НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних наук

_____ Голуб Б.Л.

" " _____ 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Програмне забезпечення мобільного додатку генерації рецептів на основі введених інгредієнтів»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми

к.т.н., доцент _____ /Вайганг Г.О. /
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

к.т.н., доцент _____ Бородкіна І.Л.
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Виконав _____ Свирид Д.А
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ

Завідувач кафедри

Комп'ютерних наук

_____Голуб Б.Л.

_“ ”_____20 р._

З А В Д А Н Н Я

на виконання бакалаврської кваліфікаційної роботи студенту

Свириду Дмитру Анатолійовичу

Спеціальність 121 – «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи: «Програмне забезпечення мобільного додатку генерації рецептів на основі введених інгредієнтів»

Затверджена наказом ректора НУБіП України від 16.12.2024 № 2249 «С».

Термін подання завершеної роботи на кафедру _____

Вихідні дані до бакалаврської кваліфікаційної роботи: документація бібліотек, нормативні та технічні документи

Перелік питань , які потрібно розробити: проаналізувати предметну область, аналоги, розробити вимоги до системи, спроектувати та розробити систему

Дата видачі завдання “_____”_____20___ р.

Керівник бакалаврської кваліфікаційної роботи

к.т.н., доцент_____ _____ ___Бородкіна І.Л. _

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _____ Свирид Д.А

(підпис) (ПІБ студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

БД – база даних

СУБД – симтема управління базою даних

HTTP – Hypertext Transfer Protocol

API – Application Programming Interface

JS – JavaScript

UUID – Universally Unique Identifier

ORM – Object-Relational Mapping

JWT – JSON Web Token

JSON – JavaScript Object Notation

JWS – JSON Web Signature

ВСТУП

Сучасний розвиток інформаційних технологій надає великі можливості для створення нових програмних рішень, здатних покращити якість життя користувачів та оптимізувати процеси в різних сферах діяльності. Одним із таких напрямів є розробка мобільних додатків, що допомагають користувачам у прийнятті рішень, спрямованих на досягнення конкретних цілей.

Одним із прикладів таких додатків є мобільний додаток для генерації рецептів на основі введених інгредієнтів. Цей інструмент здатний значно полегшити процес приготування їжі, дозволяючи користувачам швидко отримувати рецепти на основі тих продуктів, які вони мають у своєму розпорядженні.

Це питання особливо актуальне для молодих людей, які ведуть активний спосіб життя, та для тих, хто бажає оптимізувати процес приготування їжі, уникнути харчових відходів і зменшити витрати часу.

Об’єктом дослідження є створення програмне забезпечення мобільного додатку для генерації рецептів на основі введених інгредієнтів. Дослідження об’єкта включає аналіз функціональних і нефункціональних вимог, архітектурних рішень, засобів реалізації, а також способів забезпечення ефективності, точності та зручності використання такого програмного забезпечення. Створений мобільний додаток, повинен відповідати всім необхідним вимогам, а також сучасним стандартам розробки та засобам забезпечення захисту даних.

Предметом дослідження є процес генерації рецептів на основі введених користувачем інгредієнтів, генерація рецептів, можливість фільтрації за різними критеріями. Ці процеси виконують ключову роль у зручності користування додатком, дозволяючи користувачу швидко та ефективно знаходити відповідні рецепти, використовуючи доступні продукти.

Метаю бакалаврської роботи є аналіз актуальності проблематики, обґрунтування вибору інструментальних засобів та технологій, створення

технічного завдання, детальний аналіз предметної області та основних компонентів системи, проектування архітектури та інформаційної моделі додатку, розробка та реалізація програмного продукту. Необхідно проаналізувати сучасні підходи до генерації рецептів, дослідити функціональні та нефункціональні вимоги до мобільного додатку, визначити його архітектуру та створити інформаційну модель, що описує структуру та взаємодію компонентів системи. Також необхідно розробити дизайн користувацького інтерфейсу з урахуванням принципів зручності та реалізувати відповідне програмне забезпечення

Завдання розробки програмного додатку полягає у створенні мобільного застосунку, що дозволяє користувачам швидко і зручно отримувати рецепти на основі введених ними інгредієнтів. Завдяки реалізації цієї ідеї користувачі отримають можливість ефективно планувати своє харчування, використовуючи доступні продукти, що значно підвищує актуальність розроблюваного рішення.

Методи та технології, які використовуються при розробці. В процесі створення мобільного додатку застосовувалися сучасні підходи розробки програмного забезпечення, зокрема React Native для реалізації кросплатформенного мобільного застосунку. Використовувалися мова програмування TypeScript, бібліотека Axios для роботи з API, Firebase для авторизації користувачів, зберігання даних та забезпечення реального часу, а також i18next для підтримки різних мов.

Апробацією програмного додатку було представлено тези на навчальній конференції НУБІП.

Структура роботи складається зі вступу, 4 розділів, висновків, списку використаних джерел із 10 найменувань та 4 додатків. Загальний обсяг дипломної роботи становить 112 сторінок.

1 Системний аналіз предметної області

1.1 Опис предметної області

На сьогоднішній день мобільні додатки для підбору рецептів на основі введених інгредієнтів є важливою частиною сучасних технологій у сфері харчування. Тому мобільні програми, що дозволяють підбирати рецепти за наявними продуктами, набувають все більшої популярності серед споживачів. Розвиток таких додатків значно спрощує процес приготування їжі та підвищує ефективність у використанні продуктів. Вони можуть використовуватись не тільки в побуті, але й в ресторанному бізнесі для створення спеціалізованих меню або для персоналізованих рекомендацій для клієнтів.

Предметною областю цієї роботи є створення мобільного застосунку, який здатен генерувати кулінарні рецепти відповідно до переліку доступних інгредієнтів. Особливо цінним цей додадо є для людей з обмеженим бюджетом або тих, хто прагне експериментувати з кулінарією, використовуючи лише те, що вже є в холодильнику. Принцип роботи застосунку базується на введенні користувачем списку продуктів, після чого система пропонує відповідні варіанти рецептів.

Метою створюваного мобільного додатку для генерації рецептів на основі введених інгредієнтів є забезпечити користувачам швидкий та інтуїтивно зрозумілий спосіб пошуку кулінарних рецептів, використовуючи продукти, що перебувають у їх розпорядженні..

Мобільний додаток створювався, щоб мінімізувати витрати часу на планування меню та максимально ефективно використовувати наявні харчові ресурси для створення поживних та смачних страв. Це сприяє покращенню загального кулінарного досвіду користувачів та раціональному використанню продуктів, що, у свою чергу, зменшує кількість харчових відходів.

Актуальність цієї теми зумовлена обґрунтовується зростаючим попитом суспільства на розумні технологічні рішення, здатні покращити

якість повсякденного життя. Мобільний додаток для генерації рецептів є корисним інструментом як для приватних осіб, так і для підприємств, що займаються харчуванням або постачанням продуктів. Зокрема, для ресторатоpів та виробників продуктів харчування таке програмне забезпечення може стати важливим елементом, що сприяє підвищенню ефективності їх роботи та оптимізації асортименту

1.2 Аналіз вимог до програмної системи

Одним з найважливіших початкових етапів розробки будь-якого програмного продукту, зокрема мобільного додатку для генерації рецептів за введеними інгредієнтами, є аналіз вимог до майбутньої системи. Даний етап є фундаментальним, оскільки правильне і повне визначення вимог є ключовим фактором, який впливає на якість і успіх кінцевого програмного продукту. Для забезпечення ефективності аналізу вимог слід враховувати як функціональні, так і нефункціональні вимоги до програмного продукту [1].

1.2.1 Визначення функціональних вимог. Функціональні вимоги до системи описують, що саме повинна робити програмна система, які дії вона має виконувати, і які функції повинні бути доступні користувачам. У контексті додатку для генерації рецептів ці вимоги включають можливість введення користувачем різноманітних інгредієнтів, що є важливим аспектом, адже саме ця функція дозволяє додатку генерувати релевантні рецепти.

