МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КРАВЧУК РОСТИСЛАВ ЮРІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2026 р.

ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ МОБІЛЬНОГО ЗАСТОСУНКУ «АСИСТЕНТ ДЛЯ ГАРДЕРОБУ» НА ОСНОВІ КЛІЄНТ-СЕРВЕРНОЇ АРХІТЕКТУРИ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Юрій ПОРЕМСЬКИЙ, старший
викладач кафедри інформаційних
технологій, к.т.н.
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця – 2026

АНОТАЦІЯ

Кравчук Р.Ю. Проєктування та реалізація мобільного застосунку «Асистент для гардеробу» на основі клієнт-серверної архітектури. Спеціальність 122 «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2026.

У кваліфікаційній (бакалаврській) роботі досліджено принципи побудови масштабованих розподілених систем для управління персональними даними. Розроблено програмний комплекс, що складається з мобільного клієнта на базі фреймворку Expo (React Native) та серверної частини на базі Node.js

Ключові слова: React Native, Expo, Node.js, клієнт-серверна архітектура, мобільний застосунок, асинхронна взаємодія.

91 ст. 20 рис., 5 табл., 2 дод., 42 джерела.

ABSTRACT

Kravchuk R. Design and implementation of the «Wardrobe Assistant» mobile application based on client-server architecture. Specialty 122 «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia 2026.

The qualification (bachelor’s) work explores the principles of building scalable distributed systems for personal data management. A software complex has been developed, consisting of a mobile client based on the Expo (React Native) framework and a backend based on Node.js.

Keywords: React Native, Expo, Node.js, client-server architecture, mobile application, asynchronous interaction

ЗМІСТ

ВСТУП

Актуальність теми дослідження. У сучасному постіндустріальному суспільстві цифровізація повсякденного життя стає не просто трендом, а необхідною умовою ефективного управління часом та ресурсами. Швидкий темп міського життя, зростання добробуту та культура масового споживання призводять до значного накопичення особистих речей, що створює серйозну проблему систематизації та раціонального використання персонального гардеробу. Традиційні методи організації побуту вже не задовольняють потреби сучасного користувача, який прагне мати швидкий доступ до інформації про свої активи у будь-який час. Саме тому розробка мобільного застосунку, який виконує роль інтелектуального асистента для гардеробу, є актуальним науково-практичним завданням, що поєднує в собі аспекти ергономіки, стилістики та передових інформаційних технологій.

З технічної точки зору, актуальність роботи підкріплена необхідністю дослідження методів побудови високонавантажених клієнт-серверних систем, здатних стабільно функціонувати в умовах мобільного зв’язку. Створення такого продукту вимагає вирішення складних інженерних задач: від забезпечення миттєвої синхронізації даних між різними пристроями до оптимізації зберігання та обробки медіаконтенту (зображень одягу). Використання сучасних інструментів, таких як середовище Node.js та фреймворк Expo (React Native), дозволяє реалізувати кросплатформеність, проте саме впровадження брокера повідомлень RabbitMQ та контейнеризації через Docker виводить рішення на професійний рівень, забезпечуючи архітектурну гнучкість та відмовостійкість, що часто ігнорується у подібних за тематикою розробках.

Мета дослідження полягає у проектуванні, програмній реалізації та комплексному тестуванні архітектурно стабільного мобільного додатку «Асистент для гардеробу». Основною ціллю є створення єдиної екосистеми, яка дозволяє користувачу автоматизувати процес каталогізації речей, планування щоденних образів та оптимізації використання власного майна шляхом впровадження сучасних методів збереження та передачі даних.

Завдання дослідження:

1. Проаналізувати наявний ринок мобільних рішень у сегменті персональних органайзерів, виявити їхні функціональні недоліки та на основі отриманих даних сформувати технічні вимоги до сучасної системи управління гардеробом.
2. Дослідити переваги та обмеження клієнт-серверної архітектури в контексті мобільної розробки, а також обґрунтувати вибір технологічного стеку (Node.js, PostgreSQL, Expo) як найбільш оптимального для реалізації поставленої мети.
3. Встановити механізми асинхронної взаємодії між компонентами системи, використовуючи брокер повідомлень RabbitMQ, з метою розвантаження основного потоку виконання та підвищення загальної продуктивності серверної частини.
4. Спроектувати реляційну структуру бази даних, яка забезпечить цілісність інформації про складні зв’язки між категоріями одягу, сезонами та комбінаціями образів, а також розробити протоколи взаємодії мобільного клієнта з API.
5. Розробити програмний комплекс, інтегрувавши фронтенд-частину на React Native з бекенд-сервісами, та провести розгортання системи в ізольованих контейнерах Docker для забезпечення ідентичності середовищ розробки та продакшну.
6. Проаналізувати результати тестування розробленого додатку на предмет швидкодії та коректності обробки черг повідомлень під час інтенсивної взаємодії користувача з інтерфейсом.

Об’єкт дослідження – процес автоматизації процесів управління персональними даними та ресурсами користувача у сфері побуту із застосуванням сучасних мобільних та серверних технологій.

Предмет дослідження – архітектурні підходи, алгоритмічні рішення та методи програмної інженерії, що використовуються при розробці

масштабованого клієнт-серверного додатку на базі Node.js, RabbitMQ, PostgreSQL та React Native.

Теоретичне значення роботи полягає у дослідженні та обґрунтуванні ефективності впровадження мікросервісної архітектури для розробки мобільних екосистем. У роботі продемонстровано, як поділ монолітної структури на незалежні сервіси, що взаємодіють через брокер повідомлень RabbitMQ, дозволяє досягти високого рівня модульності та гнучкості системи. Отримана програмна модель доводить, що використання асинхронних черг повідомлень у мікросервісному середовищі значно знижує взаємозалежність (coupling) між компонентами, забезпечуючи стабільну роботу мобільного клієнта навіть під час виконання інтенсивних обчислювальних процесів на сервері.

Практична цінність результатів дослідження підтверджується розробкою цілісного, готового до експлуатації програмного комплексу «Асистент для гардеробу». Впровадження мікросервісного підходу дозволило ізолювати ключові функції – такі як управління базою даних персональних речей у PostgreSQL, обробка медіафайлів та логіка формування образів – у окремі автономні одиниці. Це забезпечує можливість незалежного оновлення та масштабування кожного компонента системи залежно від навантаження.

Апробація результатів дослідження. Результати та основні положення кваліфікаційної роботи були представлені доповідями та обговорені на всеукраїнській науково-практичній конференції «Комп’ютерні технології обробки даних» (8 грудня 2022 р., м. Вінниця)

Представлені доповіді:

- Доповідь: «Еволюція об’єктно-орієнтованих баз даних від перших JSON до повноцінних об’єктно-орієнтованих СУБД»[1].
- Доповідь: «Комбінаторика в програмуванні. Основні проблеми та алгоритм вирішення їх»[2].

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ ТА СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ЦИФРОВІЗАЦІЇ ПЕРСОНАЛЬНОГО ГАРДЕРОБУ

1.1 Соціо-технічні передумови та наукова парадигма інтелектуального управління споживанням одягу

Глобальна індустрія моди на сучасному етапі перебуває у стані глибокої та багатовекторної трансформації, що зумовлена гострим конфліктом між усталеною моделлю надлишкового виробництва та зростаючими імперативними вимогами суспільства до сталого розвитку. Згідно з аналітичними даними провідних дослідницьких інститутів, концепція «швидкої моди» (*Fast Fashion*) призвела до системної кризи споживання. Дана модель базується на постійному оновленні асортименту та стимулюванні імпульсивних покупок, що призвело до ситуації, коли середньостатистичний споживач сьогодні купує на 60% більше предметів одягу, ніж це було 15 років тому, проте фактичний термін експлуатації кожної одиниці скоротився майже вдвічі [3].

У науковому дискурсі ця проблема розглядається не лише як екологічна загроза глобального масштабу, а й як специфічна інформаційна аномалія: індивід стикається з критичним надлишком несистематизованих фізичних об’єктів у власному житловому просторі. Це провокує виникнення явища, відомого як «гардеробний параліч» – когнітивний стан, при якому надмірна кількість варіантів вибору (ентропія гардеробу) блокує здатність до прийняття швидкого, раціонального та естетично виправданого рішення [7]. Цифровізація персонального гардеробу виступає як інноваційний та методологічно обґрунтований метод розв’язання цієї суперечності через концепцію створення «Цифрового двійника» (*Digital Twin*) приватних речей. Наукова новизна такого підходу полягає у фундаментальному переході від парадигми пасивного зберігання речей до стратегії активного управління цифровими активами. Теоретичний підхід до розробки інтелектуального асистента гардеробу базується на фундаментальних засадах когнітивістики та теорії інформації. Систематизація

та каталогізація речей у мобільному додатку дозволяє суттєво знизити рівень «втоми від прийняття рішень» (*decision fatigue*) – психологічного виснаження, що накопичується протягом дня через необхідність постійного вибору [9].

Дослідження сучасних психологів та соціологів моди підтверджують, що візуальна впорядкованість та структурованість бази даних одягу в цифровому середовищі безпосередньо корелює з підвищенням рівня психологічного комфорту користувача, зростанням задоволеності власним іміджем та детермінованим зниженням рівня тривожності перед соціальними взаємодіями [7]. Більше того, використання систем комп'ютерного бачення для ідентифікації речей дозволяє об'єктивізувати сприйняття власного стилю, відходячи від суб'єктивних та часто помилкових оцінок наповненості гардеробу [8].

Окрім психологічного та когнітивного аспектів, у межах дослідження вкрай важливим є екологічний вектор розвитку технологій (*Sustainable Fashion*). Цифрові інструменти стають базисом для впровадження принципів відповідального споживання. У фундаментальних роботах Niinimäki (2020) математично доводиться, що просте подовження терміну активної експлуатації одягу лише на 9 місяців дозволяє кумулятивно зменшити вуглецевий слід, обсяги текстильних відходів та питоме використання води на 20–30% для кожної одиниці товару [5].

Мобільний застосунок у цьому контексті трансформується в дієвий інструмент «циркулярної економіки». Він надає користувачеві прозору, верифіковану статистику фактичного використання кожної речі, що дозволяє виявити «неактивні» елементи гардеробу та прийняти рішення про їх подальшу долю (перепродаж, переробка або стилізація в нових образах). Це дозволяє впровадити математично обґрунтовану модель оцінки економічної та експлуатаційної ефективності гардеробу через показник вартості за одне використання – *Cost per Wear* (CPW). Даний показник є ключовим індикатором раціональності гардеробу і обчислюється за формулою:

$$CPW = \frac{P + M}{n}$$

Де P – початкова ціна речі при придбанні; M – сукупні витрати на обслуговування (хімічна чистка, прання, ремонтні роботи); n – кількість фактичних використань речі (виходів) протягом її життєвого циклу.

Проектована інтелектуальна система має на меті повністю автоматизувати цей розрахунок, використовуючи алгоритми відстеження календаря та активності користувача. Надання таких об'єктивних даних дозволяє змінити споживчу поведінку: замість купівлі великої кількості дешевих речей низької якості, користувач стимулюється до інвестування в якісні одиниці одягу, які мають нижчий CPW у довгостроковій перспективі [5].

Більше того, інтеграція мікросервісної архітектури та асинхронної обробки даних за допомогою брокерів повідомлень (наприклад, RabbitMQ) дозволяє реалізувати складні алгоритми рекомендацій образів на основі зовнішніх чинників [11]. Застосунок аналізує не лише наявні речі, а й зіставляє їх із прогнозом погоди, типом запланованої події та колірною сумісністю, що робить його повноцінною експертною системою. Таким чином, науково–практичне завдання розробки асистента гардеробу лежить у площині міждисциплінарного синтезу інформаційних технологій, прикладної математики, методів штучного інтелекту та соціальної екології, що відповідає актуальним запитам глобального цифрового суспільства.

1.2 Порівняльний аналіз технологічних рішень та критичний огляд існуючих систем автоматизації стилю

При науковому проектуванні програмного забезпечення для інтелектуального управління гардеробом критично важливо враховувати мультидисциплінарну специфіку предметної області, яка виходить далеко за межі звичайної розробки інтерфейсів. Дана сфера вимагає складного та глибокого синтезу методів комп'ютерного бачення для розпізнавання об'єктів на зображеннях зі складним фоном, технологій обробки природної мови (NLP) для семантичного аналізу текстових описів, категорій та тегів, а також розробки багатокритеріальних алгоритмів оптимізації для формування індивідуальних

рекомендацій [8]. З погляду системної інженерії, такий застосунок є не просто цифровим каталогом, а повноцінною експертною системою, що оперує великими масивами неструктурованих візуальних та текстових даних.

Ретельний та системний аналіз сучасного світового ринку мобільних застосунків дозволяє виокремити декілька ключових лідерів, таких як Stylebook, Acloset та Cladwell. Кожен із цих продуктів пропонує власний підхід до вирішення проблеми організації одягу, проте глибоке наукове дослідження їхньої внутрішньої архітектури, алгоритмічної бази та користувацького досвіду виявляє суттєві технологічні розриви. Ці прогалини суттєво обмежують їхню ефективність для широкого кола користувачів, які не готові витрачати надмірну кількість часу на обслуговування системи.

Детальний розгляд функціональних можливостей інтерфейсу Stylebook (рис. 1.1) демонструє акцент на методології «цифрового архівування». Основний інструментарій тут зосереджений на створенні статичних колажів та веденні календаря носіння. Проте, як свідчить архітектура меню, система вимагає високого рівня ручної залученості: від очищення фону до класифікації кожного об’єкта. Відсутність автоматизованих агентних пропозицій робить процес формування образів трудомістким, що є критичним недоліком для сучасного динамічного ритму життя.

Рисунок 1.1 – Інтерфейс створення образів у застосунку Stylebook

На противагу жорсткій структурі, застосунок Acloset (рис. 1.2) намагається автоматизувати вхідний потік даних за допомогою штучного інтелекту. Наявність функцій «Beautify» та автоматичного розпізнавання категорій («All Clothes») значно спрощує первинну оцифровку гардероба. Однак аналіз інтерфейсу вказує на зміщення фокусу в бік комерціалізації: інтеграція з маркетплейсом та розділом «Wishlist» часто перевантажує користувацький досвід рекламним контентом, що нівелює глибину персоналізації власного наявного одягу.

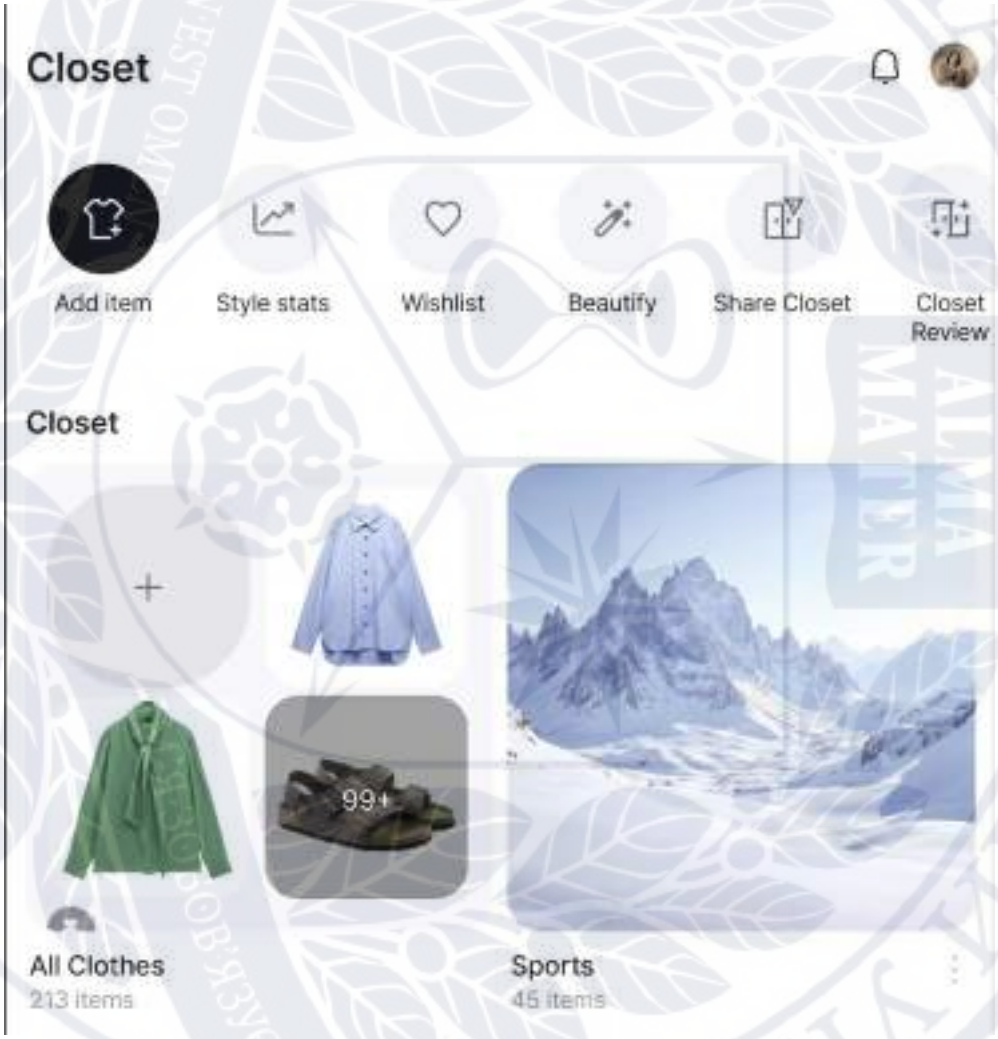

Рисунок 1.2 – Головний екран та інструменти автоматизації Acloset

Замикає трійку лідерів сервіс Cladwell (рис. 1.3), який обирає шлях стилістичної курації, розділяючи речі за напрямами (наприклад, «Romantic»). Такий підхід полегшує формування капсульного гардероба для недосвідченого користувача. Проте, як видно з логіки групування шарів («Layers») та сорочок, система працює в межах досить жорстких пресетів. Це обмежує гнучкість

алгоритмів при роботі з нестандартними елементами одягу або складними поєднаннями фактур, які не вписуються в заздалегідь визначені шаблони.

Рисунок 1.3 – Категоризація та стилістична курація в Cladwell

Фундаментальна та найбільш критична проблема більшості існуючих комерційних рішень полягає у надзвичайно «високому порозі входу» для кінцевого споживача. Процес первинної ініціалізації віртуального гардеробу вимагає значних когнітивних та часових витрат: користувач змушений власноруч фотографувати кожну одиницю одягу в однакових умовах освітлення, самостійно проводити трудомістке видалення фону (що часто реалізовано за допомогою примітивних інструментів маскування або напівавтоматичних алгоритмів, які не забезпечують високої якості країв та чистоти зображення) і, нарешті, детально прописувати всі метадані від кольору до бренду [8]. Такий рівень операційного навантаження створює так зване «тертя» (*friction*) у взаємодії, що неминуче призводить до високого рівня відтоку користувачів уже на етапі оцифрування перших десяти–п’ятнадцяти речей.

Проведений критичний огляд існуючих програмних рішень дозволяє глибше проаналізувати технологічний розрив між наявними на ринку

продуктами та динамічними вимогами сучасного користувача до швидкості, зручності та крос–платформеності. Зокрема, система Stylebook базується переважно на локальних архітектурах збереження даних. З погляду системного проектування, така «ізольована» модель суттєво обмежує можливості хмарної синхронізації, ускладнює масштабування сервісу при зростанні бази даних та, що найважливіше, залишає все обчислювальне навантаження з маніпуляції зображеннями на стороні мобільного пристрою, що може призводити до перегріву та швидкого розряду акумулятора [10].

Натомість застосунок Acloset використовує хмарну модель SaaS (*Software as a Service*) та пропонує інтегровані інструменти штучного інтелекту для обробки фотографій на серверній стороні. Проте детальний аналіз логіки формування образів свідчить, що його алгоритми рекомендацій залишаються на рівні базового машинного навчання, яке часто ігнорує складний соціальний контекст подій (наприклад, різницю між «Casual Friday» та офіційною бізнес–зустріччю) чи специфічні індивідуальні стилістичні вподобання конкретного користувача [7]. Застосунок Cladwell, своєю чергою, робить ставку на методи колаборативної фільтрації для створення готових образів, проте він майже повністю ігнорує автоматизацію етапу введення даних. Це змушує власників розлогих гардеробів витрачати десятки годин на ручне оцифрування, що зводить нанівець переваги інтелектуальних рекомендацій [8].

Ключовою технологічною перевагою проектованої системи, що вигідно вирізняє її серед конкурентів, є інтеграція брокера повідомлень RabbitMQ. Це рішення забезпечує повну асинхронність усіх внутрішніх процесів та надзвичайно високу відмовостійкість додатку навіть при пікових навантаженнях. Жоден із проаналізованих аналогів не демонструє подібного рівня архітектурної зрілості, яка дозволяє не лише миттєво реагувати на дії користувача в інтерфейсі, а й паралельно, у фоновому режимі, виконувати складні операції.

Використання RabbitMQ дозволяє реалізувати подієво–орієнтовану модель: як тільки користувач завантажує нове фото, воно негайно потрапляє в

чергу на обробку, де окремі воркери, побудовані на базі прогресивних моделей (зокрема з можливістю залучення штучного інтелекту для семантичного та візуального аналізу), проводять глибоке сканування об’єкта. Це включає не лише видалення фону, а й розпізнавання складних атрибутів, як–от тип тканини, стиль крою та дрібна фурнітура, що було б неможливо при синхронній архітектурі [11]. Такий підхід не тільки підвищує суб’єктивну швидкість роботи програми, а й закладає надійний фундамент для майбутньої інтеграції з екосистемою IoT, наприклад, інтелектуальними дзеркалами чи системами автоматизації «розумного дому», де швидкість та надійність обміну даними є критичними факторами.

Детальний порівняльний аналіз технічних характеристик та архітектурних підходів проектованої системи у зіставленні з існуючими рішеннями наведено у таблиці 1.1.

Таблиця 1.1 Порівняння серверних архітектур конкурентів

Критерій порівняння	**Stylebook**	**Acloset**	**Cladwell**	**Проектована система**
Основна архітектура	Монолітна	Клієнт-серверна	Модульна	Подієво-орієнтована
Метод обробки черг	Відсутній (Direct API)	HTTP Request-Response	HTTP Request-Response	Брокер повідомлень RabbitMQ
Відмовостійкість	Низька (блокування UI)	Середня	Середня	Висока (Persistence & Retries)
Масштабованість	Вертикальна	Гібридна	Обмежена	Горизонтальна (Worker Scaling)

Окремим, досі невирішеним аспектом у сучасних аналогах залишається науково обґрунтована інтелектуальна сумісність речей, що базується на динамічних критеріях та реальному контексті. Більшість додатків на ринку

пропонують або хаотичні комбінації, або покладаються виключно на суб'єктивний вибір власника, що зовсім не вирішує проблему «парадоксу вибору». У даній роботі пропонується реалізація інтелектуального модуля, де кожна одиниця одягу розглядається як набір ознак, що проходять через багаторівневу систему фільтрації та синтезу на основі трьох фундаментальних стовпів:

Інтелектуальна колористика та візуальний аналіз: завдяки застосуванню нейронних мереж система здатна проводити точну сегментацію зображення, виокремлюючи домінантні та акцентні кольори. На відміну від простих палітр, алгоритм перевіряє відповідність комбінацій гармонійним схемам. Моделі комп'ютерного бачення дозволяють враховувати не лише сухий код кольору, а й відтінки, текстуру матеріалу та рівень насиченості, що забезпечує естетично привабливий результат підбору образів [8].

Стилістична когерентність та готовність до використання: система проводить глибокий семантичний аналіз приналежності речей до певних категорій (*Business, Casual, Sport, Formal*) на основі тегів, ідентифікованих автоматично. Важливим інноваційним компонентом є облік «статусу готовності» речі: система інтегрує дані про історію використань та прання, автоматично виключаючи з рекомендацій ті одиниці, що на даний момент потребують обслуговування або не відповідають актуальному дрес–коду запланованої події [8].

