14

ВИСНОВКИ

1. Проаналізовано існуючі мобільні застосунки для вибору одягу, відповідно погодним та персональним рекомеданціям
2. Здійснено огляд архітектурних підходів до побудови застосунків
3. Визначено функціональні та нефункціональні вимоги до застосунку
4. Реалізовано базовий функціонал: реєстрацію, авторизацію, додавання речей до гардеробу, інтеграцію з OpenWeatherMap, Remove.bg, Cloudinary
5. Розроблено алгоритм генерації “луку” з урахуванням погоди та настрою
6. Проведено тестування застосунку на відповідність функціональним та нефукнціональним вимогам

15

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Код home_page.dart

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:geolocator/geolocator.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:weather_mood/services/weather_service.dart';
import
'package:weather_mood/features/recommendation/recommenda
tion_service.dart';

class HomePage extends StatefulWidget {
  const HomePage({super.key});

  @override
  State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  List<Map<String, dynamic>> _allItems = []; // усі речі юзера
  List<Map<String, dynamic>>? _recommendedItems;
  late Future<Map<String, dynamic>> _weatherFuture;
  bool _showMoodDialog = false;
  bool _showLookUI = false;
  Map<String, int> _categoryIndexes = {};


  final List<Map<String, String>> _moods = [
    {'emoji': '😊', 'label': 'Радісний'},
    {'emoji': '😐', 'label': 'Спокійний'},
    {'emoji': '😢', 'label': 'Сумний'},
    {'emoji': '😠', 'label': 'Злий'},
    {'emoji': '😴', 'label': 'Втомлений'},
    {'emoji': '🤩', 'label': 'Натхнений'},
  ];

  String? _aiExplanation;

  void _generateAiExplanation(Map<String, dynamic>
weather) {
    if (_recommendedItems == null ||
_recommendedItems!.isEmpty) return;

    final temp = weather['temp']?.toStringAsFixed(0) ?? '?';
    final desc = weather['description'] ?? 'незрозуміла погода';
    final mood = _selectedMood ?? 'невідомий настрій';

    final List<String> keywords = [];

    for (var item in _recommendedItems!) {
final tags = item['tags'];
if (tags != null && tags is List) {
  keywords.addAll(tags.map((e) => e.toString()));
}
    }

    final tagSummary = keywords.toSet().join(', ');

    _aiExplanation =
    'Я обрав цей лук, бо сьогодні $desc і температура
$temp°C. '
  'Ці речі мають такі властивості: $tagSummary. '
  'Вони чудово пасують до твого настрою: $mood.';
  }


  String? _selectedMood;

  @override
  void initState() {
    super.initState();
    _weatherFuture =
_checkLocationPermissionAndFetchWeather();

    final user = FirebaseAuth.instance.currentUser;
    WidgetsBinding.instance.addPostFrameCallback((_) {
if (user != null) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
content: Text('Привіт, ${user.email}!'),
behavior: SnackBarBehavior.floating,
margin: const EdgeInsets.all(16),
duration: const Duration(seconds: 3),
    ),
  );
  _checkIfNeedMood();
}
    });
  }

  void _checkIfNeedMood() {
    setState(() => _showMoodDialog = true);
  }

  Future<void> _showMoodSelectionDialog() async {
    await showDialog(
context: context,
barrierDismissible: false,
builder: (context) => AlertDialog(
  title: const Text('Який у тебе настрій?'),
  content: Wrap(
    spacing: 12,
    runSpacing: 12,
    alignment: WrapAlignment.center,
    children: _moods.map((mood) {
```

```
return GestureDetector(
  onTap: () {
    Navigator.pop(context);
    setState(() {
_selectedMood = mood['label']!;
_showLookUI = true;
    });
  },
  child: Column(
    mainAxisSize: MainAxisSize.min,
    children: [
Text(mood['emoji']!, style: const TextStyle(fontSize: 36)),
Text(mood['label']!),
    ],
  ),
);
    }).toList(),
  ),
),
    );
  }