Крім того, програмна система також повинна містити функціонал автоматичного підбору рецептів відповідно до вибраних інгредієнтів. Це є ключовою особливістю, яка визначає цінність продукту для користувача. Важливою також є можливість перегляду детальної інформації про кожен рецепт. Ця інформація повинна включати кроки приготування, кількість порцій, поживну цінність, час приготування та інші деталі.

Також слід зазначити, що функціональні вимоги мають передбачати систему автентифікації та авторизації користувачів. Цей функціонал надає можливість користувачам створювати власні профілі, зберігати улюблені

рецепти, переглядати історію пошуку та використовувати додатковіперсоналізовані можливості додатку. Наявність персоналізації також є важливим фактором, який сприяє лояльності користувачів до продукту.

1.2.2 Визначення нефункціональних вимог. Окрім функціональних вимог, значну увагу необхідно приділити і нефункціональним вимогам, які визначають, як саме система повинна працювати, зокрема продуктивність, надійність, зручність використання та масштабованість.

Перш за все, слід наголосити на вимогах до продуктивності системи. Враховуючи, що мобільні користувачі цінують швидкість і оперативність отримання результатів, додаток має забезпечувати швидку генерацію рецептів та миттєвий відгук на дії користувача. Надійність системи передбачає стабільну роботу без збоїв і помилок, що є необхідним для підтримання високого рівня довіри користувачів.

Особливу увагу слід звернути на зручність використання програмного продукту, оскільки зручний і зрозумілий інтерфейс значно полегшує взаємодію користувачів з додатком. З цією метою важливо забезпечити зрозумілу і логічну структуру інтерфейсу, просту навігацію та інтуїтивне управління додатком. Крім того, необхідно передбачити адаптивний дизайн, що забезпечить зручність використання додатку на різних мобільних пристроях з різною роздільною здатністю екрану.

Не менш важливою нефункціональною вимогою до мобільного додатку є масштабованість. Програмний продукт повинен бути розроблений таким чином, щоб легко адаптуватись до збільшення кількості користувачів та обсягу даних, які обробляє система. Це дозволить забезпечити стабільність роботи додатку навіть при значних навантаженнях.

Таким чином, детальний аналіз як функціональних, так і нефункціональних вимог дозволяє сформувати чітке уявлення про майбутній продукт, встановити його межі і визначити ключові аспекти, які будуть

враховані в процесі розробки. Такий підхід є запорукою створення якісногопрограмного забезпечення, що задовольнятиме потреби користувачів та відповідатиме сучасним технологічним стандартам.

1.3 Моделювання предметної області

1.3.1 Загальні відомості. Процес побудови програмного забезпечення неможливо уявити без детального моделювання предметної області. Саме на цьому етапі розробники мають змогу детально розглянути завдання, які ставляться перед майбутньою системою, зрозуміти, для кого вона створюється і як користувачі будуть із нею працювати. Моделювання забезпечує можливість формалізувати та зафіксувати всі ключові процеси, які відбуватимуться в рамках застосунку для генерації рецептів на основі введених інгредієнтів, і дати точне уявлення про внутрішню та зовнішню логіку роботи системи.

Особливе місце у моделюванні займає використання уніфікованої мови моделювання (UML), яка дозволяє створювати різноманітні діаграми – як структурні, так і поведінкові. Такий підхід дає змогу не лише формалізувати вимоги до додатку, але й забезпечити прозорість процесу розробки для всіх учасників проєкту, від розробників до зацікавлених сторін.

1.3.2 Мета використання UML-діаграм. Метою створення UML —діаграм у рамках цього дипломного проєкту є побудова комплексної, багатовимірної моделі предметної області, що дозволяє:

Виявити й описати основні ролі користувачів додатку, а також функціональні можливості, які система повинна забезпечити кожному з них.

Відобразити взаємодію між користувачами й програмним продуктом через різні сценарії використання.

Зобразити часову та логічну послідовність виконання дій та обробки даних.

Також необхідно ретельно формалізувати алгоритми функціонування системи, включаючи аспекти умовних переходів, розгалужень та паралельних процесів.

Таким чином, доцільно звернути увагу на діаграми, такі як діаграма прецедентів, діаграма послідовності та діаграма активності, оскільки вони наочно демонструють ключові аспекти роботи системи, які вимагають особливої уваги.

1.3.3 Діаграма прецедентів (Use Case Diagram). Діаграма прецедентів є одним із ключових інструментів у процесі моделювання, адже саме вона дозволяє наочно представити всі можливі сценарії використання додатку з позиції користувача. Зокрема, у контексті мобільного додатку для генерації рецептів, на діаграмі прецедентів можна виділити наступних акторів:

1. Звичайний користувач, який вводить наявні інгредієнти, отримує рецепти, переглядає інструкції приготування, додає рецепти до обраного тощо.
2. Адміністратор, який відповідає за модерацію контенту, оновлення бази інгредієнтів, управління списком рецептів.

На діаграмі прецедентів відображаються основні сценарії використання:

1. Введення інгредієнтів і отримання списку відповідних рецептів.
2. Перегляд детальної інформації про рецепт (інгредієнти, опис, кроки приготування, калорійність тощо).
3. Збереження рецепту у “вибране”.
4. Додавання власного рецепту.
5. Реєстрація та вхід у систему.
6. Оцінювання й коментування рецептів.