Контекстна валідність та погодна адаптація: унікальна функція автоматичного корелювання гардеробу з динамічними зовнішніми умовами. Система в реальному часі звертається до метеорологічних сервісів через API, аналізуючи не лише температуру, а й вірогідність опадів, силу вітру та вологість. На основі цих даних та розпізнаних характеристик одягу (щільність, довжина, тип виробу) інтелектуальний алгоритм відфільтровує варіанти, залишаючи лише ті комбінації, що відповідають поточному кліматичному комфорту користувача у конкретній локації [4].

Таким чином, сформовані функціональні вимоги до проектованої системи виходять далеко за межі можливостей звичайного цифрового каталогу. Система повинна забезпечувати глибоку багатофакторну аналітику, де користувач отримує не просто статичний список наявного одягу, а повноцінну, технологічно обґрунтовану стратегію його щоденної експлуатації. Ця стратегія базується на безперервному циклі обробки даних: від миттєвого аналізу зображення нейромережею до зіставлення результату з геопозицією, календарем подій та актуальним прогнозом погоди [8].

Реалізація такої амбітної моделі, де ШІ виступає як персональний стиліст–аналітик, вимагає розробки реляційної архітектури бази даних на основі PostgreSQL. Дана СУБД обрана за її здатність ефективно опрацьовувати множинні зв’язки типу «багато–до–багатьох» між категоріями, тегами, кольорами та динамічними атрибутами експлуатації речей, забезпечуючи високу швидкість запитів навіть при значних обсягах даних [10]. Це дозволяє системі миттєво генерувати варіанти образів, що є ідеальними за кольором, стилем та поточною погодною ситуацією.

1.3. Обґрунтування методології проектування та вибору інструментальних засобів реалізації

Вибір інструментарію для програмної реалізації інтелектуального додатку зумовлений необхідністю досягнення синергії між високою швидкістю розробки та екстремальною продуктивністю на стороні клієнта. Для побудови інтерфейсної частини застосунку було обрано фреймворк React Native. Наукова та інженерна доцільність цього вибору полягає у можливості використання нативного мосту (*Native Bridge*), що дозволяє безпосередньо звертатися до потужностей графічного процесора (GPU) мобільного пристрою. Це є критичним фактором для забезпечення плавної візуалізації розлогих галерей зображень високої роздільної здатності та виконання базових операцій перетворення (scaling, cropping, filtering) безпосередньо на пристрої користувача (*Edge computing*), що суттєво знижує навантаження на мережевий канал та серверні

потужності [8]. Використання компонентно-орієнтованого підходу React Native дозволяє створити декларативний інтерфейс, який легко адаптується під різні роздільні здатності екранів смартфонів.

Додатковою перевагою є використання мови TypeScript, що дозволяє впровадити сувору типізацію даних на всіх етапах розробки. У контексті даного проекту це мінімізує ризики виникнення логічних помилок при маніпуляції зі складними деревоподібними структурами об'єктів «Одяг», де кожен елемент має множинні атрибути – від колористичних характеристик та текстури тканини до динамічних статусів чистки, частоти використання та погодної відповідності. Статична типізація виступає додатковим шаром документації коду, що спрощує подальшу підтримку та масштабування проекту [10].

Для побудови бекенд–інфраструктури в межах дослідження обґрунтовано використання мікросервісного архітектурного підходу. На відміну від традиційної монолітної архітектури, яка притаманна більшості початкових проектів у цій ніші, мікросервіси дозволяють ізолювати найбільш ресурсомісткі обчислювальні процеси (як-от обробка нейромережами) від логіки обробки запитів користувачів. Центральним елементом системи обміну повідомленнями та координації сервісів обрано брокер RabbitMQ. Це рішення дозволяє реалізувати надійний патерн «Publish–Subscribe» та «Work Queues» для виконання асинхронних фонових завдань, що є життєво важливим для стабільної роботи з інтелектуальним аналізом зображень [11].

Алгоритм взаємодії в межах цієї архітектури виглядає наступним чином:

- Ініціація запиту та Optimistic UI: при завантаженні користувачем фотографії нової речі, основний API–сервер (реалізований на Node.js/NestJS) негайно підтверджує отримання файлу, забезпечуючи миттєвий відгук інтерфейсу. Завдання на сегментацію та глибокий візуальний аналіз інкапсулюється у форматі повідомлення (AMQP протокол) і надсилається в чергу RabbitMQ, що гарантує збереження даних навіть при тимчасовому збої обробника [11].

- Асинхронна інтелектуальна обробка: окремий спеціалізований worker–сервіс (реалізований на Python для кращої сумісності з ML–бібліотеками) вилучає завдання з черги. Він взаємодіє з OpenAI Vision API для семантичного опису або використовує власні моделі на базі PyTorch/TensorFlow для видалення фону та колористичного розбору. Результат обробки трансформується у структурований JSON та зберігається у базі даних [8].
- Real-time сповіщення через WebSockets**:** після успішного завершення обробки та оновлення стану в БД, система ініціює подію, яка через протокол WebSocket (Socket.io) миттєво інформує клієнтський застосунок про готовність результату. Це дозволяє уникнути застарілого методу періодичного опитування сервера (*polling*), що критично для економії заряду акумулятора мобільного пристрою.

Такий архітектурний підхід забезпечує майже необмежену горизонтальну масштабованість системи: при різкому зростанні кількості активних користувачів розробник може динамічно збільшити кількість примірників worker–сервісів (наприклад, у Docker–контейнерах), не вносячи змін до основного коду системи [11].

Щодо стратегії збереження даних, найбільш доцільним визначено використання реляційної СУБД PostgreSQL. Її вибір науково обґрунтований підтримкою складних реляційних запитів, що необхідні для багатофакторної фільтрації гардеробу наприклад, пошук комбінацій за кольором, стилем та поточною температурою одночасно. Потужний інструментарій для роботи з типом JSONB ідеально підходить для зберігання неструктурованих метаданих одягу, які можуть сильно варіюватися залежно від категорії речі, дозволяючи зберігати гнучкість схеми даних без втрати продуктивності [10].

Для оптимізації швидкодії та кешування результатів аналізу, а також керування сесіями користувачів та станами черг, передбачено інтеграцію In–memory бази даних Redis. Використання Redis як проміжного шару дозволяє

скоротити час відгуку системи при повторних запитах до позначки <100 мс, що є стандартом для сучасних високопродуктивних мобільних рішень [11].

Методологія дослідження в наступних розділах кваліфікаційної роботи базуватиметься на системному науковому підході, що включає:

- Об’єктно–орієнтоване проектування (OOAD) та моделювання: розробка детальних UML–діаграм (Use Case, Class, Sequence та State diagrams). Це дозволить формалізувати архітектурні зв’язки, визначити зони відповідальності кожного мікросервісу та спроектувати оптимальні потоки даних [10].
- Методологія Agile та ітеративна розробка: застосування гнучких методів дозволяє проводити постійне тестування та валідацію інтелектуальних модулів розпізнавання образів. Кожна ітерація завершується оцінкою точності нейронної мережі та корекцією алгоритмів підбору образів на основі зворотного зв’язку [10].
- Статистичний аналіз та верифікація результатів: оцінка якості роботи системи проводитиметься на валідаційному наборі даних (*Validation Dataset*), що включає тисячі анотованих зображень одягу. Будуть розраховані метрики точності (Accuracy), повноти (Recall) та F1-міри для підтвердження наукової гіпотези про ефективність обраних моделей ШІ [8].

У результаті проведеного теоретичного аналізу та критичного огляду предметної області було встановлено, що розробка інтелектуального асистента для управління гардеробом є актуальною відповіддю на системну кризу надлишкового споживання та пов’язаний із нею «когнітивний параліч» користувача. Цифровізація персональних речей через концепцію «цифрового двійника» дозволяє не лише впровадити принципи сталого розвитку (Sustainable Fashion) та розрахунок економічної ефективності (CPW), а й суттєво знизити психологічне навантаження на індивіда.

Проведений порівняльний аналіз існуючих комерційних рішень (Stylebook, Acloset, Cladwell) виявив низку критичних технологічних прогалин:

високий поріг входу через ручне оцифрування, низьку відмовостійкість синхронних архітектур та обмеженість алгоритмів рекомендацій, що ігнорують динамічний контекст (погоду, соціальні події, статус чистоти речі).

Обґрунтовано, що вирішення цих проблем лежить у площині використання передових інженерних підходів, а саме:

1. Архітектурна інновація: впровадження подієво-орієнтованої мікросервісної моделі з використанням брокера повідомлень RabbitMQ та WebSockets, що забезпечує асинхронну обробку важких даних (Computer Vision) без втрати відгуку інтерфейсу.
2. Інтелектуалізація: використання методів штучного інтелекту для автоматичної сегментації зображень, семантичного аналізу атрибутів одягу та багатокритеріальної оптимізації при формуванні образів.
3. Технологічний стек: вибір React Native (TypeScript) для клієнтської частини та Node.js/Python для бекенду дозволяє досягти балансу між кросплатформеністю та високою продуктивністю обчислень.
4. Таким чином, сформовано надійний теоретичний та методологічний фундамент для подальшого проектування системи, яка трансформує пасивний каталог одягу в активну експертну систему, здатну до автономної контекстної аналітики та підтримки прийняття рішень у реальному часі.

РОЗДІЛ 2. АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ ТА МОДЕЛЮВАННЯ СИСТЕМИ

2.1. Архітектурне моделювання системи на основі мікросервісного підходу та подієво-орієнтованої взаємодії

Проектування та розробка сучасних програмних комплексів, що інтегрують технології генеративного штучного інтелекту та потребують високого рівня доступності, дедалі частіше базуються на відмові від традиційної монолітної структури на користь гнучкої мікросервісної архітектури [11, 12]. Розробка інтелектуальної системи спирається на принципи глибокої декомпозиції функціональних блоків на незалежні сервіси, що дозволяє забезпечити надійну ізоляцію збоїв, можливість незалежного горизонтального масштабування та високу гнучкість технологічного стека для кожного окремого модуля. Вибір мікросервісного підходу для даного проєкту обумовлений критичною необхідністю розділення ресурсомістких завдань, пов’язаних із генерацією відповідей штучним інтелектом (модель Google Gemini) та обробкою великих обсягів медіафайлів, від типових транзакційних операцій читання та запису даних про предмети гардероба [13]. Такий розподіл дозволяє оптимізувати використання серверних ресурсів, забезпечуючи швидкий відгук інтерфейсу навіть під час виконання складних аналітичних обчислень у фоновому режимі, що є фундаментальним для позитивного користувацького досвіду в мобільних екосистемах.

Загальна концепція мікросервісного підходу передбачає створення набору автономних сервісів, кожен з яких відповідає за конкретну бізнес-функцію та має власну базу даних або ізольовану схему. На рисунку 2.1 продемонстровано схему застосунку, де клієнт звертається до системи через проміжний шар, а самі сервіси можуть обмінюватися даними між собою. Як показано на схемі архітектури, кожен мікросервіс функціонує як незалежна одиниця, що дозволяє розробнику обирати найбільш підходящі інструменти для вирішення конкретних задач та виділяти лише необхідну кількість ресурсів для їх вирішення.

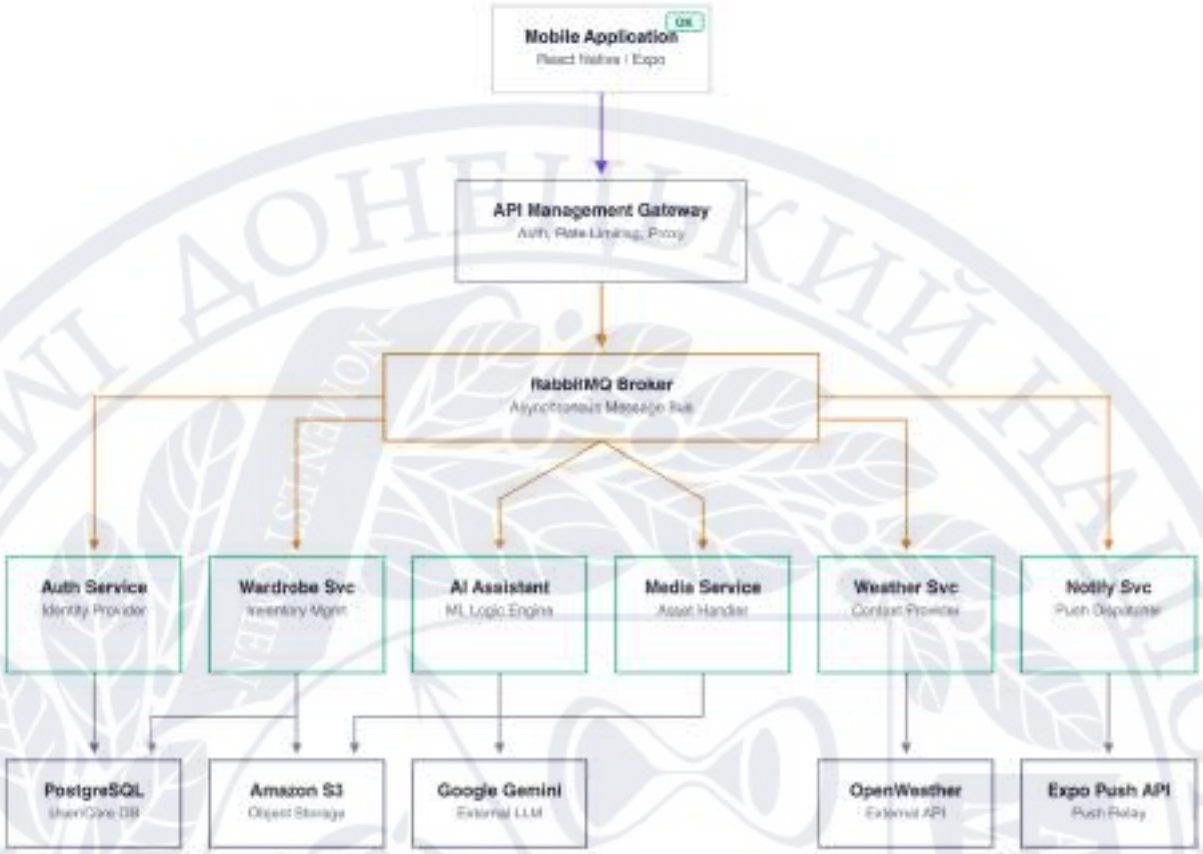

Рисунок 2.1 – Діаграма взаємодії мікросервісів застосунку

Центральним елементом побудови бекенд-частини системи виступає фреймворк NestJS, який базується на платформі Node.js [14, 15]. Даний інструментарій обрано через його високу архітектурну строгість, повну підтримку мови TypeScript та потужні вбудовані механізми для створення мікросервісних додатків [16]. NestJS дозволяє реалізувати модульну структуру всередині кожного окремого сервісу, суворо дотримуючись принципів «Clean Architecture» та «SOLID», що значно полегшує подальше тестування, супроводження та розширення коду. Застосування монорепозиторної структури дозволяє централізовано керувати спільними ресурсами через директорію libs, де розміщено конфігурації бази даних TypeORM, налаштування брокера повідомлень RabbitMQ та глобальні фільтри винятків [17]. Використання декораторів та ін’єкції залежностей (Dependency Injection) дозволяє абстрагувати логіку обміну повідомленнями від бізнес-логіки, роблячи систему стійкою до змін у транспортному рівні. Впровадження архітектурного шаблону API Gateway

дозволяє уніфікувати доступ до внутрішніх ресурсів системи, створюючи єдину точку входу для клієнтського застосунку на базі React Native, що спрощує моніторинг, безпеку та маршрутизацію трафіку [18, 19]. Завдяки інтеграції з модулем JWT, шлюз забезпечує надійну перевірку повноважень користувачів ще на етапі надходження запиту, не навантажуючи внутрішні бізнес-сервіси зайвими обчисленнями.

Для забезпечення комунікації між сервісами обрано подієво-орієнтовану модель (Event-Driven Architecture) з використанням брокера повідомлень RabbitMQ. На відміну від прямої взаємодії через HTTP/REST, використання RabbitMQ дозволяє реалізувати асинхронний обмін даними, що є критично важливим для стабільності мобільної екосистеми. Даний підхід забезпечує високий рівень відмовостійкості, оскільки сервіси залишаються слабко пов’язаними, а повідомлення можуть зберігатися в чергах до моменту їх успішної обробки цільовим сервісом. Це особливо актуально для завдань, пов’язаних з генеративним штучним інтелектом Google Gemini, які потребують значного часу для обробки та формування відповіді. Використання патерну Request-Response на базі RabbitMQ дозволяє клієнту отримувати результати обчислень у зручному форматі, зберігаючи при цьому всі переваги асинхронної шини даних. Вся внутрішня комунікація суворо типізована через спільні бібліотеки, що виключає помилки при передачі об’єктів між сервісами.

Кожен запит користувача проходить через шлюз API, який виконує роль єдиної точки входу (Entry Point). Шлюз не містить бізнес-логіки, а лише трансформує HTTP-запити у RPC-повідомлення (Remote Procedure Call) для відповідних черг брокера. Він виступає в ролі посередника, який мапує вхідні маршрути на конкретні ключі маршрутизації RabbitMQ, такі як AUTH_REQUESTS для авторизації або WARDROBE_REQUESTS для управління речами. Окрім маршрутизації, шлюз виконує валідацію вхідних даних за допомогою DTO (Data Transfer Objects) та механізмів class-validator, що гарантує цілісність інформації до її передачі у внутрішні мікросервіси. Такий розподіл обов’язків дозволяє ізолювати критичну інфраструктуру, наприклад

базу даних PostgreSQL та хмарне сховище AWS S3, від прямого доступу з мережі інтернет. Також шлюз реалізує механізм обмеження частоти запитів (Throttler), що захищає систему від перевантажень [20]. Детальний опис функціональних обов'язків та характеристик кожного мікросервісу наведено у таблиці 2.1.

Таблиця 2.1. Опис функціональних зон мікросервісів системи

Назва сервісу	Функціональна роль	База даних / Сховище	Ключові технології
API Gateway	Маршрутизація, Throttle, JWT	Відсутнє (Stateless)	NestJS, Passport
Auth	Управління користувачами та сесіями	PostgreSQL (user_account)	JWT, Bcrypt
Wardrobe	CRUD операції з одягом, лог образу	PostgreSQL (wardrobe_item)	TypeORM
AI Assistant	Обробка запитів до Gemini, контекст	PostgreSQL (sessions, jobs)	Gemini AI, OpenWeatherMap
Media Storage	Управління фотографіями в хмарі	AWS S3	AWS SDK, Presigned URLs

Ключовою архітектурною особливістю системи є впровадження спеціалізованої стратегії «Pre-fetched Context», яка спрямована на оптимізацію взаємодії між мікросервісами та великими мовними моделями. Замість надання моделі штучного інтелекту безпосереднього та неконтрольованого доступу до бази даних через інструменти на кшталт Function Calling, що суттєво підвищує ризики порушення безпеки та створює непередбачувані затримки у виконанні запитів, мікросервіс ai-assistant функціонує як інтелектуальний агрегатор даних. Такий підхід дозволяє здійснювати повний контроль над обсягом та конфіденційністю переданої інформації, оскільки сервіс виступає в ролі фільтра, який відсікає надлишкові технічні метадані та готує лише змістовний контекст, необхідний для прийняття рішення. Це гарантує, що модель Google Gemini

отримує структуровану інформацію, яка вже пройшла стадію попередньої обробки та валідації всередині екосистеми NestJS.

Процес формування відповіді ініціюється відразу після надходження запиту від користувача через шлюз API, після чого мікросервіс починає виконання чітко визначеного алгоритму агрегації. Першим етапом є звернення до внутрішнього сховища профілів для отримання поточного місця проживання користувача, яке зберігається у полі city. На основі цих географічних даних сервіс виконує зовнішній запит до OpenWeatherMap API для отримання актуального прогнозу погоди, включаючи температуру повітря, вологість та наявність опадів [21]. Наступним кроком є асинхронна комунікація з мікросервісом wardrobe через брокер повідомлень RabbitMQ для отримання повного списку речей, що мають статус ready. Це дозволяє відфільтрувати одяг, який на даний момент знаходиться у пранні або позначений як відсутній. Фінальним етапом алгоритму є конструювання складного системного промпту, де всі отримані змінні поєднуються з інструкціями щодо стилістики та правил комбінування кольорів.

Такий підхід забезпечує високий рівень детермінованості відповідей штучного інтелекту, оскільки кожна порада ґрунтується на об’єктивних факторах навколишнього середовища та реальному стані гардероба в конкретний момент часу. Це мінімізує ймовірність виникнення галюцинацій моделі, коли система могла б запропонувати одяг, який не відповідає погоднім умовам або просто недоступний для використання. Впровадження стратегії «Pre-fetched Context» дозволяє досягти оптимального балансу між швидкістю роботи системи та якістю генерації персоналізованих рекомендацій, що є критично важливим для забезпечення високого рівня задоволеності користувачів мобільним додатком.

Кожен мікросервіс у системі спроектований із суворим дотриманням принципів високої згуртованості (High Cohesion) та слабкої пов’язаності (Low Coupling). Це забезпечує високу автономність кожного вузла системи та дозволяє вносити зміни у бізнес-логіку одного модуля без ризику порушення цілісності всього програмного комплексу. Такий підхід є фундаментальним для розробки

розподілених систем, оскільки він спрощує процес тестування, налагодження та подальшого масштабування окремих компонентів. Розглянемо внутрішню організацію модуля на прикладі сервісу wardrobe, який є ключовим у системі управління цифровим гардеробом.

Внутрішня архітектура мікросервісу базується на трьох логічних рівнях, що відповідає принципам чистої архітектури (Clean Architecture):

1. Транспортний рівень (Transport Layer). Даний рівень представлений контролерами, які використовують декоратори @MessagePattern. Оскільки взаємодія між сервісами відбувається через брокер повідомлень RabbitMQ, контролери не обробляють прямі HTTP-запити, а слухають конкретні черги повідомлень. На цьому рівні виконується десеріалізація вхідних даних, їх первинна валідація за допомогою DTO (Data Transfer Objects) та передача очищених об’єктів до рівня бізнес-логіки. Це дозволяє ізолювати ядро системи від особливостей протоколу передачі даних.
2. Рівень бізнес-логіки (Business Logic Layer). Це ядро мікросервісу, реалізоване у вигляді класів із декоратором @Injectable(). Саме тут зосереджена вся інтелектуальна складова модуля: складні алгоритми фільтрації одягу, перевірка прав власності для забезпечення безпеки даних та специфічні бізнес-правила.
3. Рівень доступу до даних (Data Access Layer). Даний рівень забезпечує взаємодію з реляційною базою даних PostgreSQL за допомогою ORM-системи TypeORM. Використання паттерну «Repository» дозволяє абстрагувати складні SQL-запити та працювати з даними як із об’єктами TypeScript. Це забезпечує високу швидкість розробки та легкість внесення змін до схеми бази даних без необхідності переписування бізнес-логіки [22, 23, 24].

Для того, щоб модель штучного інтелекту Google Gemini могла генерувати релевантні та стилістично правильні поради, структура даних повинна підтримувати складні зв’язки та містити детальні метадані про кожну одиницю одягу. Кожне поле у базі даних було спроектовано з урахуванням потреб «Pre-

fetched Context» стратегії, де дані трансформуються у текстовий опис для нейронної мережі. Детальний опис схеми основних атрибутів сутності wardrobe_item наведено у таблиці 2.2.