  Future<void> _generateLook() async {
    final user = FirebaseAuth.instance.currentUser;
    if (user == null || _selectedMood == null) return;

    final querySnapshot = await FirebaseFirestore.instance
  .collection('clothes')
  .doc(user.uid)
  .collection('items')
  .orderBy('createdAt', descending: true)
  .get();

    final items = querySnapshot.docs.map((doc) {
final data = doc.data();
data['id'] = doc.id;
return data;
    }).toList();
    _allItems = items; // зберігаємо всі речі юзера
    _categoryIndexes.clear();
    for (var category in ['Голова', 'Тіло', 'Ноги', 'Ступні',
'Аксесуари']) {
_categoryIndexes[category] = 0;
    }


    final weather = await
WeatherService.fetchWeatherByLocation(
latitude: 50.4501,
longitude: 30.5234,
    );

    final recommended = await
RecommendationService.generateLookFromItems(
items: items,
mood: _selectedMood!,
weather: weather['main'] ?? '',
    );
    _allItems = items;
    _categoryIndexes.clear();
    for (var item in recommended) {
final category = item['category'];
_categoryIndexes[category] = 0;
    }

    if (!context.mounted) return;
    setState(() {
_recommendedItems = recommended;
    });
    _generateAiExplanation(weather);
    _categoryIndexes.clear();
    for (var item in recommended) {
final category = item['category'];
_categoryIndexes[category] = 0;
    }


  }

  Future<Map<String, dynamic>>
_checkLocationPermissionAndFetchWeather() async {
    bool serviceEnabled = await
Geolocator.isLocationServiceEnabled();
    if (!serviceEnabled) {
await _showLocationDialog(
  title: 'Служба геолокації вимкнена',
  message: 'Будь ласка, увімкніть GPS у налаштуваннях
пристрою.',
);
throw Exception('GPS вимкнено');
    }

    LocationPermission permission = await
Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
permission = await Geolocator.requestPermission();
if (permission == LocationPermission.denied) {
  await _showLocationDialog(
    title: 'Доступ до геолокації відхилено',
    message: 'Надайте доступ до геопозиції, щоб ми могли
показати погоду.',
  );
  throw Exception('Геолокацію відхилено');
}
    }

    if (permission == LocationPermission.deniedForever) {
await _showLocationDialog(
  title: 'Доступ до геолокації заблокований',
  message: 'Відкрий налаштування → Додатки → Дозволи
→ Геолокація.',
);
throw Exception('Геолокація заблокована назавжди');
    }

    try {
Position position = await Geolocator.getCurrentPosition(
  desiredAccuracy: LocationAccuracy.high,
).timeout(
  const Duration(seconds: 10),
```

```
  onTimeout: () => throw Exception('Таймаут: не отримано
координати'),
);

return await WeatherService.fetchWeatherByLocation(
  latitude: position.latitude,
  longitude: position.longitude,
);
    } catch (e) {
debugPrint(' Геолокація не вдалася: $e\n Використовуємо
Київ');
return await WeatherService.fetchWeatherByLocation(
  latitude: 50.4501,
  longitude: 30.5234,
);
    }
  }

  Future<void> _showLocationDialog({
    required String title,
    required String message,
  }) async {
    if (!context.mounted) return;

    await showDialog(
context: context,
builder: (_) => AlertDialog(
  title: Text(title),
  content: Text(message),
  actions: [
    TextButton(
child: const Text('OK'),
onPressed: () => Navigator.of(context).pop(),
    )
  ],
),
    );
  }

  Widget _buildLookCategory(String label, String category) {
    final categoryItems = _allItems.where((item) =>
item['category'] == category).toList();
    final currentIndex = _categoryIndexes[category] ?? 0;

    if (categoryItems.isEmpty) {
return Column(
  children: [
    Row(
mainAxisAlignment: MainAxisAlignment.center,
children: [
  const Icon(Icons.arrow_back_ios, color: Colors.grey),
  _buildImageBox(null),
  const Icon(Icons.arrow_forward_ios, color: Colors.grey),
],
    ),
    const SizedBox(height: 4),
    Text(label, style: const TextStyle(fontWeight:
FontWeight.bold)),
    const SizedBox(height: 16),
  ],
);
    }

    final currentItem = categoryItems[currentIndex];