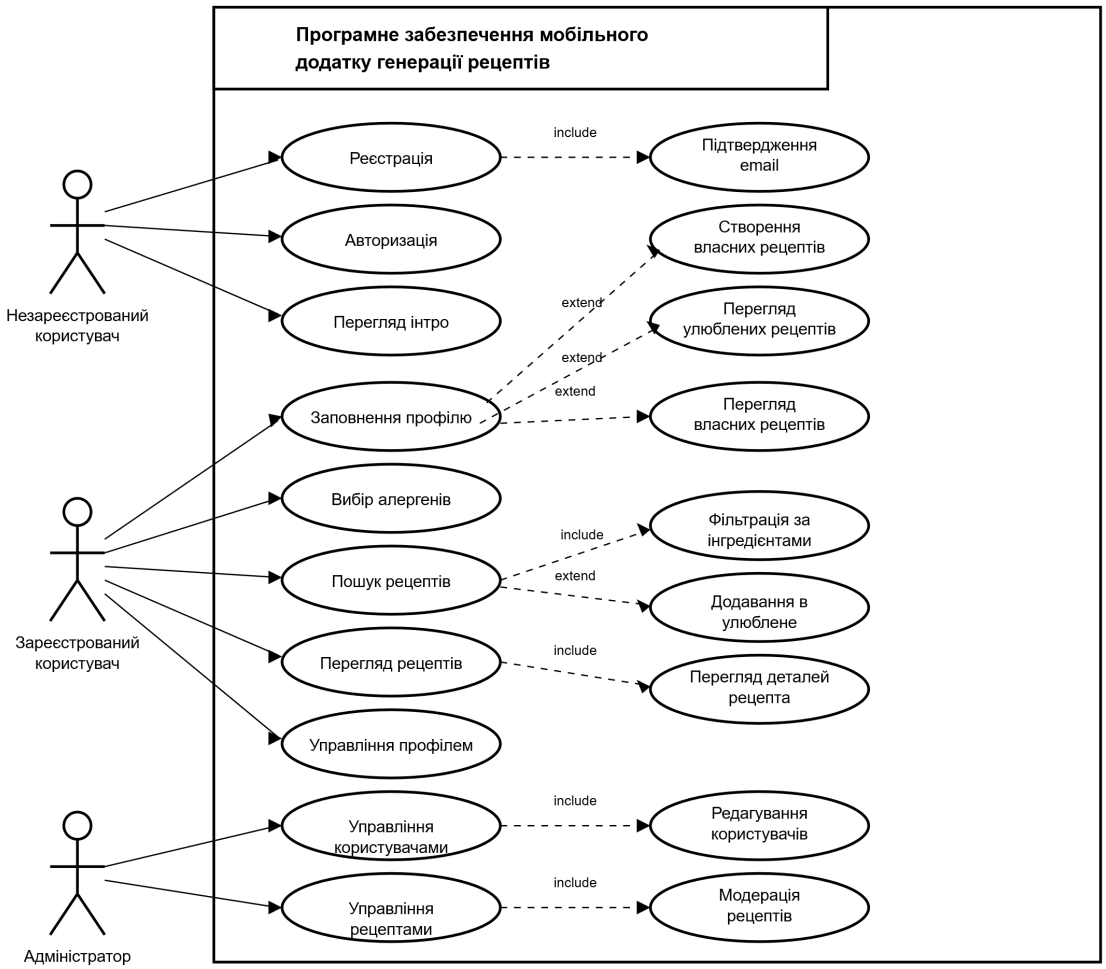

Рис. 1.1 – Діаграма прецедентів для мобільного додатку генерації рецептів

Використання діаграми прецедентів дозволяє не лише структуровано підійти до формалізації функціоналу, а й уникнути пропуску важливих сценаріїв, які можуть бути критичними для кінцевого користувача.

1.3.4 Діаграма послідовності (Sequence Diagram). Діаграма послідовності є потужним інструментом для відображення порядку взаємодії між різними об’єктами та компонентами системи під час виконання певного сценарію. У випадку розробки додатку для генерації рецептів, найбільш показовим є сценарій, який відноситься до створення рецепту відповідно до інгредієнтів, введених користувачем.

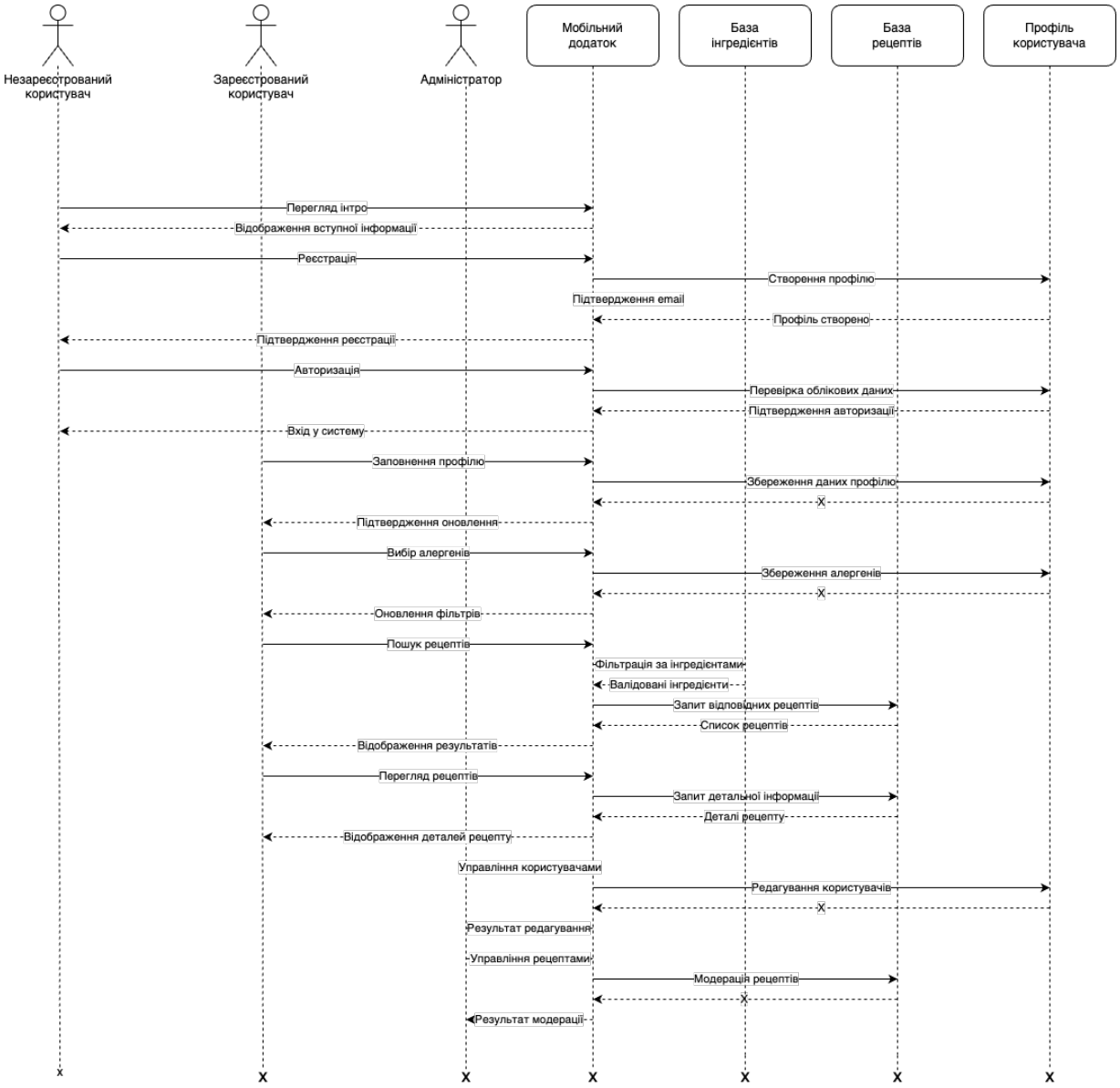

Рис. 1.2 – Діаграма послідовності для мобільного додатку генерації рецептів

На цій діаграмі фіксується взаємодія між:

- Користувачем.
- Інтерфейсом додатку.
- Хмарним сервісом (API для взаємодії з базою даних рецептів).
- Базою даних інгредієнтів/рецептів.

Послідовність дій зазвичай виглядає так:

1. Користувач вводить список інгредієнтів і надсилає запит.
2. Інтерфейс додатку обробляє запит і пересилає його на сервер.
3. Серверна частина здійснює пошук та обробку даних, підбираючи відповідні рецепти.
4. Отримані результати надсилаються назад у додаток.
5. Користувач переглядає сформований список рецептів.

Завдяки такій діаграмі можна чітко простежити потік даних і взаємодій, а також вчасно ідентифікувати “вузькі місця” в алгоритмах чи можливі проблеми з асинхронністю або черговістю виконання операцій.

1.3.5 Діаграма активності (Activity Diagram). Діаграма активності дозволяє глибоко проаналізувати логіку виконання окремих бізнес-процесів, зокрема таких, як “Пошук та підбір рецепту за інгредієнтами”. Цей тип діаграм наочно демонструє, як саме відбувається послідовність дій, починаючи від введення користувачем даних і закінчуючи отриманням готового рецепту чи рекомендації.

Головними етапами, які відображаються на діаграмі активності, є:

1. Початок процесу (ініціація дії користувачем).
2. Введення інгредієнтів.
3. Валідація введених даних.
4. Запит до бази рецептів.
5. Фільтрація та ранжування отриманих результатів.
6. Відображення релевантних рецептів.
7. Можливість переходу до перегляду детальної інформації.
8. Завершення сценарію.