Таблиця 2.2. Схема основних атрибутів сутності wardrobe_item та їх призначення для ШІ

Поле	**Тип даних**	**Опис**	**Призначення для ШІ**
id	UUID	Унікальний ідентифікатор об’єкта	Точна ідентифікація речі у відповіді
type	Enum	Тип одягу (Top, Bottom, Shoes, etc.)	Визначення шарів та структури образу
color	String	Основний колір виробу	Врахування колірної сумісності та стилю
season	Enum	Сезонність (Summer, Winter, etc.)	Фільтрація речей відповідно до погоди
status	Enum	Поточний стан (Ready, Washing, Missing)	Перевірка фізичної доступності речі
img_path	String	Шлях до медіафайлу в сховищі S3	Візуальна верифікація для користувача

Робота із зображеннями одягу в системі вимагає особливого архітектурного підходу через значний обсяг бінарних даних, які можуть створювати критичне навантаження на пропускну здатність мережі та обчислювальні ресурси бекенда. Для вирішення цієї проблеми сервіс media-storage реалізує паттерн «Pre-signed URLs», який базується на принципі делегування прав доступу. Це дозволяє мобільному додатку завантажувати фотографії високої якості безпосередньо з хмарного сховища AWS S3, повністю минаючи основний канал передачі даних між мікросервісами. Таке рішення є критично важливим для збереження стабільності брокера повідомлень RabbitMQ та шлюзу API Gateway, оскільки виключає необхідність серіалізації та передачі важких об’єктів (Buffer) через шину подій, що могло б призвести до затримок у роботі інших сервісів та переповнення черг [25].

Процес завантаження медіафайлів реалізований як багатоступенева безпечна транзакція, що включає наступну послідовність дій:

1. Ініціалізація та запит метаданих: під час активації інтерфейсу wardrobe мобільний застосунок ініціює HTTP-запит до wardrobe-api-gateway. Шлюз виконує валідацію сесії користувача та трансформує запит у RPC-повідомлення для мікросервісу wardrobe. На цьому етапі система ідентифікує перелік необхідних об’єктів, які належать конкретному користувачу, та готує їхні технічні ідентифікатори.
2. Динамічне формування безпечних посилань: оскільки прямий доступ до об’єктного сховища AWS S3 обмежений правилами безпеки (Private Access), мікросервіс wardrobe через шину повідомлень RabbitMQ звертається до сервісу media-storage. Останній, використовуючи метод getSignedUrl з AWS SDK, генерує тимчасовий Presigned GET URL для кожного зображення. Це посилання є криптографічно підписаним і вже містить у собі необхідні токени авторизації, параметри ідентифікації (X-Amz-Algorithm, X-Amz-Credential) та обмеження за часом життя (TTL – Time To Live).
3. Трансфер авторизованих посилань клієнту: Після агрегації всіх посилань, мікросервіс повертає сформовану відповідь через шлюз API до мобільного застосунку. У структурі JSON-відповіді внутрішні шляхи до файлів (S3 Keys) автоматично замінюються на повні зовнішні URL-адреси. Це дозволяє клієнтській стороні оперувати готовими ресурсами без необхідності додаткової автентифікації на боці сховища.
4. Пряма доставка контенту з хмари: рендеринг зображень у мобільному додатку здійснюється шляхом передачі отриманого URL безпосередньо в нативні компоненти зображення. Запит на отримання бінарних даних виконується клієнтом безпосередньо до серверів Amazon S3 за протоколом HTTPS.

Дана схема архітектурно мінімізує використання оперативної пам’яті (RAM) та процесорного часу (CPU) серверів бекенда, оскільки важкі бінарні

потоки даних не проходять через Node.js процеси та не потребують буферизації. Крім того, це підвищує загальну безпеку системи: сервіси не зберігають постійних ключів доступу до S3 на стороні клієнта, а тимчасові посилання стають невалідними одразу після закінчення ліміту часу. Таким чином, мікросервіс media-storage забезпечує максимальну продуктивність та масштабованість системи при роботі з мультимедійним контентом, що є необхідною умовою для візуально-орієнтованого додатку.

Оскільки мікросервіси в системі використовують спільну базу даних PostgreSQL, архітектурне проектування вимагає чіткого логічного розділення даних між доменами для запобігання жорсткій зв’язності на рівні схем. Попри фізичне перебування даних в межах одного екземпляра СУБД, кожен сервіс оперує лише своїм набором таблиць через власні сутності TypeORM, що робить неможливим використання традиційних каскадних видалень (ON DELETE CASCADE) на рівні бази даних для об’єктів, що належать різним сервісам. У зв’язку з цим, у системі застосовується принцип Eventual Consistency (кінцевої узгодженості), де синхронізація стану між функціональними блоками забезпечується через асинхронний обмін подіями.

Наприклад, при видаленні користувачем конкретної речі через інтерфейс застосунку, мікросервіс wardrobe видаляє запис із таблиці wardrobe_item. Однак, для повної цілісності системи необхідно також очистити пов’язані дані в інших доменах: видалити відповідні медіафайли з хмари AWS S3 та оновити історію або кеш рекомендацій у сервісі ai-assistant. Оскільки пряме звернення одного сервісу до таблиць іншого порушило б принцип ізоляції, процес реалізується через подієво-орієнтований алгоритм:

1. Емісія події: після успішного видалення запису про річ у своїй схемі, сервіс wardrobe публікує повідомлення wardrobe.item.deleted у RabbitMQ Fanout Exchange. Це повідомлення містить ідентифікатор речі (itemId) та посилання на файл зображення, яке потрібно видалити.
2. Асинхронна обробка в межах екосистеми: усі зацікавлені мікросервіси отримують копію цієї події:

3. Media Storage Service: отримує сигнал та ініціює запит до AWS S3 для фізичного видалення файлу, звільняючи місце у хмарному сховищі.
4. AI Assistant Service: оновлює контекст або видаляє застарілі логи рекомендацій, де згадувалася ця річ, щоб при наступному запиті не посилатися на неіснуючий об’єкт.

Такий підхід гарантує, що навіть при використанні спільної бази даних, система залишається гнучкою та відмовостійкою. Завдяки механізмам підтвердження (Message Acknowledgement) у RabbitMQ, подія про видалення буде опрацьована кожним сервісом гарантовано, навіть якщо в момент публікації один із них (наприклад, сервіс медіа через проблеми із зовнішнім API Amazon) був тимчасово недоступний. Це виклює ризик появи зайвих даних (зображень у S3, на які ніхто не посилається) та забезпечує цілісність інформаційної моделі додатку.

Архітектурне моделювання програмного комплексу на основі мікросервісного підходу та брокера повідомлень RabbitMQ забезпечує необхідний рівень масштабованості та гнучкої ізоляції бізнес-логіки. Використання спільної бази даних PostgreSQL із логічним розподілом доменів дозволяє оптимізувати інфраструктурні витрати й спростити адміністрування системи, зберігаючи при цьому ключові переваги мікросервісної архітектури. Вибір фреймворку NestJS як базового інструментарію став фундаментом для реалізації строгої типізації та модульності, що є критичним фактором для стабільної роботи та довгострокової підтримки проекту.

Розроблена стратегія «Pre-fetched Context» для взаємодії зі штучним інтелектом та впровадження патерну «Pre-signed URLs» для обробки медіаданих дозволили розв’язати проблему продуктивності при роботі з ресурсомісткими операціями. Спроектована архітектура демонструє високу готовність до практичної експлуатації, забезпечуючи швидкий відгук інтерфейсу, надійний захист даних та ефективну координацію між ізольованими функціональними блоками через асинхронну шину подій. Описані методи взаємодії та структури

даних, систематизовані у таблицях 2.1 та 2.2, формують цілісний технічний базис для подальшої програмної реалізації всіх модулів системи.

2.2. Розробка інформаційної моделі даних та структури бази знань гардероба

Ефективне функціонування інтелектуальної системи безпосередньо залежить від якості та глибини структури бази даних, яка спроектована не просто як статичне сховище записів про предмети одягу, а як динамічна база знань, що надає змістовний контекст для генеративної моделі штучного інтелекту. Для реалізації надійного, продуктивного та масштабованого сховища було обрано реляційну систему керування базами даних PostgreSQL, яка відома своєю підтримкою складних типів даних та високою швидкістю виконання аналітичних запитів. Логічна структура бази даних побудована за суворими принципами нормалізації (до третьої нормальної форми включно), що дозволяє мінімізувати надликовість даних та забезпечити цілісність інформації в межах розподіленої мікросервісної архітектури. Це також спрощує горизонтальне масштабування окремих сервісів та забезпечує швидку синхронізацію станів через брокери повідомлень без ризику виникнення аномалій оновлення чи видалення.

Особлива увага при проектуванні приділялася здатності системи трансформувати технічні параметри: тканини, сезонність чи колірна палітра) у семантично багаті структури, зрозумілі для моделі Google Gemini. Завдяки використанню PostgreSQL, система ефективно поєднує класичні реляційні зв’язки між сутностями з можливістю гнучкого розширення атрибутів, зокрема через використання індeforceованих JSONB-полів для збереження слабоструктурованих метаданих. Такий гібридний підхід дозволяє формувати релевантні prompt-контексти «на льоту», агрегуючи реляційні дані гардероба користувача в єдиний структурований JSON-об`єкт, який передається ШІ-моделі для генерації персоналізованих порад. Повна логічна та фізична структура бази даних, яка відображає всі ключові сутності, їхні атрибути, типи полів та зовнішні зв`язки (foreign keys), детально зображена на рисунку 2.2.

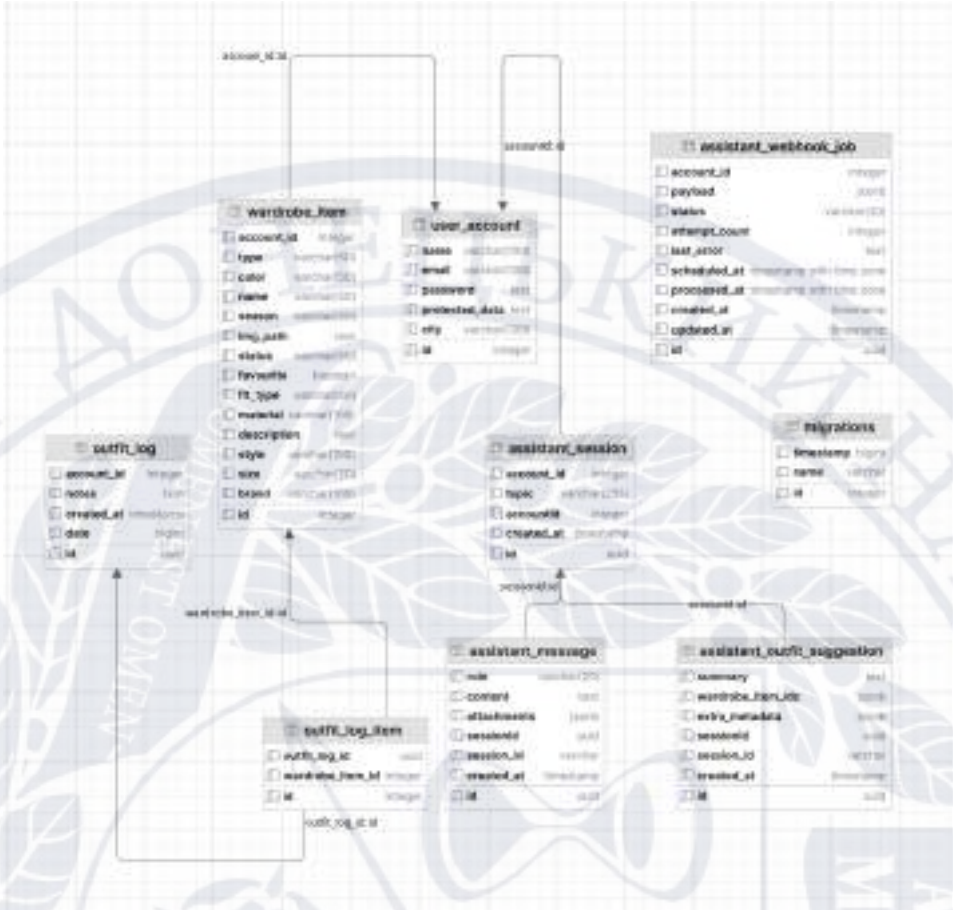

Рисунок 2.2. – Схема реляційної моделі даних та зв’язків між сутностями бази даних застосунку

Інформаційна модель системи базується на чотирьох ключових сутностях, які формують повне уявлення про стан гардероба користувача та історію його взаємодії з асистентом:

1. user_account (обліковий запис користувача) Ця сутність виконує роль фундаменту персоналізації в системі. Окрім стандартних атрибутів безпеки, таких як UUID (для унікальної ідентифікації в мікросервісному середовищі), валідований email та хеш пароля, вона містить ключові метадані профілю. Поле city є критичним вузлом інтеграції: воно ініціює ланцюжок запитів до OpenWeatherMap API, результати яких стають частиною системного промпту для ШІ.
2. wardrobe_item (предмет гардероба) Центральна одиниця бази знань, що представляє цифровий двійник реальної речі. Кожна одиниця описується через набір атрибутів. Атрибут type (верхній одяг, низ, взуття тощо) дозволяє ШІ структурувати образ за шарами. Поле color використовується

для аналізу колірної сумісності, а season – для відсікання неактуальних речей. Стан status (Ready, Washing, Missing) виступає динамічним фільтром: асистент ніколи не запропонує річ, яка перебуває у пранні, що забезпечує практичну цінність кожної поради. Також сутність містить img_path, що є ключем до об'єкта в AWS S3.

3. outfit_log (журнал образів) Ця сутність реалізує механізм «пам'яті стилю». Вона фіксує комбінації речей, які користувач обрав для виходу в конкретну дату. Окрім функції особистого щоденника, цей журнал слугує обмежувальним фактором для ШІ: алгоритм аналізує записи за останні 3 – 5 днів, щоб уникнути монотонності у рекомендаціях. Таблиця зв'язків (many-to-many між outfit_log та wardrobe_item) дозволяє системі швидко розраховувати частоту використання кожної речі, виділяючи «фаворитів» та ті предмети гардероба, які використовуються рідко, що може стати підґрунтям для порад щодо оновлення гардероба.
4. assistant_session (сесія асистента) Сутність, що забезпечує неперервність та контекстуальність взаємодії з моделлю Gemini. Замість обробки кожного запиту як ізольованого, система групує повідомлення в межах сесії. Це дозволяє користувачеві ставити уточнювальні запитання наприклад: «А якщо піде дощ?» або «Заміни ці штани на джинси», без необхідності заново описувати всі вхідні умови. Журнал зберігає не лише текст запиту, а й згенеровані структуровані JSON-об'єкти з рекомендаціями, що дозволяє миттєво відновити попередньо запропонований варіант образу при повторному відкритті вікна чату.

Враховуючи високі вимоги сучасних мобільних екосистем до швидкості рендерингу та завантаження медіаконтенту, у системі реалізовано гібридну модель зберігання даних. Цей підхід передбачає чіткий розподіл за типом інформації: у той час як структуровані текстові метадані (назви, описи, технічні характеристики речей) зберігаються у реляційних таблицях PostgreSQL, «важкі» фотографії одягу у високій роздільній здатності розміщуються в спеціалізованому хмарному об'єктному сховищі AWS S3. Таке розмежування

дозволяє базі даних залишатися компактною та швидкодійною, тоді як медіафайли зберігаються в інфраструктурі, що спеціально оптимізована для роздачі контенту з мінімальними затримками.

Для забезпечення безкомпромісного рівня безпеки персональних даних у поєднанні з високою продуктивністю, реалізовано інтелектуальну логіку Pre-signed URLs. Ця архітектурна модель дозволяє повністю відмовитися від публічного доступу до приватних файлів користувачів у бакетах S3, що виключає можливість несанкціонованого перегляду контенту сторонніми особами. Щоразу, коли мобільний застосунок потребує завантажити нове фото або відобразити наявну галерею, мікросервіс media-storage генерує тимчасове, криптографічно підписане посилання. Це посилання містить унікальний токен автентифікації, що видається під конкретний запит, роблячи пряму взаємодію з хмарою максимально захищеною [25].

Впровадження стратегії Pre-signed URLs вирішує проблему безпеки. Файли в S3-бакеті за замовчуванням залишаються закритими для всього інтернету. Доступ до них надається динамічно лише на обмежений проміжок часу після чого посилання автоматично стає невалідним. Це нівелює ризики витоку даних навіть у разі перехоплення URL-адреси, оскільки термін її дії є обмеженим для використання зловмисниками.

2.3. Модель користувацького інтерфейсу та людино-машинної взаємодії мобільного застосунку

Перехід від технічної моделі мікросервісного бекенда до безпосередньої взаємодії з користувачем реалізовано через кросплатформний мобільний застосунок на базі фреймворку React Native та екосистеми Expo. Людино-машинна взаємодія у системі спроектована таким чином, щоб максимально мінімізувати когнітивне навантаження на користувача. Це досягається шляхом трансформації складних процесів агрегації даних, запитів до брокера повідомлень та взаємодії з ШІ у зрозумілий, інтуїтивний та естетично привабливий візуальний досвід. Головний акцент зроблено на швидкодії та

відсутності зайвих кроків: користувач отримує персоналізовану пораду лише одним натисканням, у той час як фронтенд-частина виконує складну координацію асинхронних потоків даних.

Архітектура клієнтської частини побудована на фундаментальному принципі розділення обов'язків (Separation of Concerns). У цій парадигмі інтерфейс (UI) виконує роль виключно презентаційного рівня, який відображає поточний стан (State), тоді як вся складна логіка взаємодії з API, обробка помилок та трансформація даних винесена в ізольований сервісний шар. Це не тільки спрощує підтримку коду, але й дозволяє легко масштабувати функціонал застосунку, додаючи нові мікросервіси без необхідності радикальної перебудови візуальних компонентів.

Сервісна абстракція системи базується на централізованому класі HttpService, який є єдиною точкою входу для всієї мережевої комунікації. Він функціонує як інтелектуальна обгортка над HTTP-клієнтом, виконуючи наступні критичні функції:

- Управління авторизацією: Автоматичне впровадження Bearer-токенів (JWT) у заголовки кожного запиту, що звільняє розробника від ручного керування безпекою в кожному окремому сервісі.
- Перехоплення відповідей (Interceptors): Глобальна обробка помилок (наприклад, 401 Unauthorized), що дозволяє миттєво реагувати на завершення сесії користувача та перенаправляти його на екран логіна.
- Спеціалізовані сервіси: Високорівневі класи, такі як AuthService (керування сесіями), WardrobeService (маніпуляція предметами одягу) та AiService (взаємодія з асистентом), використовують HttpService для виконання конкретних бізнес-операцій.

Така багаторівнева абстракція робить код компонентів максимально «чистим» – вони оперують лише бізнес-об'єктами, не знаючи деталей мережевих протоколів. Це значно підвищує придатність системи до автоматизованого тестування та забезпечує стабільну роботу застосунку навіть у складних умовах мережевого з'єднання.

Питання безпеки персональних даних та захисту конфіденційної інформації користувачів є фундаментальним пріоритетом архітектури. Враховуючи вразливість мобільних пристроїв до певних видів атак, для зберігання критичних даних на стороні клієнта було обрано професійні інструменти, що базуються на апаратному шифруванні:

Secure Storage та криптографічний захист: JWT-токен (Access Token), що видається сервером після успішної автентифікації, є ключем до персональної бази знань гардероба. Його зберігання у стандартному AsyncStorage є неприпустимим, оскільки дані там зберігаються у відкритому текстовому вигляді та можуть бути прочитані у разі компрометації пристрою. Замість цього в проекті впроваджено бібліотеку expo-secure-store. Вона задіює нативні механізми операційних систем: Keychain для iOS та Keystore для Android. Це гарантує, що дані шифруються на рівні файлової системи пристрою за допомогою апаратних модулів безпеки, а доступ до них має виключно застосунок.

Інтелектуальне управління життєвим циклом сесії: Процес контролю доступу інтегрований безпосередньо в життєвий цикл мобільного застосунку. При кожному «холодному» запуску або поверненні з фонового режиму сервіс AuthService ініціалізує процедуру верифікації. Якщо в захищеному сховищі знайдено валідний токен, система здійснює безшовний вхід у внутрішній контур app/(app), забезпечуючи високий рівень UX-комфорту без необхідності повторного введення пароля.

Реактивна обробка завершення сесії: Для підтримки безпеки в реальному часі використовується механізм HTTP-інтерцепторів (interceptors) у межах HttpService. Якщо сервер повертає помилку 401 Unauthorized (наприклад, через закінчення терміну дії токена або його відкликання на бекенді), застосунок реагує миттєво:

1. Автоматично очищується захищене сховище.
2. Глобальний стан авторизації скидається.

3. Користувач перенаправляється на екран входу за допомогою router.replace, що блокує можливість повернення на захищені екрани за допомогою кнопки «Назад».

Така модель управління сесіями створює надійний захисний бар'єр. Навіть у разі фізичної втрати пристрою або спроб несанкціонованого доступу до файлової системи, дані користувача залишаються зашифрованими на рівні ОС. Це робить людино-машинну взаємодію не лише зручною та швидкою, а й повністю відповідною сучасним галузевим стандартам безпеки мобільних застосунків. Використання нативних засобів шифрування у поєднанні з архітектурним розподілом маршрутів забезпечує цілісність екосистеми та високу довіру з боку користувача.

Для забезпечення візуальної цілісності та технічної гнучкості інтерфейсу застосунку було розроблено авторську дизайн-систему, в основу якої покладено концепцію дизайн-токенів. Такий підхід дозволяє абстрагуватися від прямого вставлення конкретних значень у коді компонентів, замінюючи їх семантичними змінними. Усі ключові параметри – від кольорової палітри до сітки відступів та радіусів закруглення – централізовано зберігаються та зчитуються з глобальних конфігураційних файлів (theme/colors.ts, theme/layout.ts), що забезпечує миттєву синхронізацію стилів у всьому додатку.

Кольорові токени та естетика. Візуальна мова застосунку базується на використанні висококонтрастної палітри, де домінують чисті фонові кольори у поєднанні з акцентними тонами. Така стратегія обрана не випадково: вона дозволяє змістити фокус безпосередньо на контент – фотографії одягу користувача. Інтерфейс навмисно позбавлений надлишкового декору, що наближає його естетику до сторінок мінімалістичного цифрового модного журналу. Використання токенів дозволяє легко реалізувати підтримку світлої та темної тем, адаптуючи інтерфейс під системні налаштування пристрою без дублювання логіки стилів.

Окрему роль у людино-машинній взаємодії відіграють статус-беджі для предметів одягу. Вони використовують кольорове кодування для миттєвого інформування про доступність речі:

- Зелений (Ready): Річ доступна для формування образу.
- Блакитний (Washing): Річ у пранні, автоматично виключається з пропозицій ШІ.
- Сірий (Missing): Річ позначена як відсутня.

Типографіка та іконографіка Для покращення читабельності тексту в умовах мобільного використання було впроваджено ієрархічну систему типографіки. Токени розмірів шрифтів та міжрядкових інтервалів підібрані таким чином, щоб забезпечити чітке візуальне розділення між заголовками та описовими характеристиками речей. Використання векторної іконографіки в поєднанні з чіткими лейблами дозволяє користувачу інтуїтивно навігувати додатком, навіть при першому знайомстві з інтерфейсом.

Така комплексна побудова дизайн-системи не лише підвищує якість користувацького досвіду, але й значно прискорює розробку нових екранів, оскільки розробник оперує готовим набором перевірених компонентів та стилістичних правил.

Для організації навігації та структури додатку використано Expo Router – сучасне рішення для файлової маршрутизації в екосистемі React Native, яке базується на концепціях, аналогічних Next.js. Такий підхід дозволив створити чітку, декларативну та безпечну ієрархію екранів, де структура файлів у каталозі app безпосередньо відображає карту переходів користувача. Використання груп маршрутів дозволило сегментувати застосунок на логічні контури:

- Авторизаційний контур (app/(auth)): Повністю відокремлений на рівні файлової структури. Це гарантує ізоляцію логіки входу та реєстрації від основного коду застосунку. Таке розділення спрощує підтримку та запобігає випадковим витокам логіки авторизації в бізнес-функціонал.
- Захищений контур застосунку (app/(app)): Містить основний бізнес-функціонал (головний екран, гардероб, чат з ШІ). Використання груп

маршрутів у поєднанні з механізмом Layout дозволяє реалізувати автоматичний захист конфіденційних даних. Система перевіряє наявність активної сесії та валідність токена в SecureStore ще до моменту рендерингу цільової сторінки. Якщо користувач не авторизований, архітектура навігаційного графа виконує миттєвий редирект, що унеможливлює доступ до персональних даних навіть за прямою адресою екрана.