    return Column(
children: [
  Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
IconButton(
  icon: const Icon(Icons.arrow_back_ios),
  onPressed: () {
    setState(() {
_categoryIndexes[category] =
    (currentIndex - 1 + categoryItems.length) %
categoryItems.length;
    });
  },
),
_buildImageBox(currentItem['imageUrl']),
IconButton(
  icon: const Icon(Icons.arrow_forward_ios),
  onPressed: () {
    setState(() {
_categoryIndexes[category] =
    (currentIndex + 1) % categoryItems.length;
    });
  },
),
    ],
  ),
  const SizedBox(height: 4),
  Text(label, style: const TextStyle(fontWeight:
FontWeight.bold)),
  const SizedBox(height: 16),
],
    );
  }
  Widget _buildImageBox(String? imageUrl) {
    return Container(
width: 80,
height: 80,
margin: const EdgeInsets.symmetric(horizontal: 16),
decoration: BoxDecoration(
  border: Border.all(color: Colors.black, width: 2),
),
child: imageUrl != null
    ? Image.network(imageUrl, fit: BoxFit.cover)
    : const Center(child: Text('❌\nНемає', textAlign:
TextAlign.center)),
    );
  }


  @override
  Widget build(BuildContext context) {
    if (_showMoodDialog) {
WidgetsBinding.instance.addPostFrameCallback((_) async {
  await _showMoodSelectionDialog();
```

```
  setState(() => _showMoodDialog = false);
});
    }

    return Scaffold(
appBar: AppBar(
  title: const Text('Головна'),
  actions: [
    IconButton(
icon: const Icon(Icons.person),
onPressed: () => Navigator.pushNamed(context, '/profile'),
    ),
    IconButton(
icon: const Icon(Icons.logout),
onPressed: () async {
  await FirebaseAuth.instance.signOut();
  if (context.mounted) {
    Navigator.pushReplacementNamed(context, '/login');
  }
},
    )
  ],
),
body: RefreshIndicator(
  onRefresh: () async => setState(() => _weatherFuture =
_checkLocationPermissionAndFetchWeather()),
  child: FutureBuilder<Map<String, dynamic>>(
    future: _weatherFuture,
    builder: (context, snapshot) {
if (snapshot.connectionState == ConnectionState.waiting) {
  return const Center(child: CircularProgressIndicator());
} else if (snapshot.hasError) {
  return Center(child: Text('Помилка: ${snapshot.error}'));
} else if (!snapshot.hasData) {
  return const Center(child: Text('Немає даних про погоду'));
}

final weather = snapshot.data!;

return ListView(
  padding: const EdgeInsets.all(16),
  children: [
    Center(
child: Column(
  children: [

Image.network("https://openweathermap.org/img/wn/${weathe
r['icon']}@2x.png", width: 100, height: 100),
    const SizedBox(height: 8),
    Text(
'${weather['temp'].toStringAsFixed(1)}°C',
style: const TextStyle(fontSize: 42, fontWeight:
FontWeight.bold),
    ),
    const SizedBox(height: 4),
    Text(weather['description'], style: const TextStyle(fontSize:
20)),
    const SizedBox(height: 8),
    Text('Місто: ${weather['city'] ?? 'Невідомо'}', style: const
TextStyle(fontStyle: FontStyle.italic)),
  ],
),
    ),
    const SizedBox(height: 24),
    if (_showLookUI && _recommendedItems == null)
ElevatedButton(
  onPressed: _generateLook,
  child: const Text('Згенерувати лук'),
),
    AnimatedSwitcher(
duration: const Duration(milliseconds: 500),
switchInCurve: Curves.easeIn,
switchOutCurve: Curves.easeOut,
child: _recommendedItems != null
    ? Column(
  key: ValueKey(_recommendedItems.hashCode), // важливо!
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    const SizedBox(height: 24),
    const Text(
'Згенерований лук:',
style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
textAlign: TextAlign.center,
    ),
    const SizedBox(height: 16),
    _buildLookCategory('Голова', 'Голова'),
    _buildLookCategory('Тіло', 'Тіло'),
    _buildLookCategory('Ноги', 'Ноги'),
    _buildLookCategory('Ступні', 'Ступні'),
    const Divider(thickness: 1.5, height: 32),
    _buildLookCategory('Аксесуари', 'Аксесуари'),
    const SizedBox(height: 24),
    if (_aiExplanation != null)
Container(
  padding: const EdgeInsets.all(16),
  decoration: BoxDecoration(
    color: Colors.purple.shade50,
    borderRadius: BorderRadius.circular(12),
    border: Border.all(color: Colors.purple.shade200),
  ),
  child: Text(
    '🤓 ІІІ: $_aiExplanation',
    style: const TextStyle(fontSize: 16),
  ),
),
    const SizedBox(height: 16),
    Center(
child: ElevatedButton.icon(
  icon: const Icon(Icons.refresh),
  label: const Text('Згенерувати ще раз'),
  onPressed: _generateLook,
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.deepPurple,
    foregroundColor: Colors.white,
    padding: const EdgeInsets.symmetric(horizontal: 24,
vertical: 12),
    shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(8)),
  ),
),
```