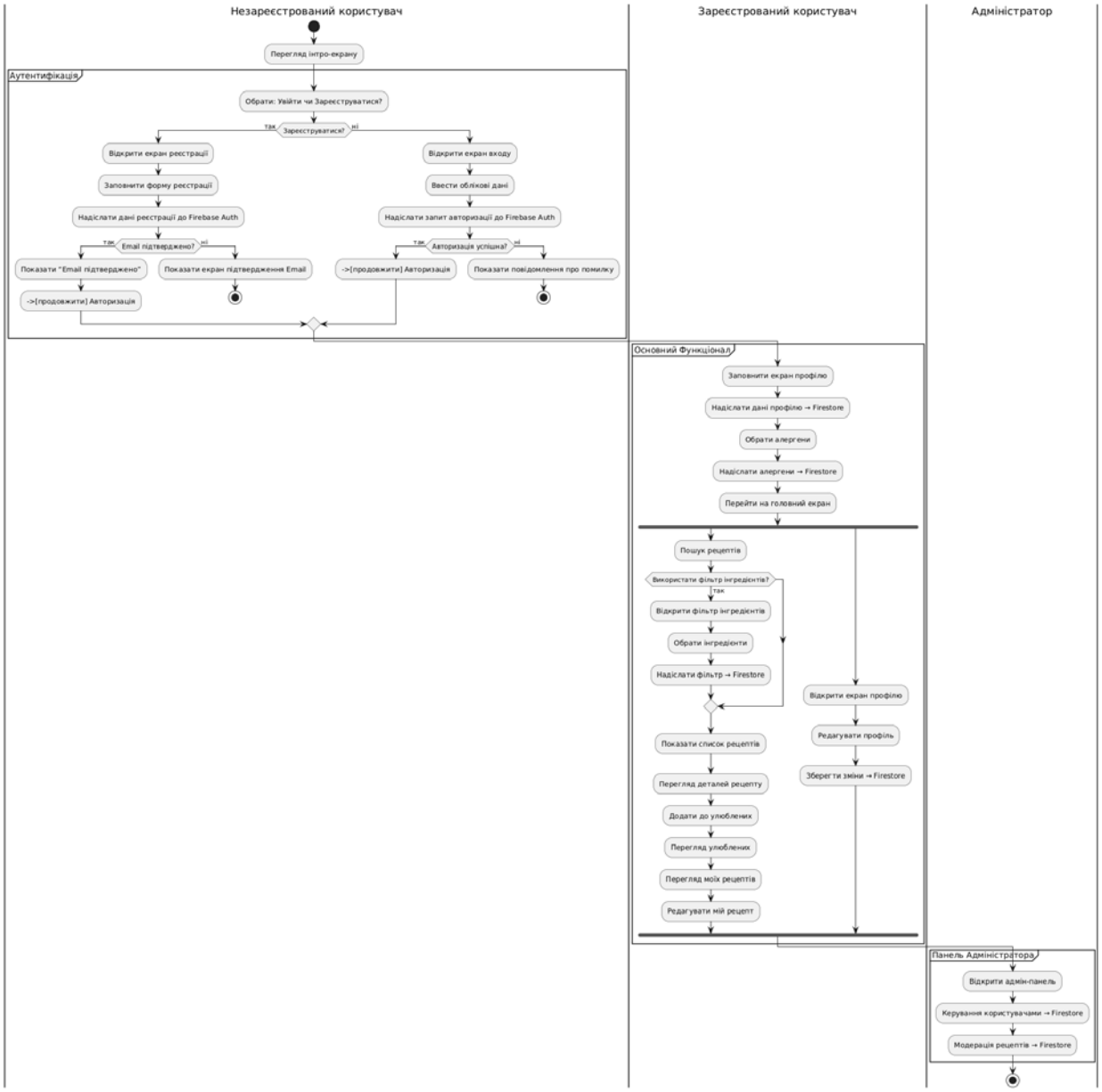

Рис. 1.3 – Діаграма активності для мобільного додатку генерації рецептів

Діаграма активності наочно показує наявність розгалужень (наприклад, коли для введених інгредієнтів не знайдено жодного рецепту, система пропонує додати власний рецепт або змінити список інгредієнтів), а також можливість паралельного виконання деяких підпроцесів (наприклад, одночасне завантаження зображень і текстових інструкцій).

1.3.6 Важливість моделювання та застосування діаграм. Загалом, використання UML-діаграм дозволяє підвищити прозорість і керованість процесу розробки, уникнути двозначності у трактуванні вимог і забезпечити спільне розуміння структури та поведінки системи всіма учасниками проєкту. Формалізовані моделі предметної області є надійною основою для подальшого проектування архітектури програмного продукту, спрощують комунікацію між командою та замовником, а також мінімізують ризик виникнення помилок на пізніших етапах розробки.

У цьому розділі подано основні UML-діаграми (діаграма прецедентів, діаграма послідовності, діаграма активності), які були побудовані для даного мобільного додатку. Вони ілюструють структуру, поведінку та сценарії використання системи у вигляді, що максимально наближений до реальних вимог та очікувань кінцевих користувачів [2].

1.4 Огляд інформаційних джерел та існуючих рішень.

Вивчення та аналіз наявних інформаційних джерел і існуючих рішень є надзвичайно важливим етапом у процесі розробки будь-якого програмного продукту. Саме цей етап дозволяє, з одного боку, виявити основні методи, що вже використовуються в подібних застосунках, а з іншого — уникнути повторення ідентичних функціональних можливостей. Також він дощволяє виявити недоліки конкурентних рішень і представити власні оригінальні підходи.

1.4.1 Дослідження існуючих рішень. Перш за все, аналіз інформаційних джерел передбачає знайомство з актуальними публікаціями, статтями, науковими роботами, довідковими матеріалами та відкритими базами знань, присвяченими створенню програмного забезпечення для кулінарної сфери, мобільним додаткам, а також алгоритмам підбору рецептів на основі вхідних даних. Дослідження наукової літератури дозволило зробити висновок, що останнім часом все більшої популярності набувають мобільні

застосунки, які орієнтуються на персоналізований підбір рецептів з урахуванням смакових вподобань, дієтичних обмежень та наявних інгредієнтів.

Для більш детального розуміння ринку та вибору кращих підходів до розробки власного мобільного додатку, було здійснено порівняльний аналіз найбільш популярних сучасних рішень у сфері підбору рецептів за наявними інгредієнтами. Огляд охоплює такі додатки: SuperCook, Yummly, Tasty, Allrecipes Dinner Spinner та Cookpad. Для зручності результати порівняння представлені у вигляді таблиці 1.1.

Першим було розглянуто застосунок SuperCook.

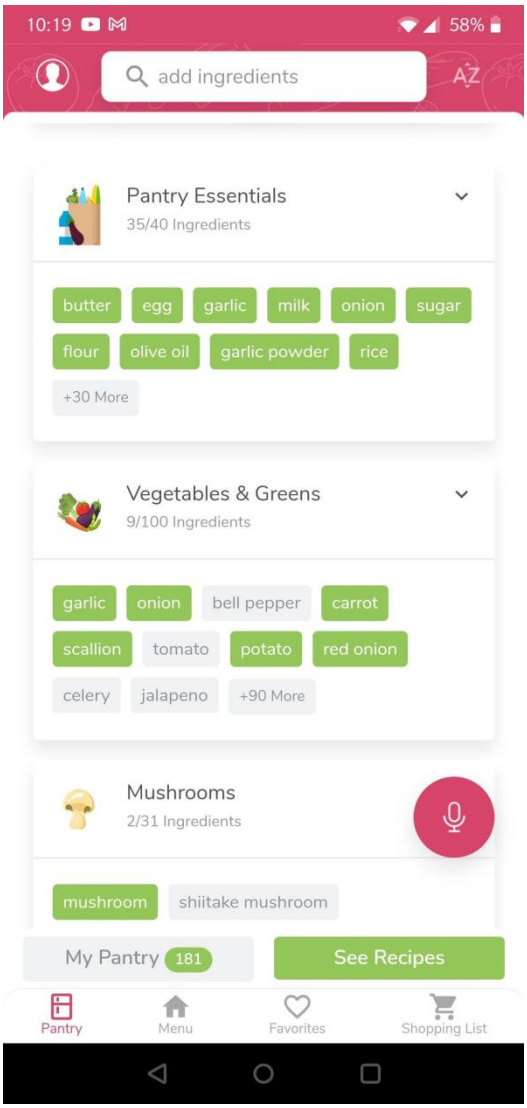

Рис. 1.4. Скріншот роботи застосунку «SuperCook»

Додаток спеціалізується саме на підборі рецептів за введеними інгредієнтами. Інтерфейс простий: достатньо вказати, що є у холодильнику, і система пропонує всі можливі страви. Відзначається висока швидкість роботи, проте відсутня персоналізація за алергенами або смаковими уподобаннями.