Візуальна архітектура застосунку спроектована за принципом Bottom Tab Navigation, що є золотим стандартом для сучасних мобільних систем. Це забезпечує високу ергономіку та дозволяє користувачу виконувати навігацію «одним великим пальцем», що критично важливо для додатків щоденного використання. Основний простір застосунку розподілено між чотирма стратегічними екранами:

Головний екран (Home): Слугує інтелектуальним дашбордом. Він інтегрує дані з метеорологічних сервісів у вигляді динамічного віджета погоди та відображає миттєву персоналізовану рекомендацію образу, сформовану ШІ. Це «центр управління», де користувач за лічені секунди отримує відповідь на головне запитання: «Що мені сьогодні вдягнути, враховуючи прогноз та мій розклад?».

Цифровий гардероб (Wardrobe): Високопродуктивна інтерактивна галерея, де відображаються цифрові копії всіх збережених речей. Тут реалізовано складну логіку фільтрації за категоріями (верхній одяг, низ, взуття), сезонами та поточними статусами (у пранні, готово до використання). Це дозволяє користувачу миттєво оцінити склад свого інвентарю та керувати наявністю речей.

Інтелектуальний асистент (AI Chat): Спеціалізований інтерфейс для глибокої діалогової взаємодії з моделлю Google Gemini. На відміну від статичних рекомендацій на головному екрані, чат дозволяє вести контекстуальний діалог: уточнювати запити наприклад – «Підбери щось більш офіційне», отримувати стилістичні поради або планувати образи для майбутніх подорожей.

Профіль користувача (Profile): Модуль керування персональною екосистемою. Тут зосереджені інструменти для редагування даних профілю, зміни геопозиції (міста) для коректної роботи погодного модуля, а також налаштування конфіденційності та параметрів відображення інтерфейсу.

Така структура не лише спрощує користувацький шлях, але й робить застосунок масштабованим: завдяки Expo Router додавання нових розділів або вкладених маршрутів відбувається без порушення цілісності вже існуючої навігаційної логіки. Повну схему користувацького інтерфейсу застосунку можна побачити на рисунку 2.3.

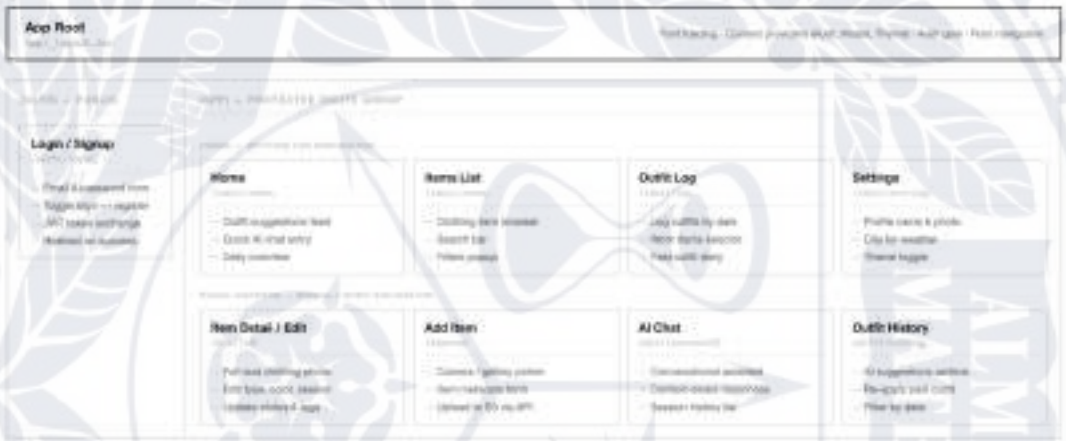

Рисунок 2.3 – Структурна схема графічного інтерфейсу застосунку

Для суттєвого покращення користувацького досвіду (UX) та зниження когнітивного бар'єру на початковому етапі взаємодії, у додатку впроваджено концепцію гібридної форми авторизації. Традиційна модель, що передбачає розділення процесів реєстрації та логіну на різні екрани, часто стає причиною відмов (churn rate) через необхідність зайвих переходів. Замість цього система використовує єдине інтелектуальне вікно входу, яке адаптується до статусу користувача в реальному часі.

Такий підхід дозволяє мінімізувати кількість кліків та переходів між екранами, що критично важливо для утримання користувача під час першого знайомства з продуктом. Технічно це реалізується через реактивні стани в React Native, які миттєво змінюють заголовки та написи на кнопках зі збереженням вже введених даних.

Крім того, гібридна форма інтегрована з механізмами валідації на стороні клієнта. Це гарантує, що запит до сервісу auth буде надіслано лише після того, як введена пошта пройде перевірку на коректність формату, а пароль – на відповідність вимогам безпеки. У підсумку, така модель ідентифікації не лише спрощує шлях користувача, а й підвищує надійність системи, виключаючи зайві запити з некоректними даними.

Найскладнішим аспектом людино-машинної взаємодії у системі є управління станом під час очікування відповідей від генеративної моделі ШІ. Оскільки генерація персоналізованих рекомендацій через Google Gemini може займати тривалий час, критично важливо забезпечити плавний та інформативний інтерфейс. Для вирішення цієї задачі обрано гібридну модель управління станом (State Management), що поєднує декларативний підхід React із реактивним програмуванням.

Глобальний стан через React Context API Для зберігання високорівневих станів, які використовуються у всьому додатку, впроваджено React Context API. Це забезпечує стабільний доступ до критично важливої інформації на будь-якому рівні дерева компонентів без необхідності «прокидання» даних (prop drilling).

- Auth Context: Керує даними авторизованого користувача, станом токена та правами доступу.
- Theme Context: Відповідає за динамічну зміну візуальних токенів (світла/темна теми) та адаптацію стилів інтерфейсу.

Для обробки складних асинхронних процесів, зокрема багатоетапної взаємодії з мікросервісом ai-assistant, впроваджено бібліотеку RxJS. На відміну від стандартних промісів (Promises), реактивні потоки (Observables) дозволяють реалізувати більш гнучку модель оновлення інтерфейсу:

- Керування потоком подій: коли застосунок надсилає запит до асистента, RxJS-суб’єкт транслює серію подій: початок запиту, прогрес генерації (loading states) та отримання фінального результату.

- Обробка переривань: використання операторів RxJS дозволяє легко скасовувати попередні запити, якщо користувач змінив параметри або покинув чат, що оптимізує навантаження на мережу та бекенд.

Якість відповідей штучного інтелекту прямо залежить від точності вхідних даних. Для забезпечення чистоти інформації, що потрапляє до системи, використовується зв’язка Formik та Yup.

- Formik оркеструє стан форм додавання одягу, відстежуючи кожну зміну атрибутів (колір, тип, сезон).
- Yup виконує сувору схематичну валідацію. Це гарантує, що до бази знань PostgreSQL потрапить лише повна та коректна інформація. Наприклад, система не дозволить зберегти річ без вказання сезону, оскільки це зробило б неможливим адекватний аналіз образу моделлю Gemini.

У підсумку, така архітектура фронтенду дозволяє приховати технологічну складність розподіленої системи за елегантним та чуйним інтерфейсом. Користувач взаємодіє з додатком як з цілісним інтелектуальним партнером, не відчуваючи затримок або технічних бар’єрів, притаманних складним мікросервісним рішенням.

РОЗДІЛ 3. РОЗРОБКА ЗАСТОСУНКУ ТА ТЕСТУВАННЯ

3.1 Розробка серверної частини застосунку

Серверний сегмент розробленого програмного забезпечення спроєктовано та реалізовано на засадах розподіленої мікросервісної архітектури на базі технологічної платформи Node.js із використанням фреймворку NestJS версії 10, що забезпечує високу модульність, керованість та масштабованість системи. З метою оптимізації процесів розробки, супроводу та розгортання, всі автономні мікросервіси інтегровано в єдиний монорепозиторій під управлінням інструментарію Nx Workspace, що дозволило організувати ефективне спільне використання загальносистемних бібліотек (зокрема базового модуля @app/common), забезпечити уніфіковану конфігурацію компілятора TypeScript, а також гарантувати централізоване керування залежностями й версіями сторонніх пакетів [16, 26]. Міжсервісна комунікація в межах системи є асинхронною та базується на подієво-орієнтованій моделі, де центральним компонентом виступає брокер повідомлень RabbitMQ, що функціонує за протоколом AMQP. Процес обміну даними регулюється маршрутизацією та ідентифікацією запитів за допомогою унікальних рядкових констант, які імпортуються зі спільної бібліотеки типів. Безпосередня комунікація між API-шлюзом та цільовими сервісами відбувається за шаблоном «Request-Response», що дозволяє шлюзу ініціювати асинхронне очікування результату виконання операції без блокування головного потоку обробки подій Event Loop. При цьому кожен мікросервіс підключається до виділеної черги повідомлень брокера RabbitMQ, де для запобігання втрати даних у разі виникнення непередбачуваних збоїв параметр автоматичного підтвердження встановлено у значення noAcK: false, завдяки чому видалення повідомлення з черги відбувається виключно після явного отримання сервером підтвердження про успішне завершення його обробки на стороні сервісу-споживача.

Для забезпечення надійної акумуляції, структурування та довготривалого зберігання інформації в системі розгорнуто реляційну систему керування базами

даних PostgreSQL, взаємодія з якою реалізована за допомогою технології Object-Relational Mapping на базі фреймворку TypeORM [23]. Це дозволило детально описити реляційну схему даних через класи-сутності мови TypeScript, які успадковують базовий клас із обов'язковими системними атрибутами: автоінкрементним первинним ключем id, а також часовими мітками створення createdAt та модифікації запису updatedAt. Логічна структура бази даних представлена такими сутностями:

- UserAccountEntity – зберігає облікові та автентифікаційні дані користувачів (електронну пошту, криптографічний хеш пароля, ім'я та місто) і пов'язана відношенням «один-до-багатьох» (OneToMany) із сутностями гардероба та діалогів;
- WardrobeItemEntity – містить характеристики елементів одягу (тип, колір, сезонність, поточний статус, шлях до файлу зображення, ознаку додавання до обраного) та зовнішній ключ accountId для забезпечення ізоляції та безпеки даних на рівні користувачів;
- OutfitLogEntity – фіксує історію формування готових образів, де для уникнення надлишкового дублювання рядків у проміжних таблицях ідентифікатори вибраних елементів одягу зберігаються в одному полі itemIds у форматі масиву JSON разом із текстовим описом наряду.

З метою підготовки шару даних до експлуатації у продакшн-середовищі параметр автоматичної синхронізації структури synchronize примусово переведено у стан false, тому будь-які зміни в архітектуру вносяться виключно шляхом застосування міграцій TypeORM перед розгортанням нової версії системи. Для мінімізації часу виконання найбільш частотних операцій фільтрації активних елементів одягу конкретного користувача було створено складений індекс за атрибутами (accountId, status) на базі таблиці wardrobe_item, а всі параметри підключення до бази даних зчитуються динамічно зі змінних системного оточення за допомогою сервісу ConfigService.

Безпеку системи та розмежування прав доступу забезпечує функціональний мікросервіс автентифікації (auth), який реалізує дворівневу

специфікацію верифікації користувачів. Первинну перевірку автентичності здійснює метод validateUser(), який виконує пошуковий запит у базі даних за електронною поштою та за допомогою сервісу BcryptService проводить криптографічне порівняння надісланого пароля у відкритому вигляді з його збереженим соленим хешем. У випадку успішного проходження перевірки метод signIn() ініціює процедуру генерації сесійного токена доступу за стандартом JWT (JSON Web Token), інкапсулюючи в його корисне навантаження некритичні ідентифікаційні дані (id, name, email) та підписуючи його через JwtService [27]. Паралельно з цим метод signup() забезпечує реєстрацію нових облікових записів, де перед збереженням даних у базу даних пароль піддається обов'язковому односторонньому хешуванню через алгоритм bcrypt із коефіцієнтом обчислювальної складності 10 [28]. Для створення уніфікованого досвіду взаємодії користувача з інтерфейсом (UX) метод реєстрації повертає об'єкт структури AuthUserAccount, повністю ідентичний об'єкту стандартної авторизації.

На рисунку 3.1 наведено програмну реалізацію методу signIn(), що демонструє фінальний етап авторизації користувача та випуск сесійного токена, який дійсний протягом семи днів.

```
public async signIn(user: UserAccountPreview): Promise<AuthUserAccount> {
  const tokenPayload = {
    id: user.id,
    name: user.name,
    email: user.email,
  };

  const accessToken : string  = await this.jwtService.signAsync(tokenPayload);

  return { accessToken, ...user };
}
```

Рисунок 3.1 – Програмна реалізація методу signIn

Мікросервіс також надає функціонал update-profile для зміни імені та міста, що безпосередньо впливає на функцію отримання прогнозу погоди: оновлене місто використовується при наступному зверненні до AI-асистента без потреби в перезапуску сервера.

Центральним компонентом предметної логіки розроблюваної системи виступає мікросервіс керування цифровим гардеробом (wardrobe), який забезпечує виконання повного спектра CRUD-операцій (Create, Read, Update, Delete) над сутністю wardrobe_item, а також здійснює ведення й актуалізацію журналу сформованих образів у таблиці outfit_log. Однією з ключових архітектурних особливостей даного сервісу є реалізація методу getItemsWithImageUrls(), спрямованого на оптимізацію взаємодії з шаром збереження медіаданих. Метод акумулює відносні шляхи до файлів усіх запитуваних предметів одягу та делегує їх обробку сервісу MediaStorageService в межах одного пакетного (батчевого) запиту, що дозволяє повністю нівелювати класичну проблему надлишкового навантаження «N+1 запитів» на рівні прикладного інтерфейсу. Процедура видалення елементів гардероба реалізована за транзакційним принципом із чіткою послідовністю дій: на першому етапі ініціюється деструкція відповідного медіафайлу з хмарного сховища AWS S3, і лише після успішного підтвердження цієї операції відбувається остаточне видалення бізнес-запису з реляційної бази даних. Такий підхід унеможливлює появу незв’язаних або «сирітських» (orphan) файлів у сховищі, забезпечуючи сувору цілісність даних системи.

На рисунку 3.2 наведено програмну реалізацію методу delete(), яка демонструє описаний алгоритм безпечного видалення сутності та пов’язаного з нею медіаконтенту.

```
173     async delete(id: number, accountId: number) {  Show usages  ± rostyslav48
174       const item = await this.wardrobeItemRepository.findOneByOrFail({
175         id,
176         accountId,
177       });
178
179       if (item.img_path) {
180         await this.mediaStorageService.delete(item.img_path);
181       }
182
183       return this.wardrobeItemRepository.delete({ id, accountId });
184     }
```

Рисунок 3.2 – Програмна реалізація методу delete

Метод вибірки даних findAll() проективного модуля гардероба підтримує багатокритеріальну фільтрацію за такими атрибутами, як тип (type), сезонність (season), колірна гама (color), поточний статус (status) та ознака пріоритетності (favourite). Обробка вхідних фільтрів відбувається динамічно за допомогою вбудованого інструментарію ORM-фреймворку TypeORM шляхом формування об’єкта умов where, при цьому всі параметри зі значенням undefined автоматично ігноруються, що дозволяє гнучко опрацьовувати частково заповнені пошукові запити без необхідності написання ручного або низькорівневого SQL-коду. У свою чергу, сервіс OutfitLogService, який відповідає за координацію готових образів, зберігає масив ідентифікаторів предметів одягу у спеціалізованому бінарному форматі JSONB. Під час зчитування запису сервіс виконує зворотну десеріалізацію та розгортає масив ідентифікаторів у повноцінні об’єкти сутностей за допомогою одноразового оптимізованого запиту findByIds().

Взаємодію з файловими ресурсами покладено на мікросервіс media-storage, який реалізує гнучку абстракцію над хмарною інфраструктурою Amazon Web Services (AWS S3) за допомогою патерна «Стратегія» (*Strategy*) [24]. Вибір конкретної програмної реалізації інтерфейсу DiskUtil визначається динамічно на основі аналізу змінної системного оточення NODE_ENVIRONMENT:

- У тестовому та локальному середовищах розгортання активується клас LocalDiskUtil, який здійснює збереження та зчитування файлів, використовуючи локальну файлову систему сервера.
- У виробничому середовищі система задіює клас S3DiskUtil, який реалізує пряму взаємодію з об’єктним сховищем AWS S3 через офіційний інструментарій @aws-sdk/client-s3 [29].

Для запобігання колізіям імен при завантаженні контенту, метод upload() виконує генерацію унікальних назв файлів на основі стандарту UUID з обов’язковим збереженням вихідного розширення файлу. Метод getSignedUrl() формує тимчасове підписане посилання (Pre-signed URL), термін дії якого обмежено константою у 3600 секунд. Графічне представлення

процесу автентифікації та завантаження зображень користувача за допомогою механізму Pre-signed URL наведено на рисунку 3.3.

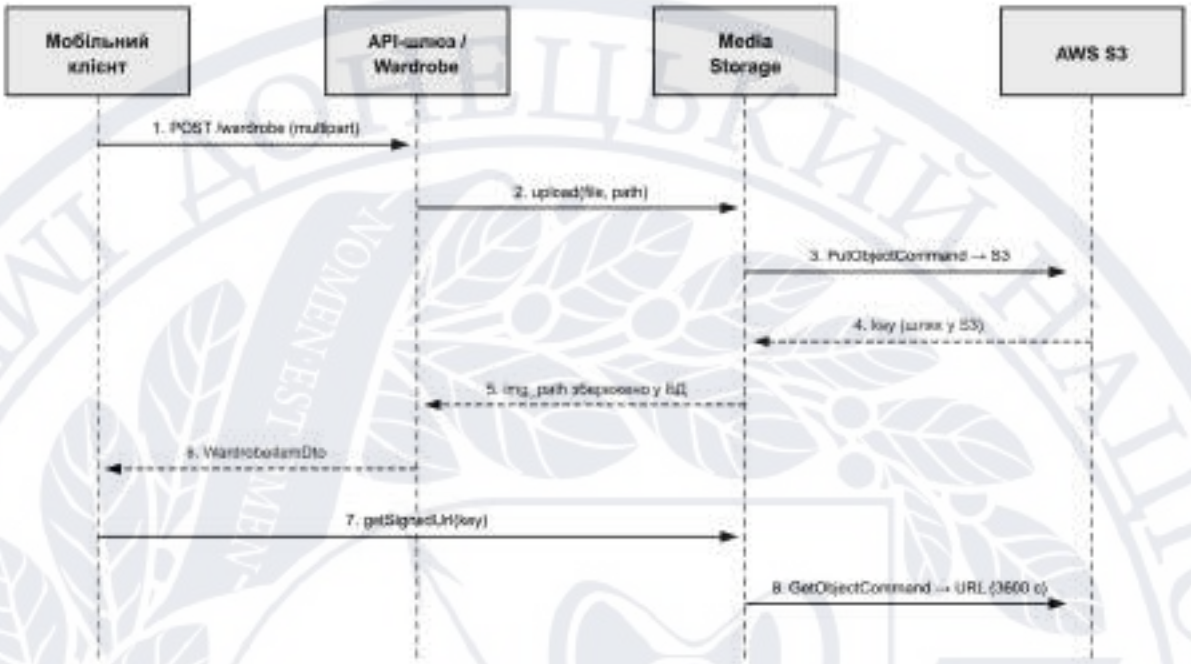

Рисунок 3.3 – Діаграма послідовності завантаження зображення через Pre-signed URL

Використання концепції Pre-signed URL дозволяє повністю виключити необхідність передачі або збереження приватних AWS-ключів доступу на стороні клієнтського застосунку, а автоматичне анулювання посилання після закінчення встановленого таймауту суворо відповідає парадигмі безпеки та принципу мінімальних привілеїв. Клієнтська сторона отримує генероване підписане посилання через API-шлюз, після чого здійснює пряме завантаження бінарного файлу безпосередньо на сервери S3 за протоколом HTTP PUT, що мінімізує обчислювальне навантаження на власну серверну інфраструктуру застосунку та оптимізує мережевий трафік. Аналогічний децентралізований механізм застосовується і для читання медіаресурсів: при формуванні відповіді на запит користувача щодо вмісту гардероба, сформований пул відносних шляхів передається до MediaStorageService, який батчевим методом повертає масив тимчасових безпечних URL-адрес для кожної позиції одягу.

Інтелектуальний сегмент системи представлений мікросервісом ai-assistant, який виступає центральним вузлом обробки природномовних запитів користувача та формування персоналізованих рекомендацій. Архітектурно даний мікросервіс декомпоновано на три взаємопов’язані функціональні компоненти: сервіс акумуляції та структурування даних ContextBuilderService, клієнтський модуль взаємодії з моделлю штучного інтелекту GeminiClientService, а також сервіс менеджменту та збереження діалогових сесій ConversationService.

Для забезпечення мінімального часу відгуку системи, у межах модуля ContextBuilderService було спроєктовано та впроваджено оптимізаційну стратегію паралельного збору даних *«Pre-fetched Context»*. Замість послідовного виконання запитів, що призводило б до кумулятивного зростання затримок, сервіс ініціює одночасне асинхронне отримання п’яти незалежних інформаційних блоків через системний метод Promise.all(). До складу сформованого контекстного пулу входять:

- предмети-контекст, безпосередньо вказані користувачем у поточному запиті;
- еталонні цифрові зображення відповідних речей;
- актуальний список активних предметів одягу, що відповідають поточному сезону;
- динамічний прогноз погоди для поточного географічного розташування користувача;
- журнал нещодавно сформованих образів та комбінацій з бази даних.

На рисунку 3.4. наведено графічну схему процесу паралельного агрегування та побудови контексту, що ілюструє конвеєр збору, структурування та передачі даних, необхідних для формування релевантного системного контексту ШІ-асистента гардероба. В Додаток А наведений програмний код цього сервісу. Процес побудови контексту є критично важливим для забезпечення точних та персоналізованих рекомендацій користувачеві.

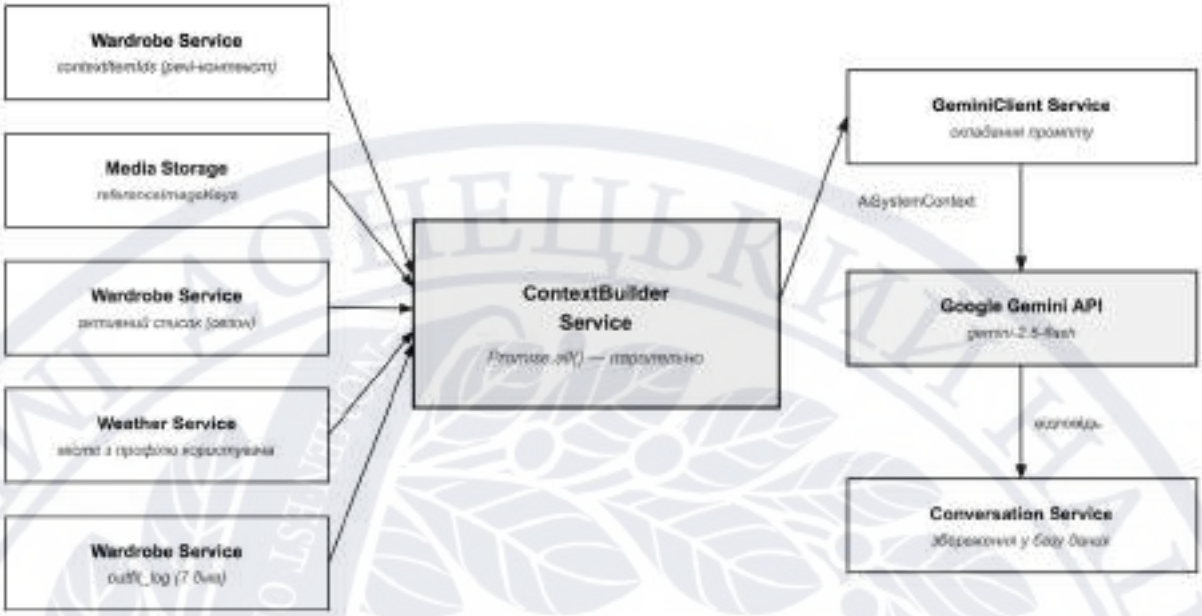

Рисунок 3.4 – Схема паралельної побудови контексту для Google Gemini

З метою дотримання обмежень на розмір вхідного вікна контексту (*context window*) та оптимізації обчислювальних витрат при взаємодії з хмарним API, обробка зібраних даних регламентується спеціалізованим методом capByFavouritesFirst(). Цей метод виконує умовне обмеження результуючого списку активних речей до верхньої межі у 50 одиниць, застосовуючи при цьому алгоритм пріоритизації на основі вподобань користувача. У разі перевищення встановленого ліміту, масив елементів гардероба піддається сортуванню, внаслідок якого об’єкти з булевою ознакою favourite = true зміщуються на початок структури, що гарантує їх першочергове потрапляння до фінального контексту, який передається нейромережі.