```
    ),
  ],
)
    : const SizedBox.shrink(),
    ),
  ],
);
```

```
    },
  ),
),
    );
  }
}
```

Код profile_page.dart

```
import 'dart:io';
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:firebase_storage/firebase_storage.dart';
import 'package:image_picker/image_picker.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'clothing_gallery_page.dart';

class ProfilePage extends StatefulWidget {
  const ProfilePage({super.key});

  @override
  State<ProfilePage> createState() => _ProfilePageState();
}

class _ProfilePageState extends State<ProfilePage> {
  final user = FirebaseAuth.instance.currentUser;
  String? avatarUrl;

  @override
  void initState() {
    super.initState();
    _loadAvatar();
  }

  Future<void> _loadAvatar() async {
    final doc = await
FirebaseFirestore.instance.collection('user').doc(user!.uid).get()
;
    setState(() {
      avatarUrl = doc.data()?['avatarUrl'];
    });
  }

  Future<void> _pickAndUploadAvatar() async {
    final picker = ImagePicker();
    final picked = await picker.pickImage(source:
ImageSource.gallery);
    if (picked == null) return;

    final file = File(picked.path);
    final storageRef =
FirebaseStorage.instance.ref().child('avatars/${user!.uid}.jpg');

    await storageRef.putFile(file);
    final url = await storageRef.getDownloadURL();

    await
FirebaseFirestore.instance.collection('user').doc(user!.uid).upda
te({
      'avatarUrl': url,
    });

    setState(() {
      avatarUrl = url;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Профіль')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            // Аватарка
            Stack(
              alignment: Alignment.bottomRight,
              children: [
                CircleAvatar(
                  radius: 50,
                  backgroundImage: avatarUrl != null
```

```
                ? NetworkImage(avatarUrl!)
                : const
AssetImage('assets/avatar_placeholder.png') as ImageProvider,
            ),
            InkWell(
              onTap: _pickAndUploadAvatar,
              child: CircleAvatar(
                radius: 16,
                backgroundColor: Colors.white,
                child: const Icon(Icons.add, size: 16),
              ),
            ),
          ],
        ),
        const SizedBox(height: 12),

        // Email
        Text(
          user?.email ?? 'Користувач',
          style: const TextStyle(fontSize: 16, fontWeight:
FontWeight.w500),
        ),
        const SizedBox(height: 24),

        // Кнопки
        ElevatedButton.icon(
          icon: const Icon(Icons.add),
          label: const Text('Немає що вдягнути? Завантажуй
фотографії!'),
          onPressed: () {
            Navigator.pushNamed(context, '/add-clothing');
```

Код add_clothing_page.dart

```
import 'dart:convert';
import 'dart:io';
import 'package:flutter/material.dart';
import 'package:image_picker/image_picker.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:http/http.dart' as http;
import 'package:http_parser/http_parser.dart';

class AddClothingPage extends StatefulWidget {
  const AddClothingPage({super.key});