Далі було проаналізовано Yummly.

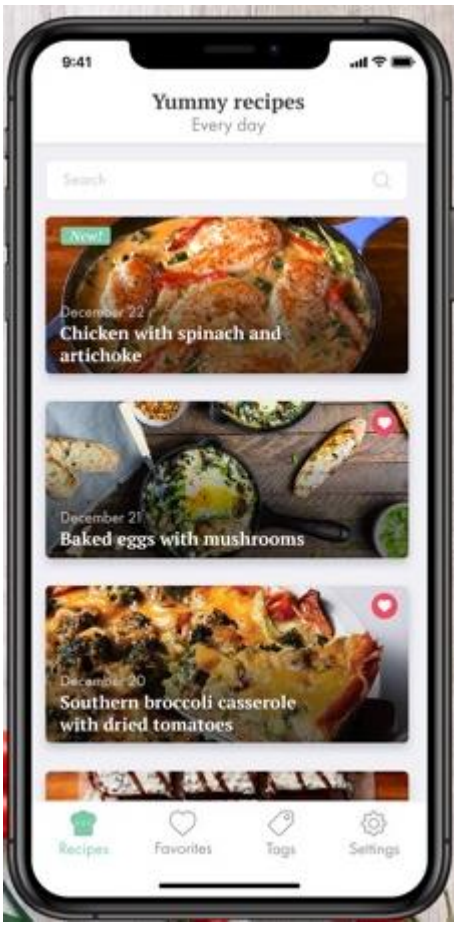

Рис. 1.5 Скріншот інтерфейсу додатку «Yummly»

Цей застосунок відомий своєю гнучкою системою налаштувань під користувача. В ньому можна врахувати дієтичні побажання, харчові алергени, національну кухню тощо. Рецепти підбираються не лише за інгредієнтами, а й з урахуванням персональних вподобань.

Наступним об’єктом для аналізу став додаток Tasty.

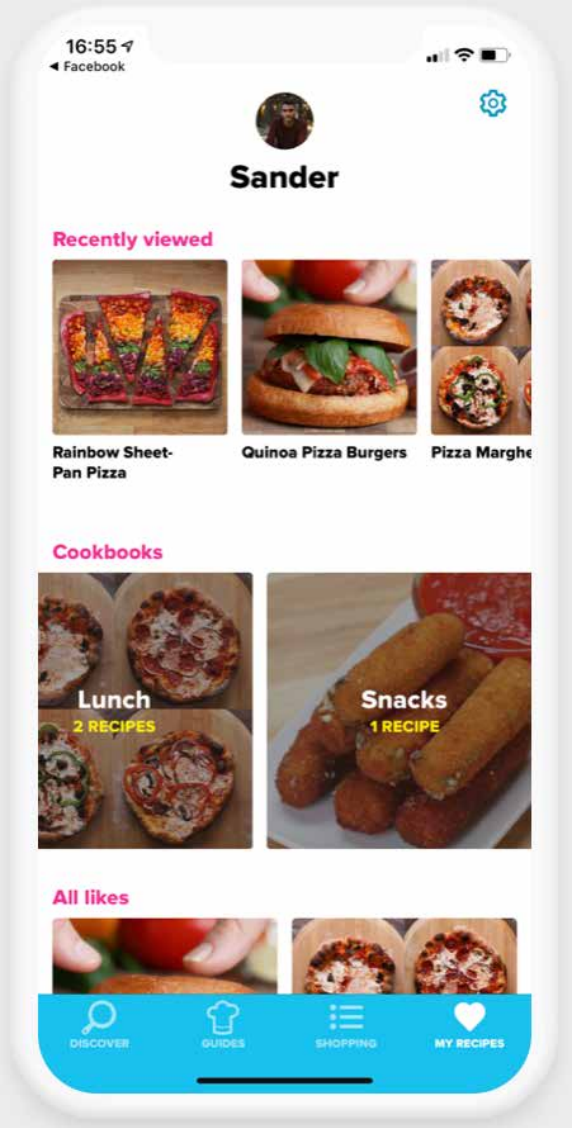

Рис. 1.6 Скріншот мобільного додатку «Tasty»

Основна перевага Tasty — інтеграція з відеоінструкціями. Багато рецептів супроводжуються детальними відеороликами, що особливо зручно для новачків.

Звернуто увагу також на Allrecipes Dinner Spinner .

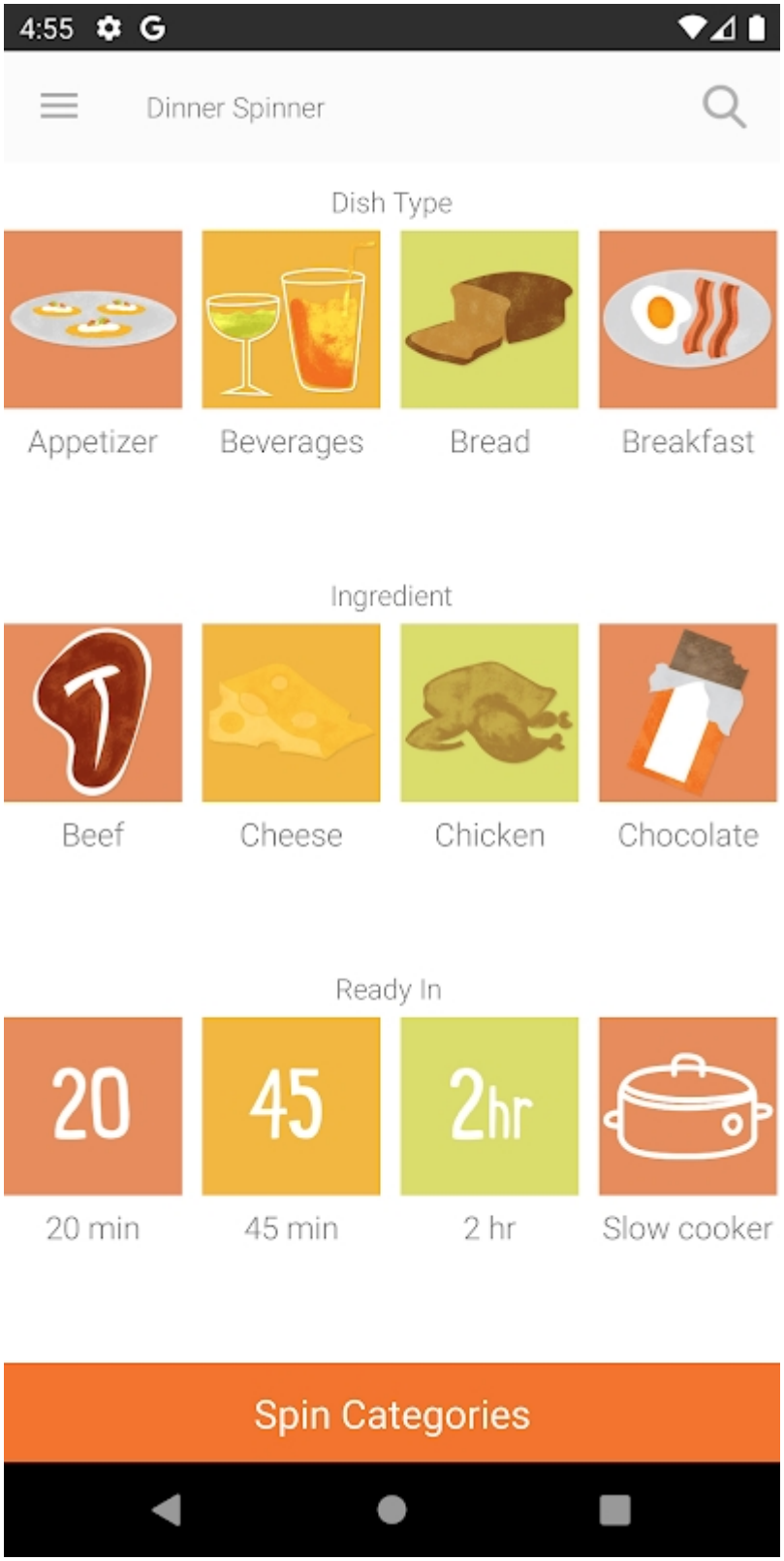

Рис. 1.7 Головний екран додатку «Allrecipes Dinner Spinner»