Наступний етап обробки покладається на модуль GeminiClientService, завданням якого є консолідація підготовлених текстових та медіаблоків (метеорологічних даних, активного гардероба, історії образів та речей-контексту) у структурований системний промпт (*prompt*). Сформований пакет директив та даних транслюється до віддаленого інструментарію Google Gemini API для обробки базовою моделлю gemini-2.5-flash. Для забезпечення коректної інтерпретації моделлю числових показників довкілля, серіалізація погодних даних здійснюється за допомогою методу serializeWeather(). Цей алгоритм

трансформує структурований об’єкт класу WeatherContext у зв’язний текстовий опис природною мовою, що деталізує поточні температурні показники, загальні метеоумови, рівень вологості та швидкість вітру. Такий підхід дозволяє великій мовній моделі ефективно інтерпретувати складні метеорологічні метрики й адаптувати рекомендації одягу під фактичні погодні умови без необхідності ускладнення системного промпту або додаткового тонкого налаштування (*fine-tuning*) нейромережі.

Шлюз прикладного програмного інтерфейсу (wardrobe-api-gateway) виступає уніфікованою та єдиною точкою входу для всіх вхідних HTTP-запитів із боку клієнтських додатків, забезпечуючи інкапсуляцію внутрішньої мікросервісної архітектури системи. На рівні шлюзу реалізовано централізований механізм контролю доступу, фільтрації трафіку та автентифікації за допомогою спеціалізованого класу захисту AuthGuard. Процедура обробки передбачає обов’язкову верифікацію сесійного JWT-токена (JSON Web Token), інкапсульованого в HTTP-заголовок Authorization (Bearer scheme) кожного захищеного маршруту. У разі успішної дешифрації та валідації токена, AuthGuard автоматично вилучає корисне навантаження (payload) з ідентифікатором користувача (userId) та впроваджує його в об`єкт запиту (Request), що дозволяє подальшим сервісам безперешкодно ідентифікувати автора запиту без повторного звернення до бази даних. При виявленні недійсного або протермінованого токена шлюз миттєво перериває життєвий цикл запиту та повертає клієнту стандартизовану помилку із кодом стану HTTP 401.

Для відкритих кінцевих точок, що забезпечують базові процеси реєстрації, авторизації користувачів або перегляду публічного контенту, застосовується кастомний метадекоратор @Public(). Цей декоратор маркує відповідні контролери або окремі методи спеціальними метаданими, які зчитуються рефлектором (Reflector) у процесі виконання запиту, тим самим динамічно виключаючи їх із глобального циклу перевірки токена AuthGuard.

Окремо, з метою захисту обчислювальних ресурсів системи від зловживань, алгоритмічного перевантаження та потенційних DDoS-атак, на рівні

шлюзу впроваджено шар обмеження інтенсивності трафіку (Rate Limiting). Зокрема, критично важливі маршрути, що відповідають за генерацію, модифікацію та створення нових об’єктів гардероба (що передбачає складну логіку обробки зображень та взаємодію з хмарним сховищем), захищені обмежувачем частоти запитів ThrottlerGuard [30]. Даний компонент аналізує унікальний IP-адрес клієнта та лімітує кількість допустимих транзакцій у заданому часовому інтервалі (наприклад, не більше 10 запитів на хвилину для мутаційних операцій). У разі перевищення встановленого ліміту, шлюз блокує подальше проходження запиту до бізнес-логіки системи та повертає відповідь HTTP 429 (Too Many Requests). Схематичне представлення процесів маршрутизації, фільтрації та автентифікації вхідних запитів на рівні API-шлюзу наведено на рисунку 3.5.

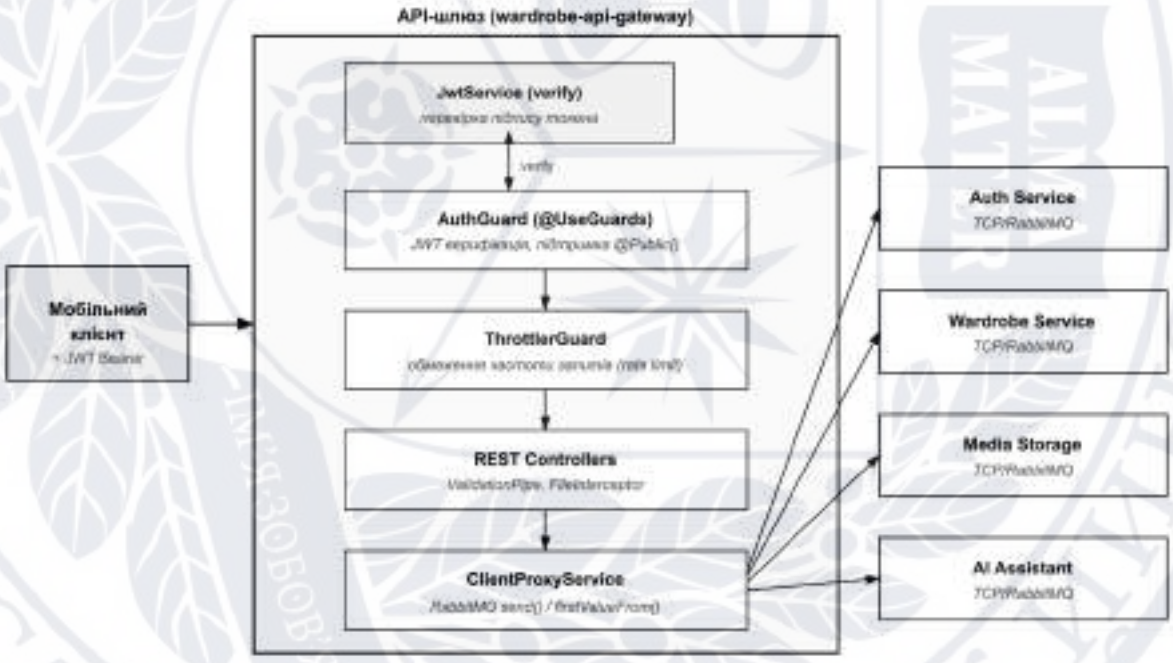

Рисунок 3.5 – Схема маршрутизації та автентифікації запитів в API-шлюзі

Суворий контроль цілісності та валідація вхідних даних забезпечується інтеграцією глобального конвеєра ValidationPipe у поєднанні з можливостями декларативної бібліотеки class-validator [31]. Кожен об’єкт передачі даних (DTO-клас) містить чіткий опис дозволених типів і меж значень за допомогою відповідних декораторів атрибутів:

- декоратор @IsEnum() регламентує фіксований перелік допустимих категорій одягу та сезонності;
- декоратор @IsOptional() маркує необов'язкові для заповнення атрибути профілю або предмета;
- декоратори @IsString() та @MaxLength() накладають типи й обмеження на максимальну довжину текстових рядків.

У разі виявлення невідповідності вхідного пакета даних встановленим правилам, конвеєр ValidationPipeперериває подальшу обробку запиту й повертає клієнту стандартну помилку HTTP 400 (Bad Request) із деталізованим масивом зауважень щодо кожного невалідного поля.

Для автоматизації рутинних операцій та підвищення рівня безпеки на рівні шлюзу розроблено глобальний перехоплювач InjectUserInterceptor. Цей компонент автоматично зчитує ідентифікатор успішно автентифікованого користувача з декодованого токена та інжектує його в тіло повідомлення безпосередньо перед ретрансляцією запиту до цільового мікросервісу. Описане архітектурне рішення повністю усуває потребу в ручній передачі параметра accountId у кожному окремому контролері, а також виступає гарантом того, що внутрішні сервіси оперують виключно верифікованими даними, які неможливо скомпрометувати чи підмінити на етапі формування HTTP-запиту клієнтом. Безпосередня взаємодія шлюзу з ізольованими мікросервісами реалізована на базі вбудованого класу ClientProxyService через метод clientProxy.send(), асинхронний контекст якого обгортається оператором RxJS firstValueFrom() для коректного перетворення потоку подій у Promise-об'єкт.

Загальна програмна реалізація серверної частини системи гарантує надійну ізоляцію користувацьких даних на рівні абсолютно всіх запитів до бази даних завдяки обов'язковій примусовій фільтрації за атрибутом accountId. Горизонтальна масштабованість та відмовостійкість архітектури досягається за рахунок повної автономності та незалежного розгортання кожного мікросервісу у вигляді окремих контейнерів Docker.

Керування конфігураційними параметрами та секретами середовища покладено на вбудований модуль ConfigModule, який реалізує сувору валідацію системних змінних на етапі ініціалізації застосунку. Якщо під час старту платформи виявляється відсутність будь-якої обов'язкової конфігураційної константи, як-от DATABASE_URL, JWT_SECRET або RABBITMQ_URL, процес ініціалізації негайно переривається, а сервіс завершує свою роботу з виведенням відповідного системного повідомлення про помилку. Цей підхід повністю відповідає інженерному принципу *«fail fast»* (швидка відмова) та надійно запобігає розгортанню й функціонуванню некоректно налаштованих екземплярів програмного забезпечення у продакшн-середовищі.

3.2 Розробка користувацького інтерфейсу застосунку

Клієнтський сегмент мобільного застосунку розроблено на базі кросплатформеного фреймворку React Native з використанням екосистеми Expo SDK 51 [32]. Навігаційну логіку та детермінацію маршрутів реалізовано за допомогою файлової системи маршрутизації Expo Router (версії 3) [33]. Фундаментальною інженерною особливістю клієнтського коду є побудова всіх мережевих та асинхронних операцій на базі реактивних потоків програмної бібліотеки RxJS (Observable) [34]. Це дозволило впровадити уніфіковану, декларативну модель обробки подій та керування асинхронними даними по всьому контуру застосунку. Архітектура клієнта має чітко виражену тришарову структуру:

- сервісний шар (services/) – відповідає за низькорівневу взаємодію з зовнішніми інтерфейсами та API;
- шар управління станом (context/) – забезпечує збереження, синхронізацію та дистрибуцію бізнес-даних між компонентами;
- презентаційний шар (app/, components/) – відповідає за безпосереднє рендеринг графічного інтерфейсу користувача та обробку UI-подій.

Загальну архітектурну модель та взаємодію між шарами клієнтської частини системи зображено на рисунку 3.6.

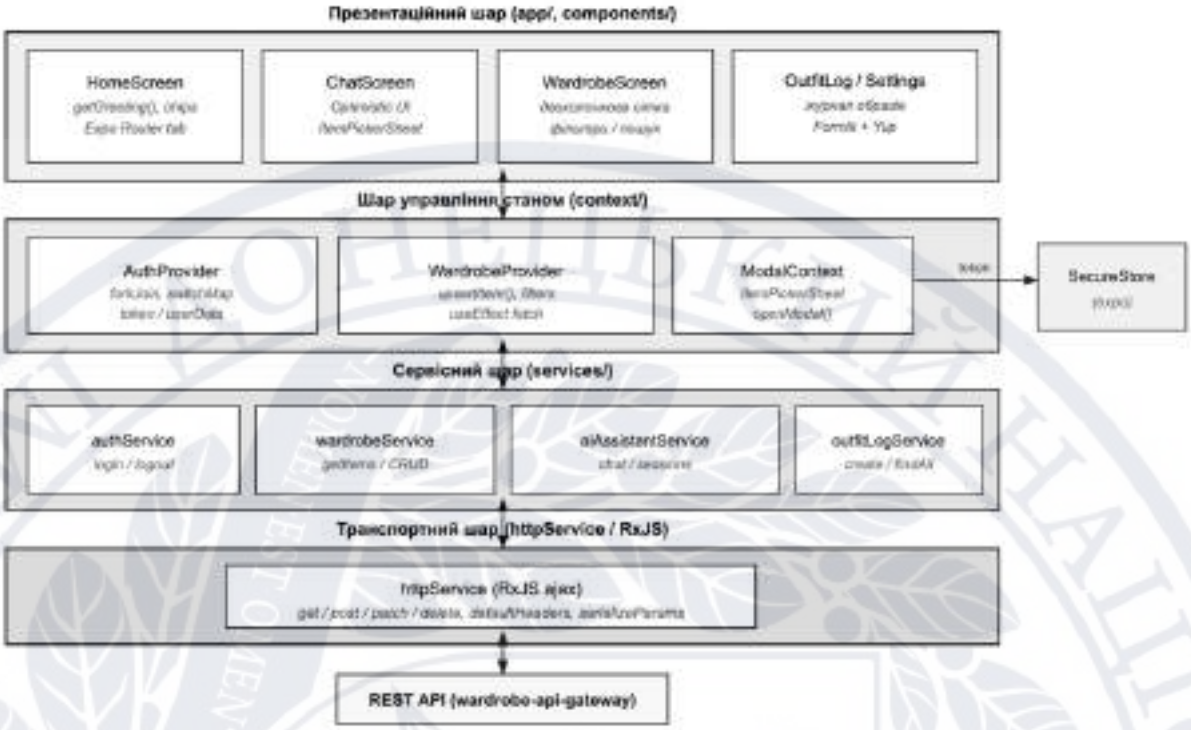

Рисунок 3.6 – Архітектура клієнтської частини додатку

Центральним елементом сервісного шару є модуль httpService, реалізований як об’єкт-синглтон на базі спеціалізованого модуля RxJS ajax. Він надає стандартизовані методи get(), post(), patch() та delete(), кожен з яких повертає об’єкт типу Observable<T>. У контур обробки потоків інтегровано глобальний перехоплювач помилок через оператор catchError, що гарантує централізовану обробку мережевих збоїв та аномалій бекенда. Безпека сесії забезпечується через глобальний словник defaultHeaders, який дозволяє динамічно інжектувати авторизаційний Bearer-токен до складу всіх наступних HTTP-запитів за допомогою методу addDefaultHeader(). Додатково впроваджено утилітарну функцію serializeParams(), яка виконує превентивну фільтрацію вхідних об’єктів, видаляючи поля зі значеннями null та undefined, що запобігає трансляції некоректних або порожніх параметрів запиту на сервер. Базова адреса хостингу бекенд-платформи конфігурується через системну змінну оточення EXPO_PUBLIC_API_BASE_URL, що надає можливість безшовно перемикатися між локальним сервером розробника та віддаленим виробничим середовищем без модифікації вихідного коду застосунку.

Глобальний стан мобільного застосунку адмініструється двома незалежними провайдерами архітектурного патерна React Context:

1. AuthProvider – координує життєвий цикл сесії користувача. Під час первинного монтування компонента сервіс ініціює асинхронне зчитування криптографічно захищеного токена зі сховища expo-secure-store за допомогою оператора комбінації forkJoin(), що дозволяє автоматично відновити авторизовану сесію без необхідності повторного введення облікових даних. Процедури автентифікації (login()) та реєстрації (register()) побудовані як послідовні Observable-ланцюжки з використанням оператора проекції switchMap(). Після отримання успішної відповіді від API-шлюзу, токен та персональні дані паралельно записуються у захищене сховище SecureStore, після чого синхронно оновлюється локальний React-стан застосунку [35].
2. WardrobeProvider – здійснює менеджмент масиву предметів одягу та супутніх параметрів фільтрації. Системний хук useEffect здійснює безперервний моніторинг об’єкта filters і автоматично ініціює метод fetchItems() при фіксації будь-яких змін у структурі фільтрів. Для покращення користувацького досвіду (UX) метод модифікації даних upsertItem() реалізує паттерн *оптимістичного оновлення* (Optimistic Updates): локальний UI-стан змінюється миттєво після дії користувача, не очікуючи фінального підтвердження транзакції від сервера [36].

Обидва контекст-провайдери задекларовані у кореневому файлі app/_layout.tsx і повністю огортають весь навігаційний граф застосунку. Важливою деталлю управління пам’яттю є те, що підписка на Observable-потік у методі fetchItems() примусово скасовується в cleanup-функції useEffect. Це нівелює ризик оновлення стану вже розмонтованого компонента інтерфейсу та запобігає виникненню критичних витоків оперативної пам’яті (Memory Leaks).

Структура навігації мобільного клієнта базується на використанні компонента нижнього меню табів (Bottom Tab Navigator), який містить чотири функціональні вкладки: Головна (HomeScreen), Гардероб (WardrobeScreen),

Журнал образів (OutfitLogScreen) та Налаштування (SettingsScreen). Фреймворк Expo Router автоматично будує інтерактивний навігаційний граф, спираючись на фізичну файлову структуру та ієрархію директорій всередині папки app/. З метою забезпечення інформаційної безпеки, всі приватні екрани системи сгруповані всередині ізольованої директорії захищених маршрутів app/(app)/ та загорнуті в спеціалізований Layout-компонент, що здійснює динамічний аудит наявності валідного сесійного токена. У разі виявлення неавторизованого доступу, компонент виконує примусове перенаправлення (Redirect) користувача на екран авторизації, розташований у відкритій директорії app/(auth)/. Такий підхід до архітектурного розподілу маршрутів повністю виключає можливість несанкціонованого перегляду захищених інтерфейсів застосунку, навіть у випадку спроби прямого переходу за глибокими посиланнями (*Deep Linking*).

Головний екран мобільного застосунку (HomeScreen) є композиційним центром інтерфейсу користувача, який інтегрує блоки динамічного вмісту та інструменти швидкої навігації. Персоналізація взаємодії починається з інтерактивного блоку привітання: вбудована функція getGreeting() здійснює аналіз системного часу пристрою та повертає відповідне текстове привітання залежно від часу доби: «Доброго ранку», «Доброго дня» або «Доброго вечора». Нижче розташовується динамічна стрічка персоналізованих рекомендацій, сформованих штучним інтелектом на основі поточного контексту, а також функціональний блок швидкого діалогу, що містить набір інтерактивних ярликів-запитів. При активації певного ярлика спрацьовує метод handleChipSelect(), який одночасно інжектує відповідний текст у поле введення та ініціює відправку запиту до ШІ-асистента без необхідності здійснення додаткових маніпуляцій з боку користувача. Цей механізм дозволяє значно скоротити час взаємодії з інтерфейсом, мінімізуючи когнітивне навантаження та забезпечуючи безшовний досвід керування додатком. Графічний інтерфейс та компонування елементів головного екрану застосунка можна побачити на рисунку 3.7.

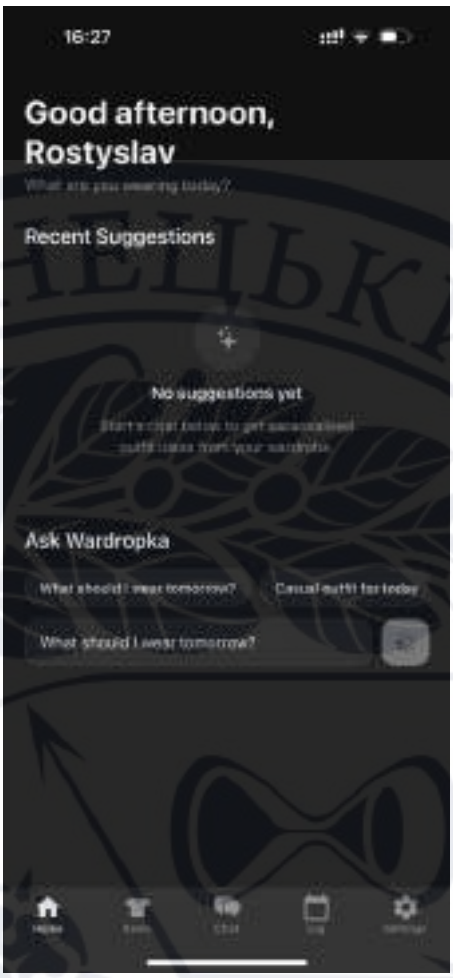

Рисунок 3.7 – Головний екран застосунка

Екран цифрового гардероба реалізує візуалізацію масиву елементів одягу у вигляді адаптивної двоколонкової сітки карткових компонентів, що забезпечує високу щільність інформації та ергономічність сприйняття на екранах мобільних пристроїв із різною роздільною здатністю. Для оптимізації продуктивності рендерингу великих масивів даних замість стандартних контейнерів використано компонент FlatList, який забезпечує динамічне перевикористання елементів інтерфейсу та ліниве завантаження карток у міру скролінгу.

Графічні мініатюри предметів завантажуються децентралізовано безпосередньо з хмарного сховища за допомогою раніше згенерованих тимчасових безпечних посилань (Pre-signed URL). Процес відображення зображень супроводжується механізмом прогресивного завантаження та агресивного кешування на стороні клієнта, що нівелює повторні мережеві запити при повторному відкритті екрана та суттєво зменшує споживання мобільного трафіку.

Панель гнучкої фільтрації інтегрована у верхній частині інтерфейсу і представлена горизонтальним компонентом ScrollView, що містить інтерактивні чіпи-перемикачі для швидкого сортування та категоризації об`єктів. Візуальний стан активних фільтрів динамічно підсвічується за допомогою контрастних елементів теми, забезпечуючи чіткий зворотний зв`язок.

Усі маніпуляції з критеріями пошуку обробляються за допомогою реактивних потоків стану, завдяки чому фільтрація речей у сітці відбувається в режимі реального часу (миттєве відсіювання невідповідних карток). При цьому сумарний лічильник задіяних критеріїв динамічно виводиться у вигляді числового індикатора (активного бейджа з контрастним фоном) безпосередньо на іконці розширеної фільтрації в головному заголовку екрана, що дозволяє користувачеві контролювати глибину вибірки даних навіть при згорнутій панелі. Графічне представлення інтерфейсу гардероба з розгорнутою панеллю активних фільтрів можна переглянути на рисунку 3.8.

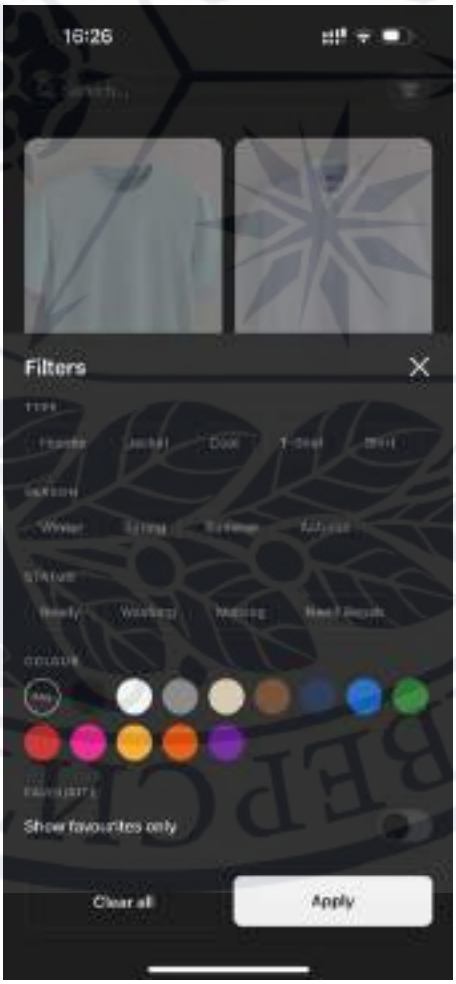

Рисунок 3.8 – Екран гардеробу з панеллю фільтрів

Кожна інформаційна картка предмета одягу оснащена функціональними елементами швидкої дії, зокрема кнопкою додавання до категорії обраного (іконка у вигляді серця) та клавішею безпосереднього видалення (іконка кошика). Довготривале натискання (довгий тап) на площину картки ініціює відкриття модального контекстного меню з можливістю переходу до спеціалізованого екрана редагування характеристик об’єкта. Окрім цього, для оптимізації ергономіки інтерфейсу реалізовано жест свайпу картки ліворуч, що викликає приховану кнопку видалення, активація якої супроводжується модальним діалоговим вікном підтвердження дії (Alert), що запобігає випадковій деструкції даних користувачем. Функціонал пошуку елементів у реальному часі побудовано за принципом локальної фільтрації вже кешованого на клієнті масиву даних, що дозволяє мінімізувати кількість повторних мережевих запитів до API-шлюзу та знижує навантаження на базу даних.