  @override
  State<AddClothingPage> createState() =>
_AddClothingPageState();
}

class _AddClothingPageState extends
State<AddClothingPage> {
  File? _imageFile;
  final _nameController = TextEditingController();
  String? _selectedCategory;
  final List<String> _selectedTags = [];
  bool _isLoading = false;

  final List<String> _categories = [
    'Голова',
    'Обличчя',
          },
          style: ElevatedButton.styleFrom(
            minimumSize: const Size.fromHeight(60),
            textStyle: const TextStyle(fontSize: 16),
          ),
        ),
        const SizedBox(height: 16),
        ElevatedButton.icon(
          icon: const Icon(Icons.photo_library),
          label: const Text('Моя галерея одягу'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (_) => const
ClothingGalleryPage()),
            );
          },
          style: ElevatedButton.styleFrom(
            minimumSize: const Size.fromHeight(60),
            backgroundColor: Colors.grey[800],
            foregroundColor: Colors.white,
            textStyle: const TextStyle(fontSize: 16),
          ),
        ),
      ],
    ),
  ),
 );
 }
}
```

```
    'Шия',
    'Тіло',
    'Руки',
    'Пояс',
    'Ноги',
    'Ступні',
    'Аксесуари'
  ];

  final List<String> _tags = [
    'теплий',
    'легкий',
    'спортивний',
    'строгий',
    'зручний',
    'універсальний'
  ];

  Future<void> _pickImage() async {
    final picked = await ImagePicker().pickImage(source:
ImageSource.gallery);
    if (picked != null) {
      setState(() {
        _imageFile = File(picked.path);
      });
    }
  }

  Future<void> _saveClothingItem() async {
    final enteredCategory = _isCustomCategory
        ? _customCategoryController.text.trim()
        : _selectedCategory;

    if (_imageFile == null || _nameController.text.isEmpty ||
enteredCategory == null || enteredCategory.isEmpty) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Будь ласка, заповніть усі
поля')),
      );
      return;
    }

    setState(() => _isLoading = true);

    try {
      final user = FirebaseAuth.instance.currentUser;
      if (user == null) throw Exception('Користувач не
авторизований');

      final processedImage = await
_removeBackground(_imageFile!);
      if (processedImage == null || !processedImage.existsSync())
{
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('Оброблений файл не
знайдено')),
        );
        setState(() => _isLoading = false);
        return;
      }

      final imageUrl = await
_uploadToCloudinary(processedImage);
      if (imageUrl == null) throw Exception('Не вдалося
завантажити зображення');

      await FirebaseFirestore.instance
          .collection('clothes')
          .doc(user.uid)
          .collection('items')
          .add({
        'name': _nameController.text,
        'category': enteredCategory,
        'tags': _selectedTags,
        'imageUrl': imageUrl,
        'createdAt': FieldValue.serverTimestamp(),
      });

      if (mounted) {
        ScaffoldMessenger.of(context).showSnackBar(
          const SnackBar(content: Text('✅ Одяг успішно
додано!')),
        );
        Navigator.pop(context);
      }
    } catch (e) {
      print('❌ Помилка при збереженні: $e');
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Помилка: ${e.toString()}')),
      );
    } finally {
      setState(() => _isLoading = false);
```

```
  }
 }


 Future<File?> _removeBackground(File imageFile) async {
  final request = http.MultipartRequest(
   'POST',
   Uri.parse('https://api.remove.bg/v1.0/removebg'),
  )
   ..headers['X-Api-Key'] =
'CVghkAGZQTHJzvnNA5TU2ecp'
   ..files.add(await http.MultipartFile.fromPath('image_file',
imageFile.path));

  final response = await request.send();
  if (response.statusCode == 200) {
   final bytes = await response.stream.toBytes();
   final tempDir = Directory.systemTemp;
   final processed = await
File('${tempDir.path}/no_bg_${DateTime.now().millisecondsS
inceEpoch}.png')
       .writeAsBytes(bytes);
   return processed;
  } else {
   print('❌ Remove.bg помилка: ${response.statusCode}');
   return null;
  }
 }