Allrecipes є одиним з найбільших кулінарних сервісів, що дає змогу шукати рецепти, переглядати рейтинги та відгуки. Проте він не надто зручний для підбору страв саме за списком власних інгредієнтів.

Завершує огляд Cookpad.

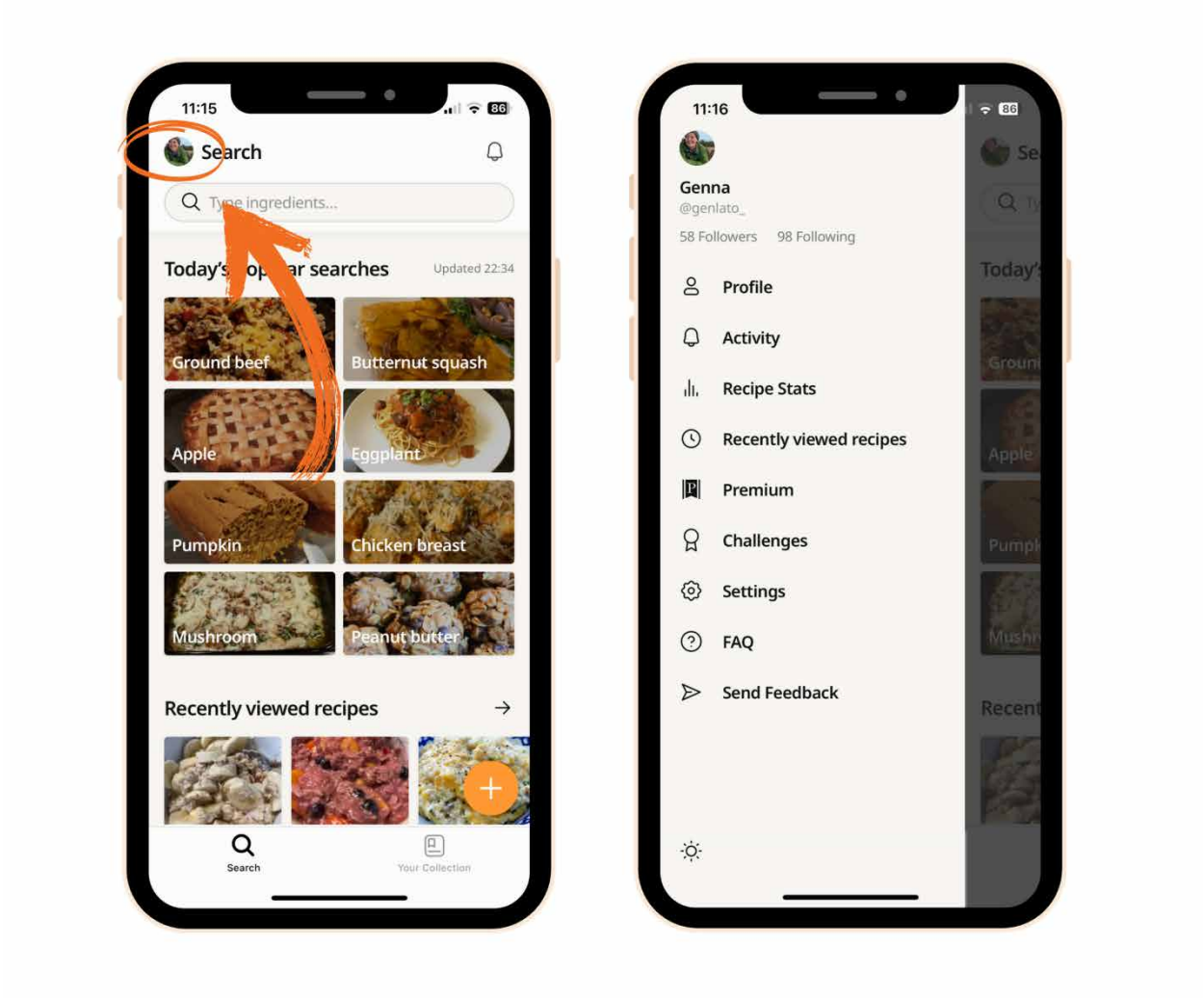

Рис. 1.8 Інтерфейс додатку «Cookpad»

На відміну від попередніх сервісів, тут в цьому додатку акцент зроблено на взаємодії користувачів. Кожен користувач може додавати свої рецепти, ділитися досвідом, коментувати та додавати до улюблених.

Таблиця 1.1. Порівняльна таблиця мобільних додатків для рецептів

Функція	**SuperCook**	**Yummly**	**Tasty**	**Allrecipes**	**Cookpad**
Введення інгредієнтів	+++	++	+	+	+
Автоматичний підбір рецептів	+++	++	++	++	+
Персоналізація (смаки, алергени)	–	++	+	+	+
Оффлайн-доступ	–	–	–	–	–
Оцінки/відгуки	+	++	++	+++	+++
Додавання власних рецептів	–	–	–	–	+++
Інтерфейс українською	–	–	–	–	–

Умовні позначення:

+++ — реалізовано повністю

++ — реалізовано частково

— реалізовано базово

– — функція відсутня

1.4.2 Висновок аналізу існуючих рішень. Отже, детальний аналіз показує, що жоден із існуючих рішень не є ідеальним з точки зору користувача, який прагне максимально швидко й ефективно знайти рецепт саме зі своїх інгредієнтів, особливо якщо таких інгредієнтів небагато, або вони специфічні. Часто додатки пропонують великий вибір страв, однак алгоритми підбору не завжди працюють коректно — частина інгредієнтів не враховується, результати не працюють за ступенем відповідності чи складністю приготування. Деякі застосунки вимагають авторизації навіть для базового функціоналу, що може відштовхнути частину користувачів.

Варто окремо згадати інформаційні ресурси, які використовуються для наповнення бази рецептів та інгредієнтів. Це, зокрема, відкриті API, відкриті кулінарні спільноти, форуми та тематичні блоги. Такі ресурси дають змогу поповнювати базу новими рецептами, слідкувати за сучасними кулінарними тенденціями та отримувати зворотний зв’язок від реальних користувачів.

Крім того, важливим аспектом є аналіз інтерфейсних рішень, включаючи присутність ефективної системи фільтрів, інтуїтивно зрозумілу навігацію, адаптивність для різних типів мобільних пристроїв, а також можливість збереження улюблених рецептів. У більшості сучасних додатків спостерігається тенденція до спрощення інтерфейсу та максимізації автоматизації взаємодії з користувачем. Однак, часто ігнорується можливість налашування власних фільтрів.

Отже, проведений огляд інформаційних джерел та існуючих рішень, можна зробити висновок, що попри різноманіття застосунків для підбору рецептів, на ринку досі залишається потреба для більш персоналізованого, простого і водночас функціонального мобільного додатку, який би не лише генерував рецепти на основі введених інгредієнтів, а й дозволяв враховувати особливі запити користувача, легко знаходити необхідну страву та отримувати детальні покрокові інструкції. Саме на реалізацію зазначеного підходу орієнтована ця дипломна робота.