Екран інтерактивного діалогу з ШІ (ChatScreen) підтримує як відновлення існуючих сесій спілкування з бази даних, так і динамічне створення нових діалогових гілок, що забезпечує гнучкість та безперервність користувацького досвіду. Процес обробки та відправки текстових повідомлень базується на передовому архітектурному патерні Optimistic UI (оптимістичний інтерфейс), який дозволяє візуально нівелювати затримки мережевої взаємодії. При ініціалізації відправки нове повідомлення користувача миттєво рендериться у списку FlatList із присвоєнням тимчасового локального ідентифікатора та статусу надсилання, не очікуючи на фінальну відповідь від віддаленого сервера або API-шлюзу. Це створює ілюзію миттєвої роботи застосунку та значно підвищує суб`єктивну швидкість інтерфейсу.

У разі виникнення мережевого збою, таймауту або критичної помилки на стороні бекенда, механізм синхронізації відкочує оптимістично доданий елемент, а користувачеві виводиться відповідне push-сповіщення або модальне вікно з пропозицією повторити спробу відправки.

Архітектурну схему обробки повідомлення та керування станом у межах екрана ChatScreen за принципом Optimistic UI зображено на рисунку 3.9.

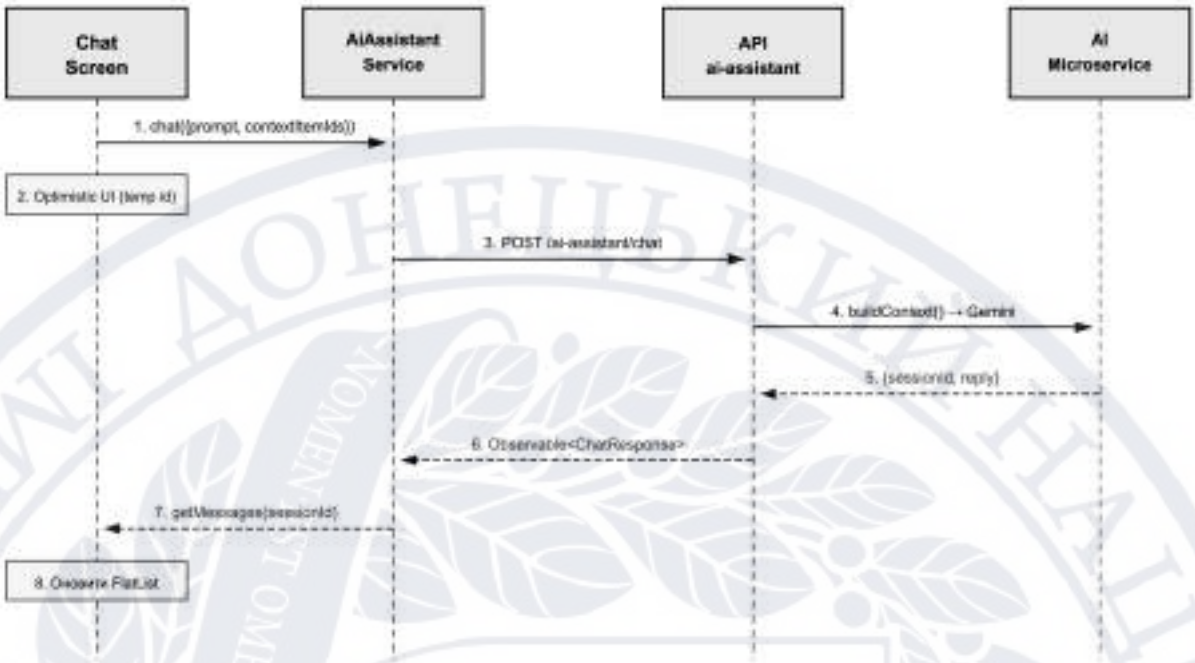

Рисунок 3.9 – Діаграма послідовності обробки повідомлення у ChatScreen

Згенеровані відповіді ШІ-асистента виводяться на екран мобільного додатка з повноцінною підтримкою Markdown-форматування, що дозволяє відображати складні аналітичні дані у звичному для користувача вигляді. Трансформація сирого текстового рядка, що надходить від API моделі штучного інтелекту і містить жирне або курсивне накреслення, марковані й нумеровані списки, емодзі, а також ієрархічні заголовки, у валідні та оптимізовані компоненти інтерфейсу React Native здійснюється за допомогою інтеграції спеціалізованої бібліотеки react-native-markdown-display. Це архітектурне рішення дозволяє уникнути використання важковагових та небезпечних контейнерів типу WebView, забезпечуючи рендеринг елементів тексту у вигляді нативних UI-компонентів (<Text>, <View>), що значно підвищує рівень сприйняття, читабельності та візуальної структурованості аналітичних рекомендацій. Окрім цього, бібліотека інтегрована із загальною системою стилізації додатка, що гарантує автоматичну адаптацію кольорів шрифтів та фону під активну тему інтерфейсу.

Для деталізації та хронологічного структурування діалогу між окремими блоками повідомлень користувача та відповідями ШІ-асистента динамічно

відображається мітка часу їх створення (timestamp). Кожне повідомлення обертається в анімований контейнер із плавним розширенням (fade-in ефект) при появі у стрічці чату. Візуальний інтерфейс вікна чату в режимі активного діалогу з ШІ-асистентом наведено на рисунку 3.10.

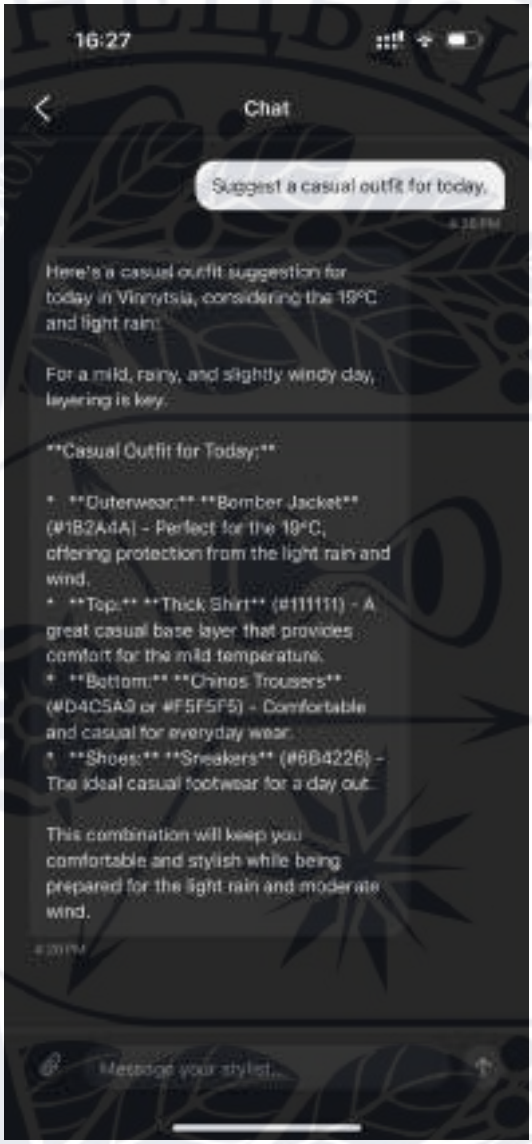

Рисунок 3.10 – Екран чату з ШІ-асистентом

Унікальною архітектурною особливістю екрана ChatScreen є можливість інтеграції конкретних предметів гардероба безпосередньо у контекст обговорення. За допомогою виклику модальної нижньої панелі ItemPickerSheet, управління якою делеговано глобальному контексту ModalContext, користувач має можливість маркувати речі, ідентифікатори яких згодом передаються у спеціалізованому полі contextItemIds у загальному пакеті даних до мікросервісу штучного інтелекту.

Екрани додавання та модифікації характеристик предметів одягу відображено в Додаток Б. Їх спроєктовано з використанням бібліотеки Formik для централізованого управління станом полів форми, а також інструментарію Yup для опису схем валідації. Такі атрибути, як тип, сезон та поточний статус, детерміновані як обов'язкові для заповнення. З метою оптимізації використання оперативної пам'яті пристрою та запобігання повторній ініціалізації об'єкта валідації при кожному рендеринг-циклі компонента, декларативна схема validationSchema винесена за межі функціонального класу екрана і визначена статично [37, 38].

Процес імпорту графічних матеріалів реалізовано через системний модуль ImagePicker. Перед безпосередньою трансляцією файлу на сервер виконується його попередня оптимізація: стиснення компресії до рівня якості 0.8 та примусове обмеження максимальних лінійних розмірів до 1200×1200 пікселів, після чого об'єкт упаковується у формат FormData. Проведення попереднього стиснення дозволяє суттєво скоротити обсяги мережевого трафіку та зменшити час очікування завантаження файлу без видимого погіршення візуальної якості мініатюр в інтерфейсі мобільного застосунку.

Екран історичного журналу образів (OutfitLogScreen) відображає структуровану в хронологічному порядку стрічку збережених комбінацій одягу із можливістю розгортання детальної інформації за допомогою дотику до елемента. Кожен інформаційний запис містить візуальні мініатюри речей, які увійшли до складу образу, календарну дату формування та розгорнутий текстовий опис стилістичних нюансів. Фільтрація записів за часовими інтервалами реалізована через інтеграцію нативного компонента DateTimePicker. Екран персональних налаштувань профілю (SettingsScreen) надає користувачеві інструменти для модифікації особистого імені та міста проживання. Слід зазначити, що зміна географічного розташування (міста) миттєво враховується при наступній ініціалізації діалогу з ШІ, оскільки сервіс WeatherService зчитує актуальне місто безпосередньо на етапі динамічної збірки контексту для великої мовної моделі.

Візуальна стилізація інтерфейсу та всіх його атомарних компонентів виконана за допомогою стандартного інструментарію StyleSheet.create() із впровадженням єдиної дизайн-системи (теми), заснованої на константних палітрах кольорів та розмірних сітках. Високий рівень адаптивності інтерфейсу до різних форм-факторів мобільних пристроїв досягається за допомогою використання Dimensions API та оперування відсотковими відступами. Це забезпечує коректне та пропорційне відображення графічних елементів на смартфонах із діагоналлю екрана від 4 до 6.9 дюймів. Інтеграція провайдера безпечних зон SafeAreaProvider гарантує стабільне позиціонування елементів інтерфейсу, виключаючи перекриття важливих навігаційних вузлів технічними вирізами (зокрема «чубчиками») або закругленнями дисплеїв сучасних мобільних пристроїв

3.3 Верифікація та тестування програмного комплексу

Комплексна верифікація та оцінка коректності функціонування розробленого програмного забезпечення здійснювалася на двох ієрархічних рівнях: модульному (юніт-тестування) для ізольованої перевірки серверних бізнес-компонентів та функціональному (інтеграційному) – безпосередньо на фізичному мобільному пристрої.

З метою забезпечення повної ізоляції об’єктів тестування від зовнішніх залежностей, у межах модульного тестування було активно задіяно спеціалізовані сурогатні об’єкти середовища Jest (jest.Mock) [39, 40]. Інтеграція мок-об’єктів дозволила замінити реальні мережеві виклики, звернення до сторонніх API та транзакції до реляційної бази даних на контрольовані програмні заглушки (*stubs*). Описаний підхід дозволив верифікувати прикладну логіку сервісів незалежно від стану зовнішньої інфраструктури, гарантуючи детермінованість, повторюваність та високу швидкість виконання тестових сценаріїв.

Архітектура кожного тестового набору даних та інструкцій структурована за загальноприйнятим інженерним патерном AAA (Arrange-Act-Assert):

- Arrange (Організація): у межах системного блоку beforeEach перед початком кожного окремого інваріанта ініціюється чистий екземпляр сервісу та свіжі мок-об’єкти, що повністю нівелює ризик взаємного деструктивного впливу або накопичення стану між різними тестовими сценаріями;
- Act (Дія): безпосередній виклик цільового методу підсистеми з передачею згенерованих вхідних параметрів;
- Assert (Перевірка): аудит отриманих результатів. Окрім прямої перевірки вихідних даних, суттєва увага приділяється верифікації поведінки об’єктів. Це дозволяє підтвердити не лише коректність фінального стану, а й відсутність побічних ефектів, таких як ініціалізація небажаних мережевих транзакцій за відсутності авторизаційних ключів.

Основними об’єктами модульного тестування на серверній стороні було обрано ключові компоненти підсистеми штучного інтелекту: сервіс агрегації даних ContextBuilderService (розроблено 5 тестових сценаріїв) та модуль взаємодії з метеорологічними сервісами WeatherService (розроблено 4 тестові сценарії).

Для сервісного компонента ContextBuilderService системній верифікації підлягали критичні стани, що охоплюють як позитивні сценарії виконання, так і граничні та аварійні випадки. Першочергово перевірялася коректність агрегації та трансформації даних за умови повної валідності всіх вхідних параметрів, що надходять від суміжних мікросервісів, та їх відповідності очікуваній JSON-схемі для формування інтегрованого контексту. Окрему увагу було приділено працездатності алгоритму обмеження об’єму даних, де тестувалася бізнес-логіка зрізання масиву об`єктів до ліміту у 50 активних предметів гардероба із забезпеченням детермінованого пріоритету для улюблених речей із прапорцем isFavorite. Також верифікувався механізм ізоляції та блокування викликів, зокрема – запобігання надсиланню зайвих мережевих запитів до зовнішнього сервісу WeatherService у разі відсутності або некоректного формату атрибута міста (city) у профілі користувача. У межах забезпечення стійкості системи до

відмов (Fault Tolerance) було протестовано обробку виняткових ситуацій при поверненні значення null або помилки з боку WeatherService, де сервіс мав коректно перехоплювати виняток, логувати інцидент та продовжувати побудову решти контексту. Наостанок аналізувалася поведінка алгоритму при абсолютно порожньому електронному гардеробі користувача для запобігання генерації невалідних порожніх промптів. Комплексна перевірка зазначених станів реалізовувалася за допомогою модульного тестування (Unit Testing) у середовищі Jest із застосуванням паттерну AAA (Arrange-Act-Assert), де всі зовнішні залежності були заміщені відповідними стабами та імітаційними об`єктами (mocks) для повної ізоляції обчислювальної логіки.

Зведені результати виконання розроблених модульних тестів для сервісу побудови контексту та часові витрати в ms на кожен з них відображено на рисунку 3.11.

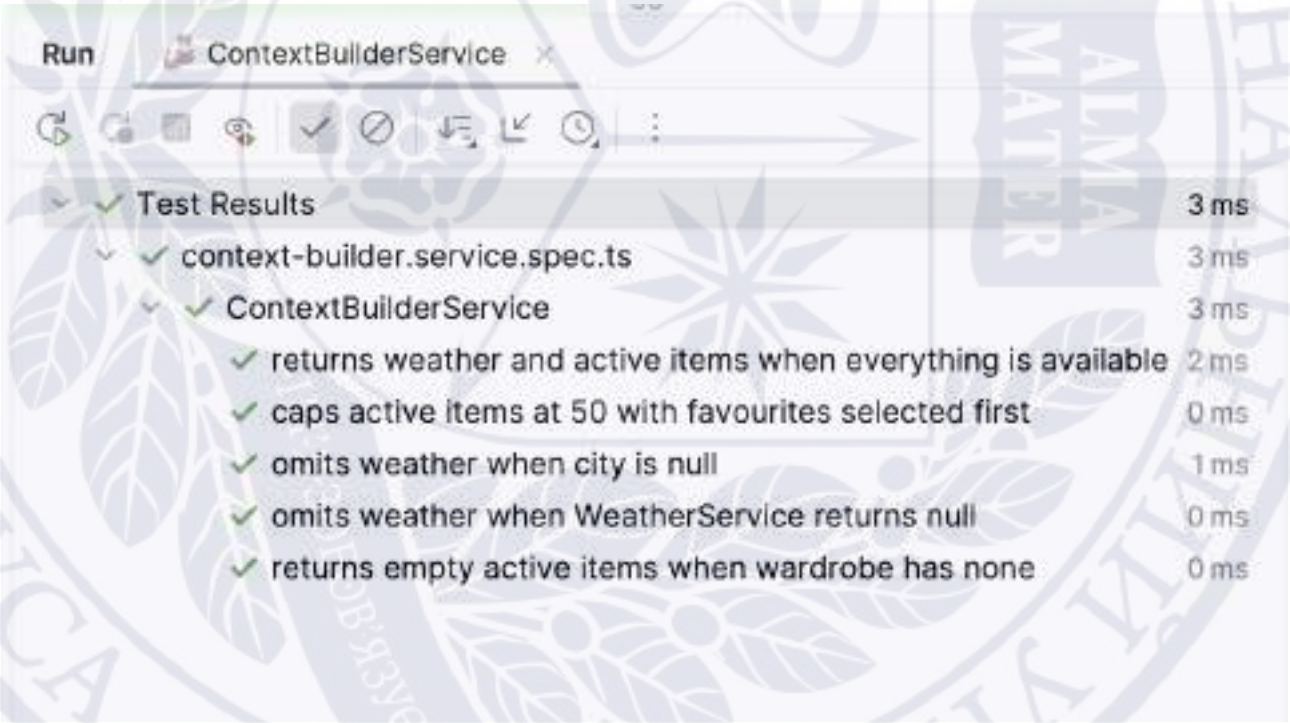

Рисунок 3.11 – Результати виконання модульних тестів у середовищі Jest

Тестовий сценарій «Відсутній API-ключ» має підвищену пріоритетність у контексті забезпечення інформаційної безпеки системи та мінімізації фінансових витрат на інтеграцію сторонніх сервісів. Під час його виконання створюється екземпляр класу WeatherService із залученням конфігураційного сервісу ConfigService, який примусово повертає стан undefined замість валідного токена

доступу. Тест вважається успішно пройденим лише за умови, що внутрішній клієнт не виконав жодного виклику за протоколом HTTP (expect(httpGet).not.toHaveBeenCalled()), повернувши безпечне значення null. Впровадження цієї перевірки повністю нівелює ризик розгортання некоректно налаштованих інстансів мікросервісу та запобігає трансляції невалідних запитів до сторонніх хмарних провайдерів.

Функціональне верифікаційне тестування розробленого програмного комплексу проводилося безпосередньо на фізичному мобільному пристрої під управлінням операційної системи iOS (версія 18) в умовах реального розгортання та взаємодії з хмарною інфраструктурою бекенд-платформи. У межах тестування було детально змодельовано та виконано сім базових сценаріїв користувацької взаємодії, зведені результати яких представлено в таблиці 3.1.

Таблиця 3.1 Результати функціонального тестування основних сценаріїв взаємодії з додатком

Функціональний сценарій	**Очікуваний результат верифікації**	**Поточний статус**
Реєстрація та автентифікація	Успішна генерація та збереження JWT, автоматичний перехід на головний екран	OK
Додавання речі з медіафайлом	Завантаження зображення до хмари AWS S3, відображення прев'ю в інтерфейсі	OK
Фільтрація елементів за сезоном	Динамічне відображення об'єктів, що відповідають обраному сезону	OK
Рекомендація ШІ на основі погоди	Генерація відповіді асистента з урахуванням актуального метеорологічного контексту	OK
Діалог із прикріпленим предметом	Формування порад ШІ з урахуванням характеристик конкретної обраної речі	OK

Фіксація образу в історичному журналі	Коректне відображення сформованого запису у вкладці журналу	ОК
Термінація та відновлення сесії	Повторне безшовне завантаження даних гардероба після перезапуску сесії	ОК

Додатково було здійснено всебічну перевірку поведінки мобільного клієнта у граничних (стресових) сценаріях функціонування. При спробі відображення критичного обсягу даних (понад 200 одночасних предметів одягу в галереї) графічний інтерфейс зберігає високу плавність та стабільні показники кадрів за секунду (FPS), що досягається завдяки інтеграції нативного компонента FlatList із механізмом віртуалізації рендерингу. Ініціалізація запиту до мікросервісу штучного інтелекту за умов відсутності підключення до мережі Інтернет обробляється за допомогою перехоплювачів помилок і супроводжується виведенням інформативного модального повідомлення для користувача, повністю виключаючи зависання інтерфейсу чи появу нескінченного індикатора завантаження (*infinite spinner*). Спроба несанкціонованого доступу до захищених екранів системи шляхом прямого переходу за внутрішніми посиланнями автоматично блокується архітектурним шаром AuthGuard із миттєвим перенаправленням на екран автентифікації, що унеможливлює компрометацію чи витік конфіденційних даних.

Аудит безпеки та розмежування прав доступу до даних виконувався за трьома критичними векторами:

1. Спроба міжкористувацької підміни ідентифікаторів (ID Spoofing): штучне моделювання та відправка HTTP-запиту з маніпуляцією параметром id предмета, що належить іншому обліковому запису, повертає стандартну помилку HTTP 404 (Not Found). Це підтверджує надійність ізоляції даних на рівні реляційної СКБД завдяки обов’язковій фільтрації за атрибутом accountId.
2. Аудит життєвого циклу токена: використання простроченого сесійного токена JWT коректно перехоплюється API-шлюзом, який повертає статус

HTTP 401 (Unauthorized), що змушує клієнтський захисний шар виконати примусову деавторизацію користувача.

3. Запити без авторизаційних заголовків: надсилання прямих запитів до захищених ендпоінтів без інжекції токена безпеки миттєво блокується на рівні шлюзу з поверненням помилки HTTP 401, не розкриваючи при цьому внутрішню топологію та структуру мікросервісів системи.

Для кількісної оцінки ефективності розробленого рішення було проведено серію хронометричних вимірювань швидкодії основних операцій. Зафіксовано, що середній час повного завантаження галереї цифрового гардероба (екземпляр вибірки – 20 предметів) разом із процедурою батчевого отримання Pre-signed URL-адрес становить менше 800 мс в умовах мобільного з'єднання за стандартом LTE. Часовий інтервал формування першої відповіді ШІ-асистента варіюється в межах від 1,5 до 4 секунд, що безпосередньо корелює з поточним рівнем обчислювального навантаження на віддалені сервери Google Gemini API та обсягом сформованого текстово-графічного контексту. Процедура реєстрації нового користувача, яка включає ресурсомісткий процес криптографічного соленого хешування пароля алгоритмом bcrypt із коефіцієнтом складності 10, займає менше 300 мс. Повні показники швидкої дії та умов проведення експериментів зведено в таблиці 3.2.

Таблиця 3.2 Показники продуктивності основних операцій системи

Назва технологічної операції	**Середній час обробки (мс)**	**Специфікація та умови вимірювання**
Завантаження галереї (20 предметів)	< 800	Мережа LTE, пакетна генерація Pre-signed URL
Генерація першої відповіді ШІ	1500 – 4000	Динамічне вікно контексту, Gemini API
Реєстрація / Автентифікація профілю	< 300	Хешування Bcrypt (salt rounds = 10)

За результатами дослідної експлуатації розробленого програмного забезпечення було виявлено та зафіксовано одне архітектурне обмеження поточної версії системи, пов`язане з життєвим циклом безпечних адрес. При первинному відкритті інтерфейсу цифрового гардероба після тривалого простою мобільного застосунку в фоновому режимі, раніше згенеровані адреси Pre-signed URL для графічних мініатюр одягу можуть виявитися неактивними через вичерпання встановленого терміну їх дії, який на рівні конфігурації AWS S3 становить 3600 секунд (1 година). У такому випадку мобільний клієнт при спробі децентралізованого завантаження медіафайлів отримує помилку доступу HTTP 403 (Forbidden) від об’єктного сховища S3, що змушує логіку фронтенду перехоплювати цей виняток та ініціювати повторний запит до API-шлюзу для примусової актуалізації та перевипуску посилань.