 Future<String?> _uploadToCloudinary(File imageFile) async
{
  const cloudName = 'dzeu8vz2b';
  const uploadPreset = 'unsigned_preset';
  final url =
Uri.parse('https://api.cloudinary.com/v1_1/$cloudName/image/
upload');

  final request = http.MultipartRequest('POST', url)
   ..fields['upload_preset'] = uploadPreset
   ..files.add(await http.MultipartFile.fromPath(
    'file',
    imageFile.path,
    contentType: MediaType('image', 'png'),
   ));

  try {
   final response = await request.send();
   final responseBody = await
response.stream.bytesToString();
   if (response.statusCode == 200) {
    final jsonData = jsonDecode(responseBody);
    return jsonData['secure_url'];
   } else {
    print('❌ Cloudinary upload failed:
${response.statusCode}');
    return null;
   }
  } catch (e) {
   print('❌ Виняток Cloudinary: $e');
   return null;
  }
 }
// Додати перед build()
 final _customCategoryController = TextEditingController();
 bool _isCustomCategory = false;
 final _customTagController = TextEditingController();

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(title: const Text('Додати одяг')),
   body: SingleChildScrollView(
    padding: const EdgeInsets.all(16),
    child: Column(
     crossAxisAlignment: CrossAxisAlignment.start,
     children: [
      GestureDetector(
       onTap: _pickImage,
       child: _imageFile != null
         ? Image.file(_imageFile!, height: 200)
         : Container(
        height: 200,
        width: double.infinity,
        color: Colors.grey[300],
        child: const Icon(Icons.add_a_photo, size: 40),
       ),
      ),
      const SizedBox(height: 16),
      TextField(
       controller: _nameController,
       decoration: const InputDecoration(
```

```
            labelText: 'Назва одягу',
            border: OutlineInputBorder(),
           ),
          ),
          const SizedBox(height: 16),

// Категорія з можливістю додавання
          DropdownButtonFormField<String>(
           value: _selectedCategory,
           items: [
            ..._categories.map((cat) =>
DropdownMenuItem(value: cat, child: Text(cat))),
            const DropdownMenuItem(value: 'custom', child:
Text('Інше (додати свою категорію)')),
           ],
           onChanged: (val) {
            setState(() {
             if (val == 'custom') {
              _isCustomCategory = true;
              _selectedCategory = null;
             } else {
              _isCustomCategory = false;
              _selectedCategory = val;
             }
            });
           },
           decoration: const InputDecoration(
            labelText: 'Категорія',
            border: OutlineInputBorder(),
           ),
          ),
          if (_isCustomCategory)
           Padding(
            padding: const EdgeInsets.only(top: 12),
            child: TextField(
             controller: _customCategoryController,
             decoration: const InputDecoration(
              labelText: 'Власна категорія',
              border: OutlineInputBorder(),
             ),
            ),
           ),

          const SizedBox(height: 24),

// Теги з чекбоксами
          const Text('Теги', style: TextStyle(fontWeight:
FontWeight.bold)),
          Wrap(
           spacing: 8,
           children: _tags.map((tag) {
            final isSelected = _selectedTags.contains(tag);
            return FilterChip(
             label: Text(tag),
             selected: isSelected,
             onSelected: (selected) {
              setState(() {
               if (selected) {
                _selectedTags.add(tag);
               } else {
                _selectedTags.remove(tag);
               }
              });
             },
            );
           }).toList(),
          ),

// Додавання нового тегу
          const SizedBox(height: 12),
          Row(
           children: [
            Expanded(
             child: TextField(
              controller: _customTagController,
              decoration: const InputDecoration(
               labelText: 'Новий тег',
               border: OutlineInputBorder(),
              ),
             ),
            ),
            const SizedBox(width: 8),
            ElevatedButton(
             onPressed: () {
              final newTag = _customTagController.text.trim();
              if (newTag.isNotEmpty &&
!_tags.contains(newTag)) {
               setState(() {
                _tags.add(newTag);
                _selectedTags.add(newTag);
                _customTagController.clear();
               });
```

```
                  }
                },
                child: const Text('Додати'),
              ),
            ],
          ),

          const SizedBox(height: 24),
          _isLoading
              ? const Center(child: CircularProgressIndicator())
              : ElevatedButton(
            onPressed: _saveClothingItem,
            child: const Text('Зберегти'),
          ),
        ],
      ),
    ),
  );
 }
}
```