1.5 Постановка завдання.

Постановка завдання — це ключовий етап, що формує чітке розуміння цілей, вимог і обмежень майбутнього продукту. Завдяки детальному формулюванню вимог і ретельному окресленню меж проєкту, з'являється можливість оптимізувати процес розробки та сконцентруватися на тих аспектах, які дійсно важливі для користувача й для досягнення конкурентних переваг на ринку мобільних застосунків [2].

1.5.1 Передумови й актуальність задачі. Сучасний ринок мобільних застосунків сповнений різноманітних сервісів для кулінарії. Проте серед них досі залишаються незадоволені потреби, а саме: швидкий підбір рецептів із наявних інгредієнтів, мінімальна кількість кроків для отримання результату, підтримка користувача без реєстрації та простий інтерфейс, адаптований для щоденного використання.

Беручи до уваги ці потреби, постає завдання створити застосунок, який би поєднував зручність, персоналізацію та сучасні технічні можливості.

1.5.2 Головна мета та очікуваний результат. Основною метою цього проєкту є розробка мобільного додатку, що дозволяє користувачу отримувати релевантні кулінарні рецепти на основі введених ним інгредієнтів. Продукт має бути інтуїтивно зрозумілим, швидким у роботі та доступним для широкого кола користувачів.

Ключові очікувані результати:

1. Створення стабільної, надійної та простої у користуванні платформи.
2. Забезпечення персоналізованого підбору страв із врахуванням переваг користувача.
3. Можливість інтеграції із зовнішніми базами рецептів і подальшого розширення функціоналу.

1.5.3 Функціональні вимоги до системи. Для досягнення поставленої мети передбачено ряд основних функцій, які мають бути реалізовані в рамках мобільного застосунку. Вимоги до функціоналу системи подані у вигляді таблиці 1.2.

Таблиця 1.2. Основні функціональні можливості мобільного додатку

Номер	Функціонал	Опис реалізації
1	Вибір інгредієнтів	Користувач може вручну додавати інгредієнти, що є у нього під рукою
2	Генерація списку рецептів	Система автоматично підбирає релевантні рецепти відповідно до введених даних
3	Перегляд детальної інформації про рецепт	Відображення повного складу, кроків приготування, фото, калорійності тощо
4	Збереження рецептів у вибране	Можливість додати рецепт до персонального списку обраного
5	Оцінювання та коментування рецептів	Додавання коментарів (за бажанням)
6	Реєстрація та вхід	Створення профілю
7	Додавання власних рецептів	Користувач може створити та поділитися власним рецептом
8	Генерація рецептів	Якщо користувач не знайшов рецепт за вибраними інгредієнтами, є можливість згенерувати рецепт
9	Фільтрація за алергенами, категоріями, типами страв	Вибір страв відповідно до особистих вподобань або обмежень

1.5.4 Технічні (нефункціональні) вимоги. Окрім основних функцій, важливо визначити критерії якості системи, що суттєво впливають на сприйняття додатку користувачем та подальшу підтримку:

1. Продуктивність: система повинна забезпечувати швидкий пошук та підбір рецептів, навіть при великій кількості користувачів одночасно.
2. Безпека: захист персональних даних через сучасні протоколи шифрування, безпечну автентифікацію.
3. Інтерфейс: простий, логічний, інтуїтивно зрозумілий дизайн, адаптація під різні розміри екранів.
4. Масштабованість: можливість розширення функціоналу без значних доопрацювань архітектури.
5. Доступність: стабільна робота на більшості сучасних пристроїв з Android та iOS.

Таблиця 1.3 – Основні нефункціональні вимоги

Номер	**Критерій**	**Вимога**
1	Відповідь системи	Відображення результатів пошуку не більше, ніж за 2-3 секунди
2	Надійність	Відсутність критичних помилок у роботі додатку
3	Захист даних	Всі персональні дані користувачів мають бути захищені відповідно до чинних стандартів
4	Адаптивність	Коректне відображення інтерфейсу на смартфонах з різними діагоналями

1.5.5 Потенційні обмеження та рамки розробки. В ході формування архітектури та вибору підходів до реалізації мобільного додатку були визначені наступні обмеження, які враховано на етапі розробки:

1. Відсутність сторонніх сервісів: Для забезпечення повної автономності та контролю над усіма даними й логікою додатку, було прийнято рішення не використовувати зовнішні сервіси або API. Всі функції реалізуються власними засобами, а база рецептів та користувацькі дані зберігаються на серверній частині, розробленій спеціально для цього проєкту.
2. Кросплатформеність (Android та iOS): Додаток розробляється як для операційної системи Android, так і для iOS. Для цього застосовуються сучасні кросплатформені технології, що дозволяють забезпечити однакову функціональність, зовнішній вигляд і якість користувацького досвіду на різних типах мобільних пристроїв.
3. Локалізація: На першому етапі додаток підтримує українську та англійську мови інтерфейсу. Це дає змогу охопити більшу цільову аудиторію й зробити продукт доступним як для українських користувачів, так і для користувачів за кордоном.
4. Обмеження MVP: На етапі розробки мінімально життєздатного продукту (MVP) фокус зроблено на основному функціоналі, без впровадження додаткових, менш критичних опцій. Подальше розширення функцій планується відповідно до зворотного зв’язку від реальних користувачів.

1.6 Висновки до розділу 1.

У межах першого розділу було проведено системний огляд проблематики та сучасного стану розробки мобільних додатків для генерації рецептів на основі введених інгредієнтів. Спочатку було проаналізовано актуальність цієї тематики, адже попит на зручні, персоналізовані сервіси для

підбору страв невпинно зростає разом із розширенням мобільних платформ та зміною підходів до приготування їжі в повсякденному житті.

Детально розглянуто предметну область, з’ясовано основні проблеми, які виникають у користувачів під час пошуку та підбору рецептів із власних інгредієнтів, а також досліджено основні вимоги до подібних застосунків. На цьому етапі було чітко сформульовано цілі і задачі, визначено функціональні й нефункціональні вимоги до майбутнього продукту, окреслено його обмеження та очікувані результати.

Окрему увагу приділено аналізу інформаційних джерел та вже існуючих рішень. Проведено порівняльний огляд популярних мобільних додатків, що спеціалізуються на підборі рецептів, здійснено їхнє порівняння за ключовими функціональними характеристиками, наведено сильні та слабкі сторони кожного із рішень, а також проаналізовано особливості їх інтерфейсів.

Важливою частиною розділу стало моделювання предметної області. Було побудовано основні UML-діаграми — діаграму прецедентів, діаграму послідовності та діаграму активності. Це дозволило більш наочно побачити логіку роботи майбутньої системи, визначити основних акторів, їх ролі й сценарії взаємодії, а також деталізувати порядок обробки даних у межах застосунку.

На завершення сформульовано постановку завдання для розробки мобільного додатку. Узагальнено ключові функції, вимоги до якості, особливості платформи та можливі обмеження, що створює чітке підґрунтя для подальших етапів проектування та реалізації системи.

Таким чином, матеріали першого розділу закладають міцну теоретичну та методологічну основу для проєктування архітектури, визначення структури БД, проектування інтерфейсів та подальшої реалізації мобільного застосунку, орієнтованого на зручність і реальні потреби користувачів.

2 Проєктування інформаційного та програмного забезпечення

2.1 Логічна модель даних у вигляді ER діаграми

Етап проектування інформаційної складової програмної системи традиційно починається із створення логічної моделі даних. Саме логічна модель дозволяє глибше зрозуміти структуру даних, які обробляє система, встановити зв’язки між основними сутностями та передбачити особливості їх взаємодії у межах майбутнього застосунку.