Даний процес реактивного оновлення створює додаткову затримку відображення графічного інтерфейсу (UI latency) в межах 200–400 мс, що негативно впливає на показники користувацького досвіду (UX), зокрема на швидкість першого відмальовування контенту (First Contentful Paint). Як рекомендацію щодо подальшого технологічного вдосконалення системи та повного нівелювання даного ефекту, запропоновано впровадження фонового механізму попереджувального оновлення (proactive URL refreshing) адресацій на стороні клієнта за допомогою сервіс-воркерів або періодичних фонових завдань (Background Tasks) до моменту фактичного закінчення терміну їхньої валідності. Альтернативним шляхом вирішення цієї проблеми у наступних ітераціях розробки є інтеграція мережі доставки контенту Amazon CloudFront із використанням підписаних файлів cookies (Signed Cookies), що дозволить кешувати статичний контент на межових серверах (Edge Locations) та підтримувати високий рівень безпеки без необхідності постійної регенерації індивідуальних посилань для кожного предмета гардероба.

ВИСНОВКИ

У кваліфікаційній (бакалаврській) роботі вирішено науково-практичне завдання проєктування та програмної реалізації мобільного застосунку «Асистент для гардеробу» на основі клієнт-серверної архітектури. За результатами проведеного дослідження сформульовано такі висновки:

1. Систематизовано теоретико-методологічні засади цифровізації персонального гардеробу в контексті постіндустріального суспільства. Встановлено, що проблема «гардеробного паралічу» є наслідком інформаційної ентропії, спричиненої надлишковим накопиченням несистематизованих речей. Обґрунтовано доцільність застосування концепції «Цифрового двійника» для трансформації пасивного каталогу одягу в активну систему управління персональними ресурсами. Виявлено економічну ефективність такого підходу через впровадження показника вартості за одне використання (Cost per Wear), що стимулює раціональне споживання та підтримує принципи сталого розвитку.
2. Проведено порівняльний аналіз існуючих комерційних рішень у сегменті персональних органайзерів гардероба (Stylebook, Acloset, Cladwell). Виявлено критичні технологічні прогалини: переважання монолітних та клієнт-серверних архітектур без підтримки асинхронної обробки, відсутність механізмів відмовостійкості, обмеженість алгоритмів рекомендацій, що ігнорують динамічний контекст (метеорологічні умови, соціальні події, статус чистоти речі), а також надмірно високий поріг входу через ручне оцифрування гардероба.
3. Обґрунтовано вибір технологічного стеку та методологію проєктування системи. Встановлено оптимальність поєднання Node.js/NestJS для серверної частини, React Native (Expo) для мобільного клієнта, PostgreSQL як реляційної СУБД та RabbitMQ як брокера повідомлень. Доведено, що використання мікросервісного архітектурного підходу в поєднанні з подієво-орієнтованою взаємодією забезпечує горизонтальну

масштабованість, ізоляцію збоїв та незалежне розгортання компонентів, що є критично важливим для стабільної роботи мобільної екосистеми.

4. Спроєктовано архітектурну модель системи на основі мікросервісного підходу та подієво-орієнтованої взаємодії. Розроблено п`ять спеціалізованих мікросервісів: API Gateway, Auth, Wardrobe, AI Assistant та Media Storage, кожен з чіткою функціональною зоною відповідальності. Встановлено механізми асинхронної взаємодії між компонентами за допомогою брокера повідомлень RabbitMQ з використанням патернів «Publish-Subscribe» та «Request-Response», що забезпечує відмовостійкість системи навіть під час виконання ресурсомістких обчислювальних завдань.
5. Розроблено реляційну структуру бази даних на основі PostgreSQL, яка забезпечує зберігання та ефективну обробку даних чотирьох ключових сутностей: user_account, wardrobe_item, outfit_log та assistant_session. Надано класифікацію атрибутів сутності wardrobe_item для інтеграції з моделлю штучного інтелекту в межах стратегії «Pre-fetched Context». Уточнено підхід до зберігання медіаданих через гібридну модель, що поєднує реляційні таблиці PostgreSQL для метаданих та хмарне сховище AWS S3 для фотографій одягу з використанням механізму Pre-signed URL.
6. Розроблено програмний комплекс, що складається з серверної мікросервісної платформи та мобільного клієнта. На серверній стороні реалізовано: двофакторну аутентифікацію на базі JWT та bcrypt, CRUD-операції над цифровим гардеробом з транзакційним видаленням медіафайлів, інтелектуальний мікросервіс ai-assistant з паралельною стратегією збору контексту через Promise.all(), а також API-шлюз з механізмами захисту (AuthGuard, ThrottlerGuard, ValidationPipe). На стороні клієнта впроваджено тришарову архітектуру з використанням RxJS для управління асинхронними потоками, патерн Optimistic UI для підвищення суб`єктивної швидкодії та expo-secure-store для безпечного зберігання токенів.

7. Проведено верифікацію та тестування програмного комплексу на двох рівнях. За результатами модульного тестування у середовищі Jest підтверджено коректність роботи 9 тестових сценаріїв для компонентів ContextBuilderService та WeatherService, включаючи граничні та аварійні випадки. За результатами функціонального тестування на фізичному мобільному пристрої (iOS 18) підтверджено успішне виконання всіх семи базових сценаріїв взаємодії. Встановлено показники продуктивності системи: завантаження галереї з 20 предметів займає менше 800 мс, генерація відповіді ШІ-асистента – від 1 500 до 4 000 мс, реєстрація користувача – менше 300 мс.
8. Виявлено архітектурне обмеження поточної версії системи, пов`язане з достроковим анулюванням Pre-signed URL після тривалого простою застосунку (термін дії – 3 600 с), що спричиняє затримку UI у 200–400 мс під час реактивного оновлення посилань. Підготовлено рекомендації щодо усунення цього обмеження: впровадження фонового механізму попереджувального оновлення адрес на стороні клієнта або інтеграція Amazon CloudFront із підписаними файлами cookies для кешування контенту на межових серверах.

Практична цінність розробленого програмного комплексу підтверджується готовністю системи до дослідної експлуатації та можливістю її подальшого розвитку у напрямку інтеграції з IoT-екосистемами (інтелектуальні дзеркала, системи «розумного дому»), а також розширення функціоналу ШІ-асистента за рахунок впровадження повноцінних моделей комп`ютерного зору для автоматичної класифікації та сегментації зображень одягу.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Кравчук Р.Ю., Гончар В.М. Еволюція об’єктно-орієнтованих баз даних від перших JSON до повноцінних об’єктно-орієнтованих СУБД. URL: https://jktod.donnu.edu.ua/article/view/13110 (дата звернення: 20.04.2026).
2. Кравчук Р.Ю., Ніколюк П.К. Комбінаторика в програмуванні. Основні проблеми та алгоритм їх вирішення. URL: https://jktod.donnu.edu.ua/article/view/13049 (дата звернення: 20.04.2026).
3. Niinimäki K., Peters G., Dahlbo H., et al. *The environmental price of fast fashion*. *Nature Reviews Earth & Environment*. 2020. Vol. 1, No. 4. P. 189–200. URL: https://www.nature.com/articles/s43017-020-0039-9 (дата звернення: 01.02.2025).
4. Ellen MacArthur Foundation. *A new textiles economy: Redesigning fashion’s future*. Ellen MacArthur Foundation, 2017. URL: https://www.ellenmacarthurfoundation.org/a-new-textiles-economy (дата звернення: 10.03.2025).
5. WRAP. *Valuing Our Clothes: the cost of UK fashion*. WRAP Report on Sustainable Clothing. 2017. URL: https://www.wrap.ngo/resources/report/valuing-our-clothes-cost-uk-fashion (дата звернення: 15.03.2025).
6. McKinsey & Company. *The State of Fashion 2026: When the rules change*. McKinsey & Company, 2026. URL: https://www.mckinsey.com/industries/retail/our-insights/state-of-fashion (дата звернення: 20.01.2026).
7. Schwartz B. *The Paradox of Choice: Why More Is Less*. New York: Harper Perennial, 2004. 265 p. ISBN: 978-0-06-000568-6.
8. Liu Z., Luo P., Qiu S., Wang X., Tang X. *DeepFashion: Powering Robust Clothes Recognition and Retrieval with Rich Annotations*. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las

Vegas, 2016. P. 1096–1104. URL: https://openaccess.thecvf.com/content_cvpr_2016/html/Liu_DeepFashion_Powering_Robust_CVPR_2016_paper.html (дата звернення: 01.03.2025).

9. Vohs K. D., Baumeister R. F. (eds.) *Handbook of Self-Regulation: Research, Theory, and Applications*. 2nd ed. New York: Guilford Press, 2011. 703 p. ISBN: 978-1-60918-507-5.
10. Grieves M. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems*. Transdisciplinary Perspectives on Complex Systems. Cham: Springer, 2017. P. 85–113. ISBN: 978-3-319-38754-3.
11. Newman S. *Building Microservices: Designing Fine-Grained Systems*. 2nd ed. Sebastopol: O'Reilly Media, 2021. 616 p. ISBN: 978-1-4920-3402-9.
12. Richardson C. *Microservices Patterns: With Examples in Java*. Shelter Island: Manning Publications, 2018. 520 p. ISBN: 978-1-61729-454-9.
13. Google AI for Developers. Gemini API Reference. URL: https://ai.google.dev/api/generate-content (дата звернення: 05.03.2026).
14. NestJS. Official Documentation. URL: https://docs.nestjs.com (дата звернення: 20.10.2025).
15. OpenJS Foundation. Node.js Documentation. Version 20 LTS. URL: https://nodejs.org/en/docs (дата звернення: 10.11.2025).
16. Microsoft. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/ (дата звернення: 05.12.2025).
17. Broadcom Inc. RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs (дата звернення: 18.11.2025).
18. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: PhD thesis. University of California, Irvine, 2000. 162 p. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 01.09.2025).
19. Meta Platforms. React Native Documentation. URL: https://reactnative.dev/docs/getting-started (дата звернення: 08.10.2025).

20. Amazon Web Services. Amazon S3 User Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/ (дата звернення: 20.01.2026).
21. OpenWeatherMap. Weather API Documentation. URL: https://openweathermap.org/api (дата звернення: 15.03.2026).
22. PostgreSQL Global Development Group. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 30.10.2025).
23. TypeORM. Official Documentation. URL: https://typeorm.io (дата звернення: 25.10.2025).
24. Refactoring.Guru. Strategy Design Pattern. URL: https://refactoring.guru/design-patterns/strategy (дата звернення: 18.09.2025).
25. Amazon Web Services. Sharing objects with presigned URLs. Amazon S3 User Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html (дата звернення: 20.01.2026).
26. Nrwl Engineering. Nx Workspace Documentation. URL: https://nx.dev/docs (дата звернення: 12.11.2025).
27. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). IETF. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 14.02.2026).
28. npm. bcrypt package. URL: https://www.npmjs.com/package/bcrypt (дата звернення: 14.02.2026).
29. Amazon Web Services. AWS SDK for JavaScript v3 Developer Guide. URL: https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/ (дата звернення: 22.01.2026).
30. NestJS. Security: Rate Limiting. NestJS Documentation. URL: https://docs.nestjs.com/security/rate-limiting (дата звернення: 15.12.2025).

31. typestack. class-validator: Decorator-based property validation for classes. GitHub. URL: https://github.com/typestack/class-validator (дата звернення: 10.12.2025).
32. Expo. Expo SDK Documentation. URL: https://docs.expo.dev/versions/latest/ (дата звернення: 12.10.2025).
33. Expo. Expo Router Documentation. URL: https://docs.expo.dev/router/introduction/ (дата звернення: 14.10.2025).
34. ReactiveX. RxJS: Reactive Extensions For JavaScript. URL: https://rxjs.dev/guide/overview (дата звернення: 20.10.2025).
35. Expo. expo-secure-store API Reference. URL: https://docs.expo.dev/versions/latest/sdk/securestore/ (дата звернення: 25.10.2025).
36. Apollo GraphQL. Optimistic UI. Apollo Client Documentation. URL: https://www.apollographql.com/docs/react/performance/optimistic-ui/ (дата звернення: 05.11.2025).
37. Formik. Formik Documentation. URL: https://formik.org/docs/overview (дата звернення: 30.10.2025).
38. jquense. Yup: Dead simple Object schema validation. GitHub. URL: https://github.com/jquense/yup (дата звернення: 30.10.2025).
39. OpenJS Foundation. Jest: Delightful JavaScript Testing. URL: https://jestjs.io/docs/getting-started (дата звернення: 10.11.2025).
40. NestJs Testing. URL: https://docs.nestjs.com/fundamentals/testing (дата звернення: 10.11.2025)
41. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River: Prentice Hall, 2008. 431 p. ISBN: 978-0-13-235088-4.
42. Hastings R., Meyer E. No Rules Rules: Netflix and the Culture of Reinvention. New York: Penguin Press, 2020. 320 p. ISBN: 978-1-9848-7116-8.

ДОДАТКИ

ДОДАТОК А

Програмний код сервісу для формування контексту

```
import { Inject, Injectable, Logger } from '@nestjs/common';
import { ClientProxy } from '@nestjs/microservices';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { firstValueFrom } from 'rxjs';

import {
  OutfitLogDto,
  WardrobeItemDto,
  WardrobeItemPreviewDto,
} from '@app/wardrobe/dto';
import {
  OUTFIT_LOG_REQUESTS,
  WARDROBE_REQUESTS,
} from '@app/wardrobe/constants';
import { ItemStatus } from '@app/wardrobe/enums';
import { MEDIA_STORAGE_REQUESTS } from '@app/media-
storage/constants/requests';
import { ItemPath } from '@app/media-storage/models';
import {
  MEDIA_STORAGE_SERVICE,
  WARDROBE_SERVICE,
} from '@app/wardrobe-api-gateway/constants';
import { UserAccountPreview } from '@app/auth/users/types';
import { UserAccountEntity } from
'@app/common/database/entities/auth';

import { WeatherService } from './weather.service';
import { getCurrentSeason } from './current-season.util';
import { WeatherContext } from '../types/weather-context.type';

const MAX_ACTIVE_ITEMS_IN_CONTEXT = 50;

export interface RecentlyWornEntry {
  date: string;
  itemNames: string[];
}

export interface AiSystemContext {
  wardrobeItems: WardrobeItemDto[];
  referenceImageUrls: string[];
  activeWardrobeItems: WardrobeItemPreviewDto[];
  weather: WeatherContext | null;
  recentlyWorn: RecentlyWornEntry[];
}

@Injectable()
export class ContextBuilderService {
  private readonly logger = new
Logger(ContextBuilderService.name);
```

```
  constructor(
    @Inject(WARDROBE_SERVICE) private readonly wardrobeClient:
ClientProxy,
    @Inject(MEDIA_STORAGE_SERVICE) private readonly mediaClient:
ClientProxy,
    @InjectRepository(UserAccountEntity)
    private readonly accountRepository:
Repository<UserAccountEntity>,
    private readonly weatherService: WeatherService,
  ) {}

  async buildContext(
    account: UserAccountPreview,
    options: {
      contextItemIds?: number[];
      referenceImageKeys?: string[];
    },
  ): Promise<AiSystemContext> {
    const [
      wardrobeItems,
      referenceImageUrls,
      activeWardrobeItems,
      weather,
      recentlyWorn,
    ] = await Promise.all([
      this.fetchWardrobeItems(account, options.contextItemIds),
      this.fetchReferenceImageUrls(options.referenceImageKeys),
      this.fetchActiveSeasonalItems(account),
      this.fetchWeatherForAccount(account.id),
      this.fetchRecentlyWorn(account),
    ]);

    return {
      wardrobeItems,
      referenceImageUrls,
      activeWardrobeItems,
      weather,
      recentlyWorn,
    };
  }

  async fetchWardrobeItems(account: UserAccountPreview, ids?:
number[]) {
    if (!ids?.length) {
      return [];
    }

    return firstValueFrom(
      this.wardrobeClient.send(WARDROBE_REQUESTS.findManyByIds, {
        data: ids,
        user: account,
      }),
```

```
      ) as Promise<WardrobeItemDto[]>;
    }

    private async fetchActiveSeasonalItems(
      account: UserAccountPreview,
    ): Promise<WardrobeItemPreviewDto[]> {
      const season = getCurrentSeason();

      try {
        const items = (await firstValueFrom(
          this.wardrobeClient.send(WARDROBE_REQUESTS.findMany, {
            data: { status: ItemStatus.Active, season },
            user: account,
          }),
        )) as WardrobeItemPreviewDto[];

        return this.capByFavouritesFirst(items ?? []);
      } catch (error) {
        this.logger.warn(
          `Failed to fetch active wardrobe items for account
${account.id}: ${
            error instanceof Error ? error.message : String(error)
          }`,
        );
        return [];
      }
    }

    private capByFavouritesFirst(
      items: WardrobeItemPreviewDto[],
    ): WardrobeItemPreviewDto[] {
      if (items.length <= MAX_ACTIVE_ITEMS_IN_CONTEXT) {
        return items;
      }

      const sorted = [...items].sort((a, b) => {
        if (a.favourite !== b.favourite) {
          return a.favourite ? -1 : 1;
        }
        return (b.id ?? 0) - (a.id ?? 0);
      });

      return sorted.slice(0, MAX_ACTIVE_ITEMS_IN_CONTEXT);
    }

    private async fetchWeatherForAccount(
      accountId: number,
    ): Promise<WeatherContext | null> {
      const account = await this.accountRepository.findOne({
        where: { id: accountId },
        select: ['id', 'city'],
      });
```

```
    if (!account?.city) {
      return null;
    }

    return
firstValueFrom(this.weatherService.getForecast(account.city));
  }

  async fetchRecentlyWorn(
    account: UserAccountPreview,
  ): Promise<RecentlyWornEntry[]> {
    const RECENT_LOG_LIMIT = 7;

    try {
      const logs = (await firstValueFrom(
        this.wardrobeClient.send(OUTFIT_LOG_REQUESTS.findMany, {
          data: { limit: RECENT_LOG_LIMIT },
          user: account,
        }),
      )) as OutfitLogDto[];

      if (!logs?.length) {
        return [];
      }

      const allItemIds = [
        ...new Set(logs.flatMap((log) => log.wardrobeItemIds)),
      ];

      const items = await this.fetchWardrobeItems(account,
allItemIds);
      const nameById = new Map(
        items.map((item) => [item.id, item.name || item.type]),
      );

      return logs.map((log) => ({
        date: new Date(log.date).toISOString().split('T')[0],
        itemNames: log.wardrobeItemIds.map(
          (id) => nameById.get(id) ?? String(id),
        ),
      }));
    } catch (error) {
      this.logger.warn(
        `Failed to fetch recently worn logs for account
${account.id}: ${
          error instanceof Error ? error.message : String(error)
        }`,
      );
      return [];
    }
  }

  private async fetchReferenceImageUrls(keys?: string[]) {
```

```
    if (!keys?.length) {
      return [];
    }

    const itemPaths: ItemPath[] = keys.map((path, index) => ({
      id: index,
      path,
    }));

    const result = (await firstValueFrom(
      this.mediaClient.send(MEDIA_STORAGE_REQUESTS.getUrls, {
        itemPaths,
      }),
    )) as Record<string, string>;

    return keys.map((_, index) => result[index] ??
null).filter(Boolean);
  }
}
```

ДОДАТОК Б

Загальний вигляд користувацького інтерфейсу застосунка

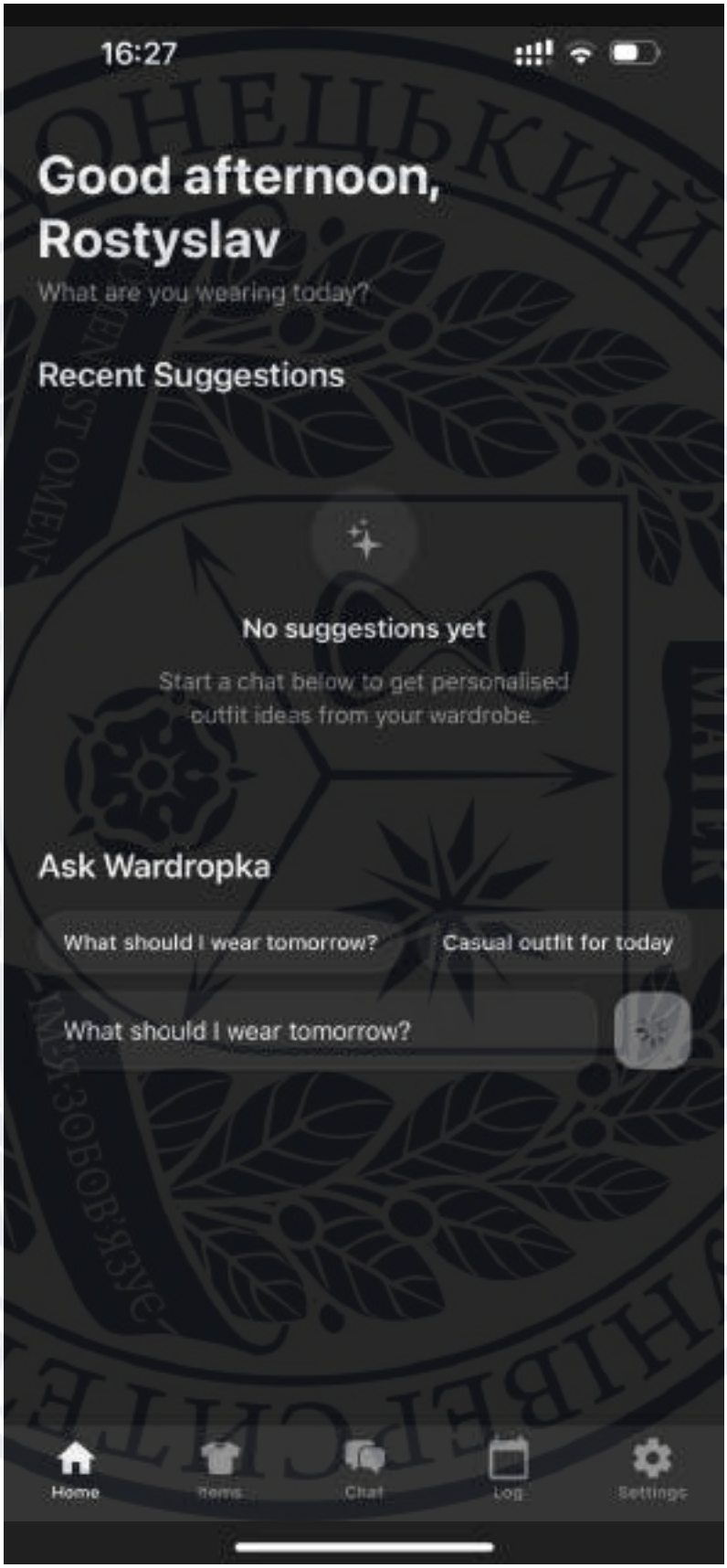

Рисунок Б.1 – Головний екран застосунку

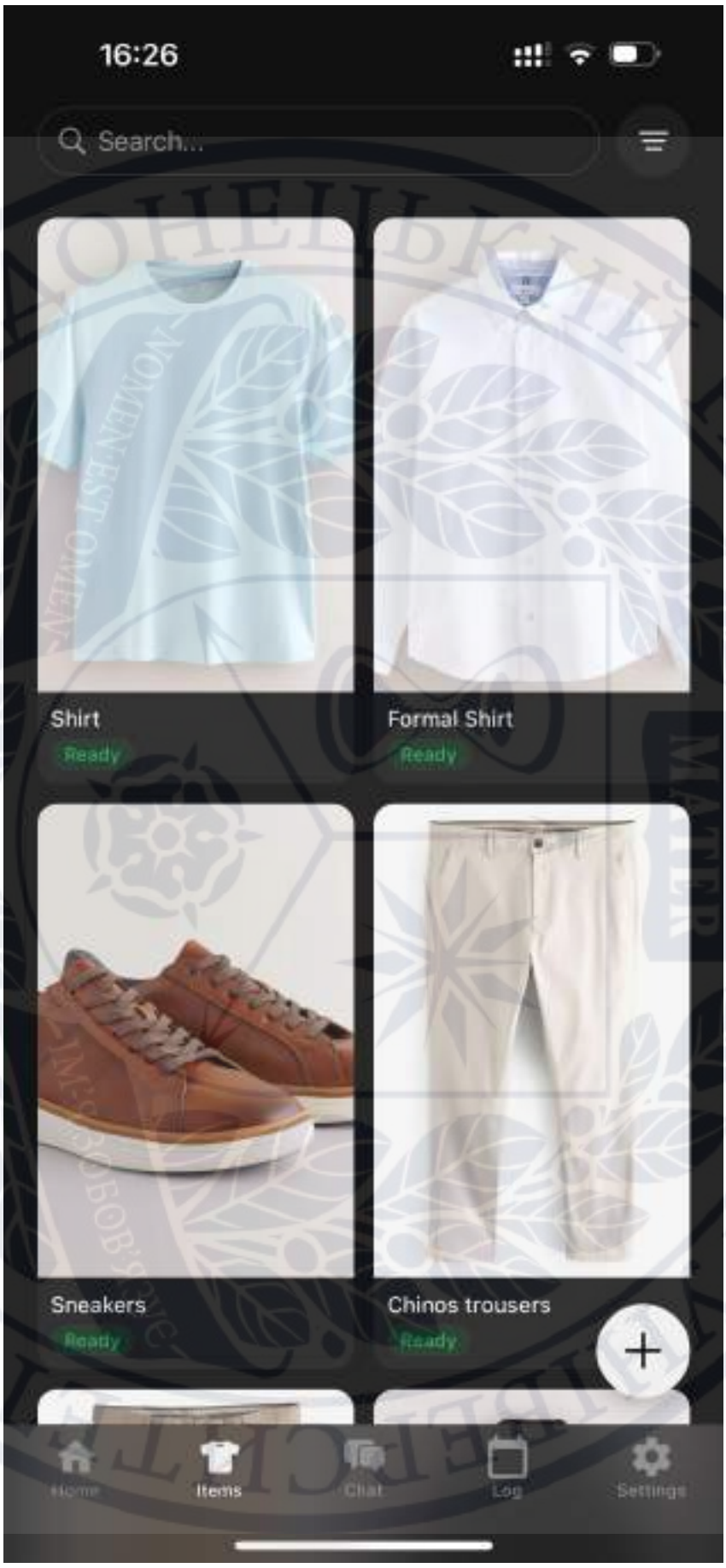

Рисунок Б.2 – Екран доступних речей в гардеробі

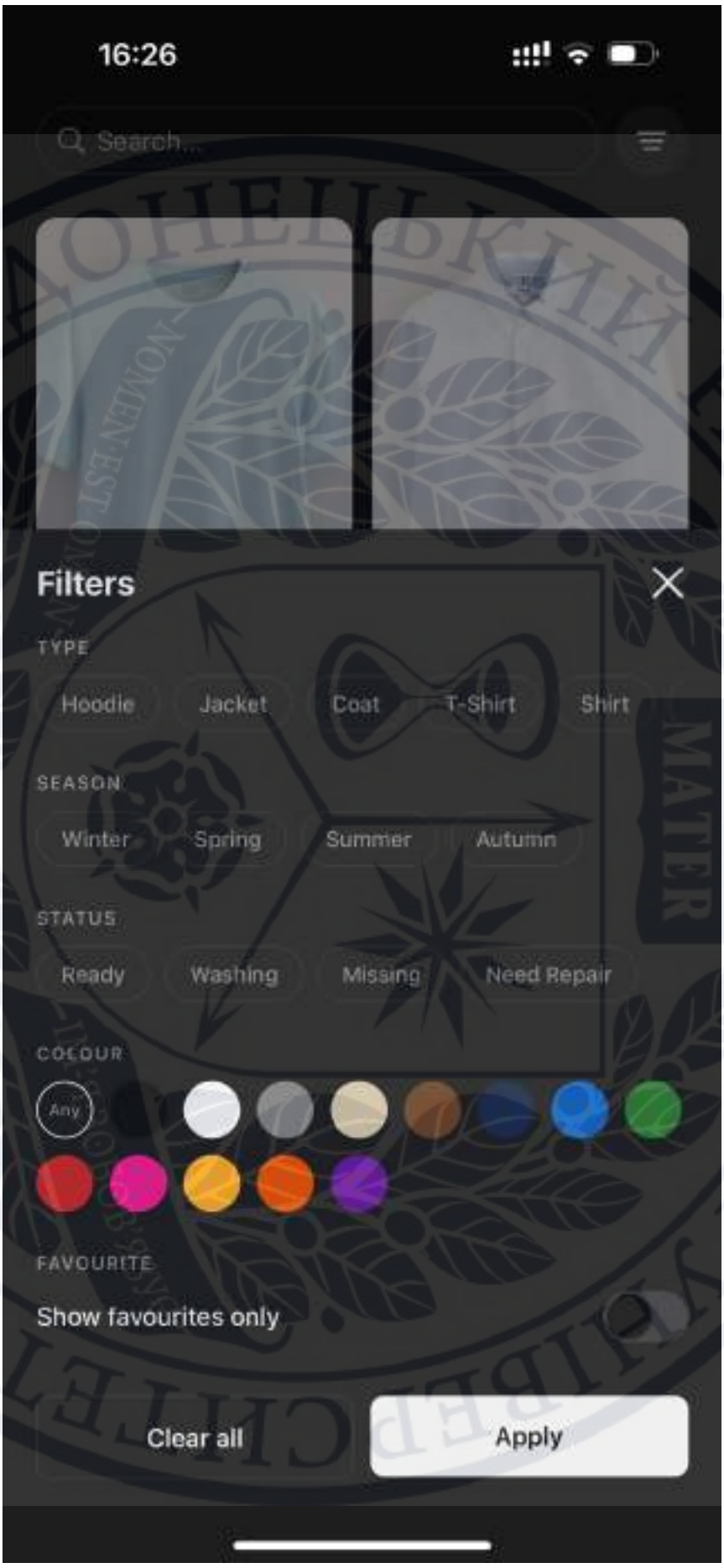

Рисунок Б.3 – Спливаюче вікно з фільтрами

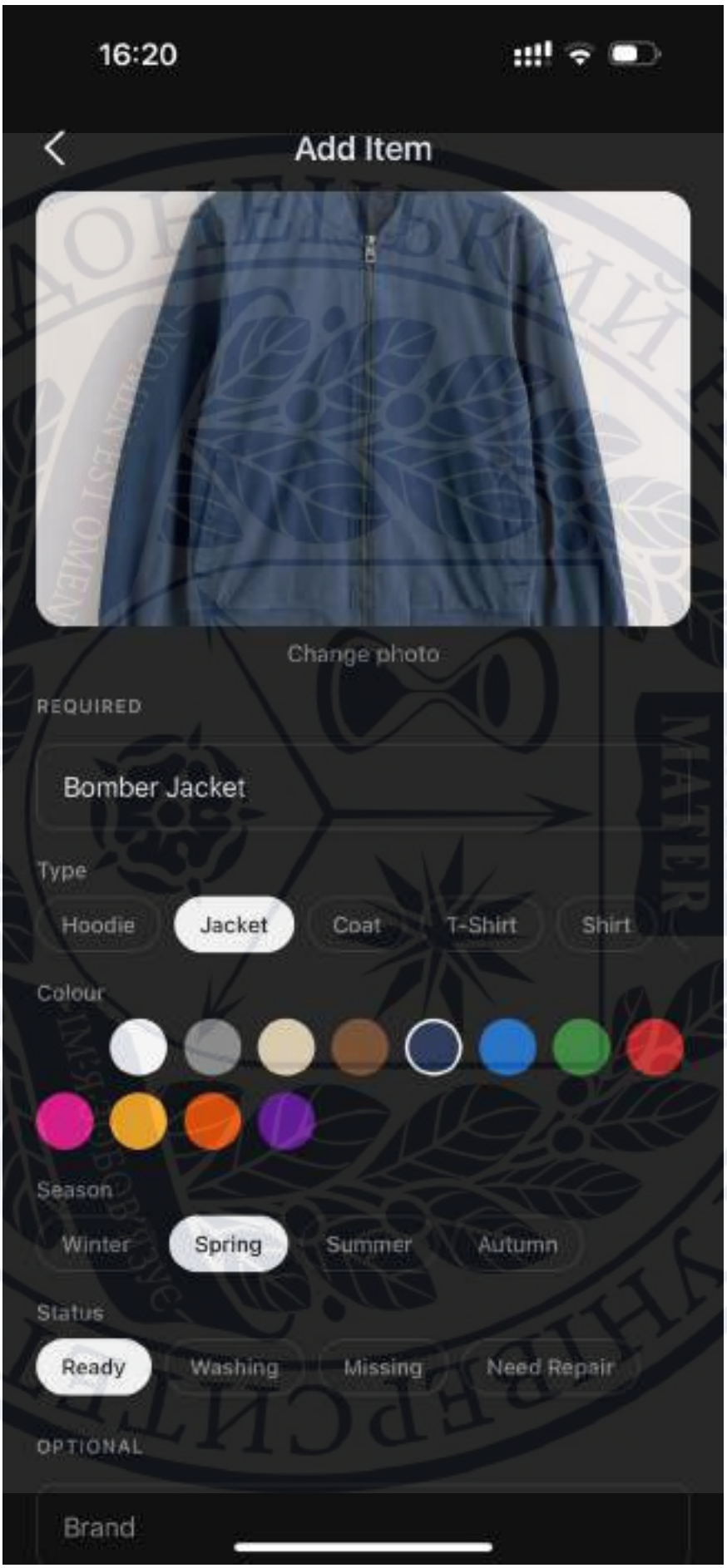

Рисунок Б.4 – Екран додавання/редагування речі

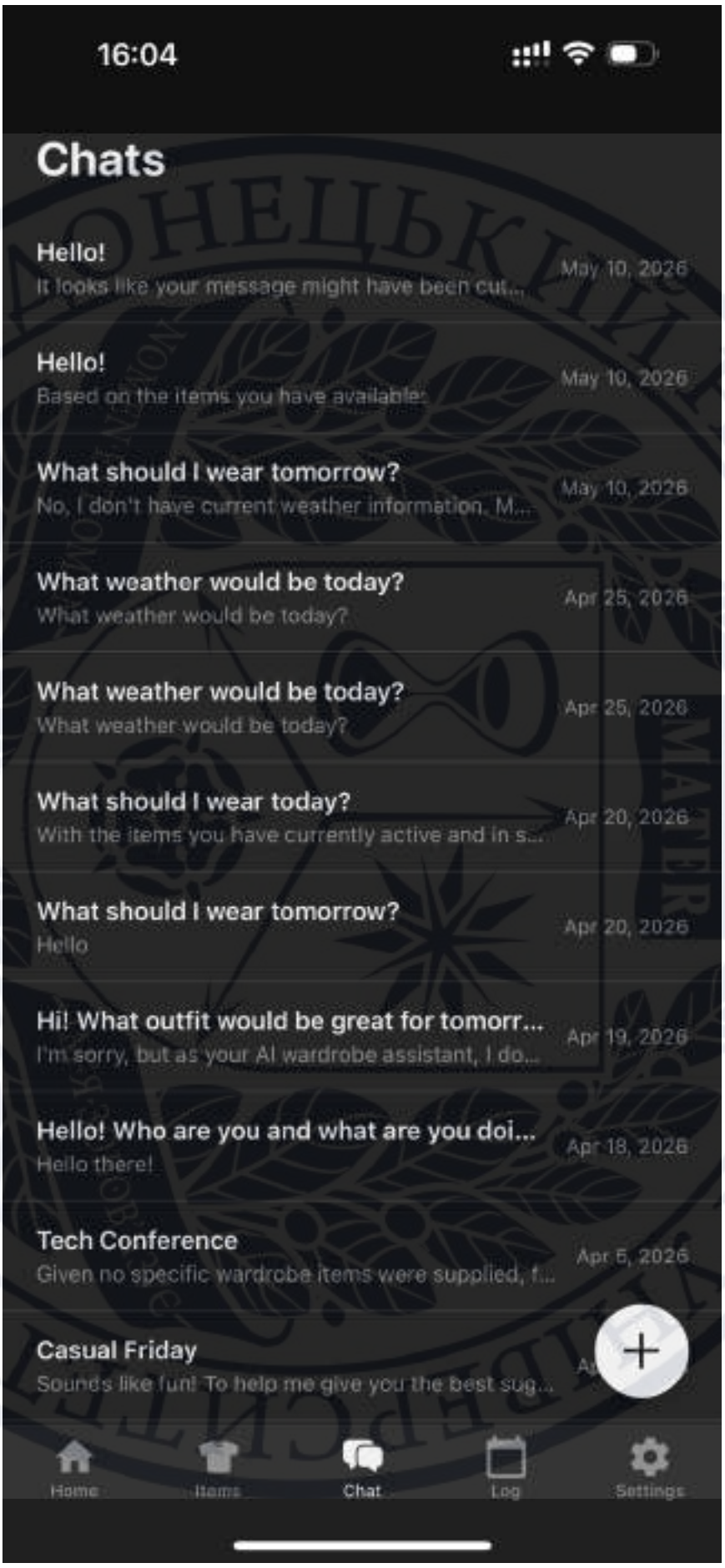

Рисунок Б.5 – Екран історії чатів з ШІ асистентом

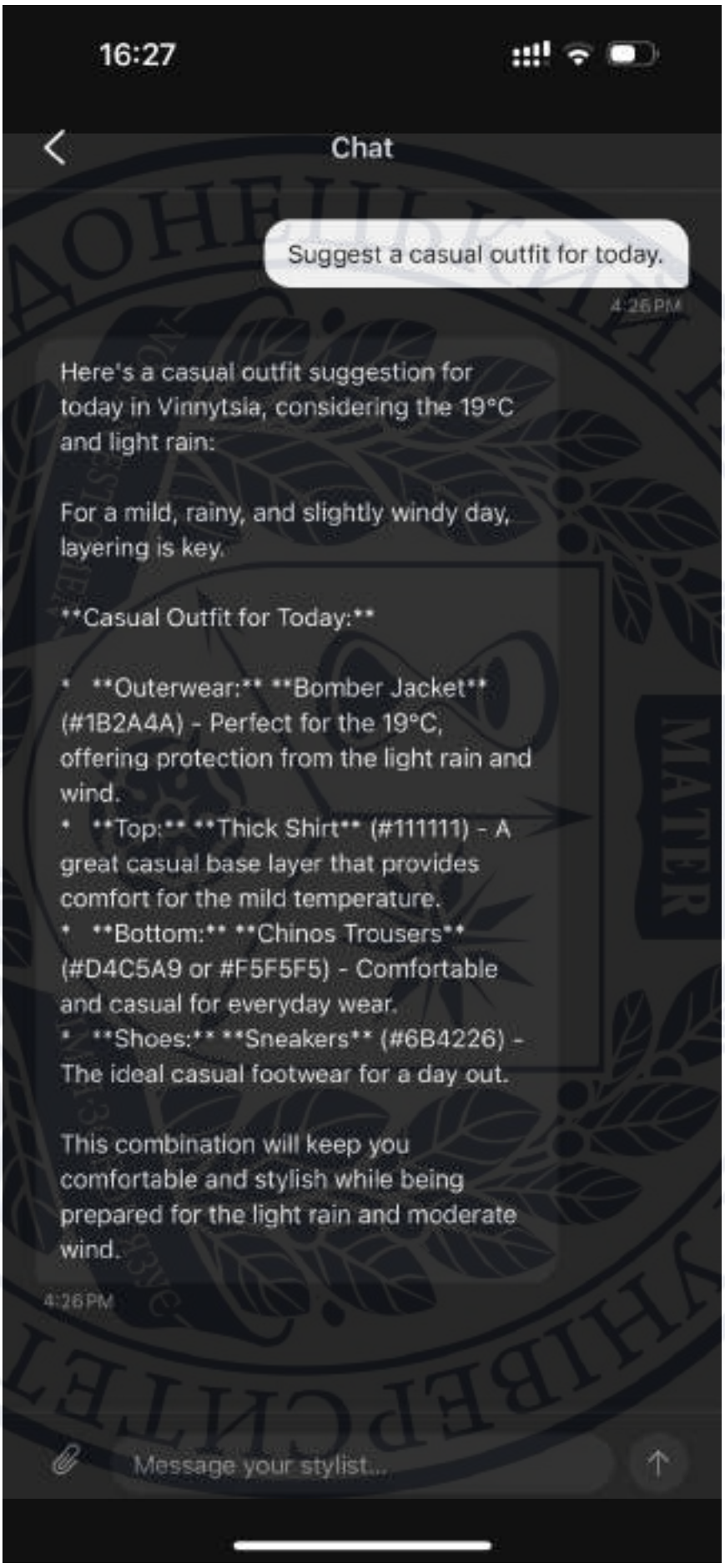

Рисунок Б.6 – Екран чату з ШІ асистентом

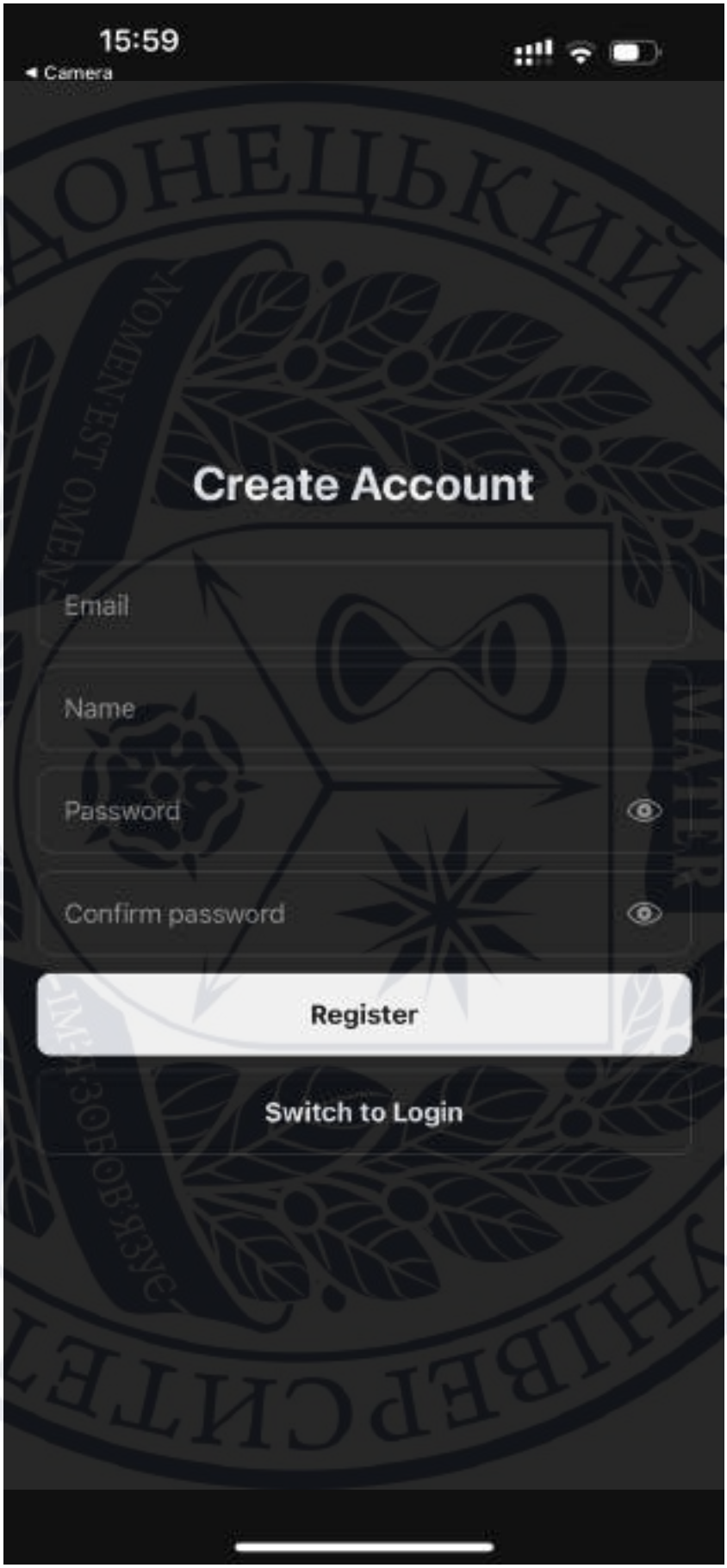

Рисунок Б.7 – Екран реєстрації

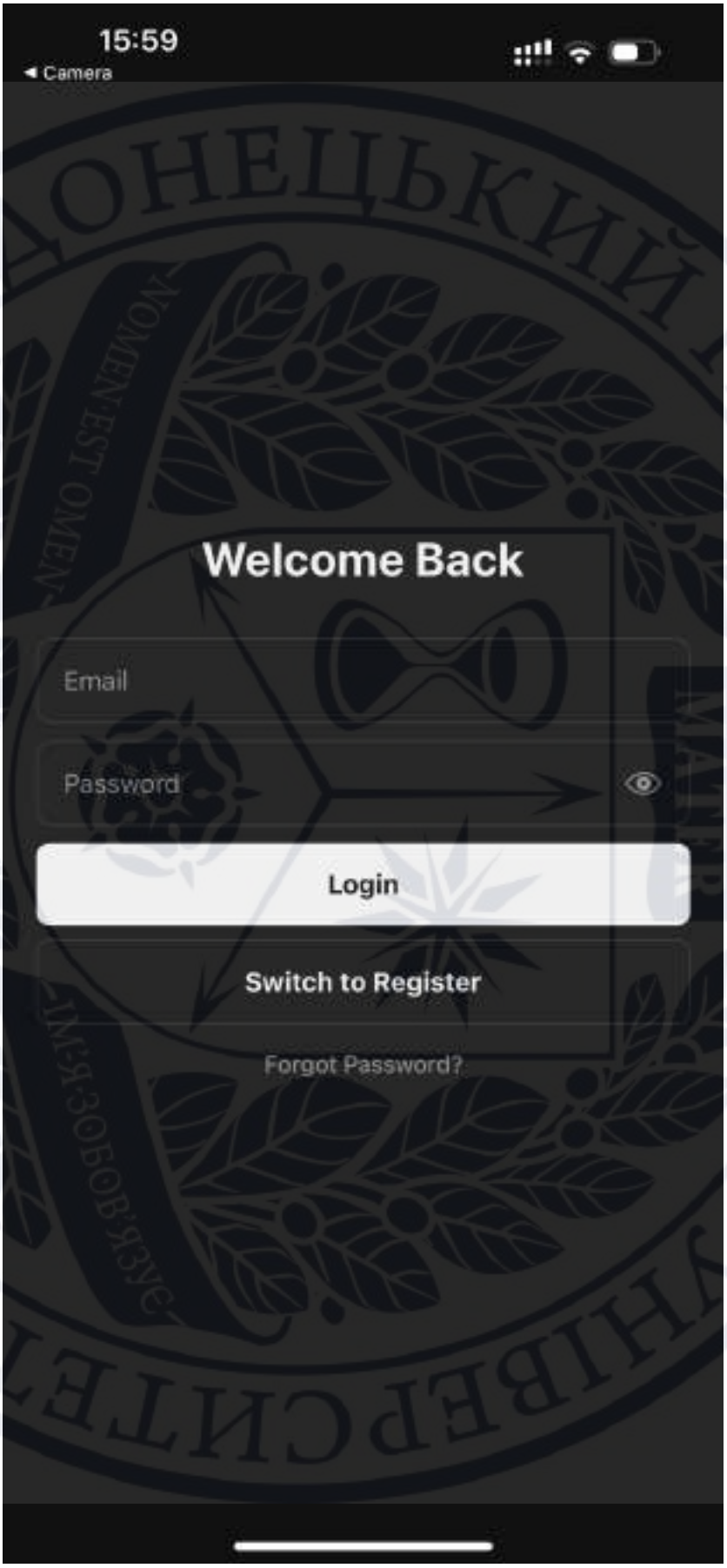

Рисунок Б.8 – Екран входу в додаток