Для наочного подання структури даних використовується ER-діаграма (Entity-Relationship Diagram, діаграма “сутність–зв’язок”). Даний тип діаграм уже давно зарекомендував себе як ефективний інструмент формалізації моделі даних у вигляді зрозумілої для всіх учасників проєкту схеми.

Перед тим, як розпочати детальне проектування інформаційного та програмного забезпечення, одним із ключових завдань є побудова логічної моделі даних, яку зазвичай представляють у вигляді ER-діаграми. Саме цей етап дає змогу не лише візуалізувати структуру майбутньої системи, але й здійснити перевірку відповідності моделі основним принципам організації даних.

При проектуванні власної моделі особливу увагу було приділено тому, щоб структура даних відповідала вимогам реляційної моделі й нормалізації, зокрема — третій нормальній формі. Це означає, що всі атрибути кожної сутності є функціонально незалежними від неключових атрибутів, і дублювання інформації у структурі зведено до мінімуму. Завдяки цьому досягається цілісність даних, спрощується оновлення й пошук інформації, а також зменшується ризик виникнення аномалій при виконанні операцій над базою.

Водночас, слід відзначити, що для побудови інформаційної системи було обрано не реляційну, а документно-орієнтовану модель зберігання даних Firestore. У цій моделі основними об’єктами є колекції та документи, які містять гнучко структуровані дані. Такий підхід дозволяє ефективно працювати з великими обсягами даних, швидко здійснювати вибірку й

масштабувати систему без втрати продуктивності. У межах цієї архітектури також можна уникнути надмірності за рахунок добре розробленого плану вкладених об'єктів і взаємозв'язків між ними. Це забезпечує ідеальний баланс між структурованістю, масштабованістю та гнучкістю інформаційної системи.

В процесі аналізу предметної області майбутнього мобільного додатку для генерації рецептів було виділено основні сутності, з якими працюватиме система. До них, зокрема, належать:

- Користувач (User): містить основну інформацію про користувача додатку — ім’я, електронну адресу, пароль (у захищеному вигляді), набір обраних рецептів, алергени, власні рецепти, дату реєстрації тощо.
- Рецепт (Recipe): об’єднує інформацію про страву: назва, опис, перелік інгредієнтів, інструкції приготування, фото, час готування, категорія, тип, калорійність, автор, а також додаткові атрибути, наприклад, рейтинг чи кількість переглядів.
- Інгредієнт (Ingredient): містить перелік доступних інгредієнтів, кожен з яких має унікальне ім’я, одиниці виміру, список рецептів, у яких він використовується, та інформацію про наявність алергенів.
- Категорія (Category): відображає належність рецептів до певної категорії (наприклад, “Сніданки”, “Вегетаріанські страви” тощо).
- Алерген (Allergen): перелік алергенів, що можуть входити до складу інгредієнтів; ця сутність дає змогу враховувати харчові обмеження користувачів.
- Коментар (Comment): сутність, що пов’язана з рецептом та користувачем, і містить текст відгуку, рейтинг, дату публікації.

Кожна із цих сутностей має власний набір атрибутів та унікальних ідентифікаторів. Усі вони взаємопов’язані між собою. Наприклад, один користувач може додати декілька рецептів, а кожен рецепт містить список інгредієнтів і належить до певної категорії. Відповідно до цього, інгредієнти

можуть входити до складу багатьох рецептів, а категорія — охоплювати багато страв [5].

На рисунку 2.1 наведено ER-діаграму логічної моделі даних мобільного додатку. Діаграма чітко відображає основні сутності системи, їхні атрибути, а також всі ключові зв’язки між об’єктами даних.

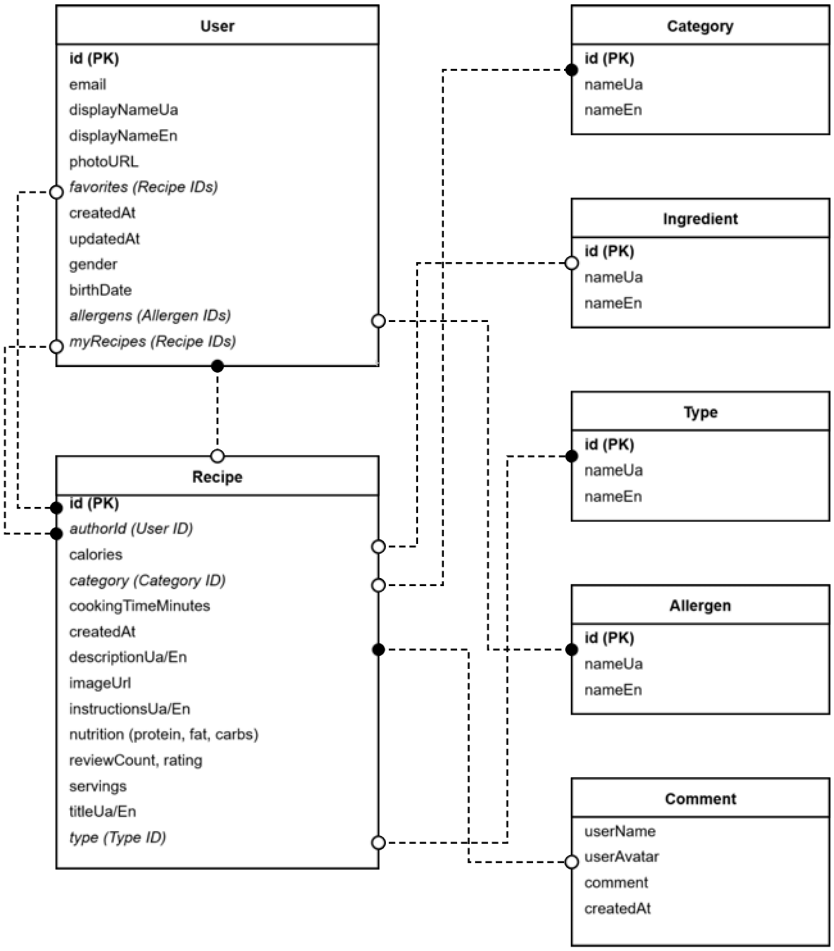

Рис. 2.1. ER-діаграма логічної моделі даних мобільного застосунку для генерації рецептів

Також варто зазначити, що дана схема не лише структурує потік інформації, але й слугує базою для подальшого фізичного моделювання: визначення структури бази даних та створення механізмів ефективного збереження й обробки інформації в системі. На основі цієї ER-діаграми проектується структура бази даних, формується логіка взаємодії між даними,

а також визначаються, які запити необхідні для виконання ключових сценаріїв використання додатку.

Таким чином, представлена у вигляді ER-діаграми логічна модель даних, забезпечує цілісний підхід до організації інформації та є початковим кроком для подальших етапів розробки програмного забезпечення.

2.2 Діаграма класів та кооперацій

В процесі проектування програмної частини інформаційної системи особливої уваги потребує побудова діаграми класів та кооперацій. Ці діаграми є фундаментом для подальшої розробки архітектури застосунку, оскільки дозволяють візуально представити структуру, взаємозв'язки та ролі основних компонентів майбутньої системи.

2.2.1 Діаграма класів. Діаграма класів є одним з ключових інструментів об'єктно-орієнтованого моделювання. Вона відображає основні класи, їх властивості (атрибути) та операції (методи), а також показує відношення між класами – наслідування, асоціації, агрегації чи композиції. Саме за допомогою діаграми класів визначається, які об’єкти існуватимуть у системі, як вони будуть взаємодіяти між собою та які дані і поведінку кожен із них ховає логіку.

У контексті мобільного додатку для генерації рецептів, на діаграмі класів можна виділити такі основні класи:

- User — містить атрибути для зберігання інформації про користувача (id, ім’я, email, список улюблених рецептів, тощо) та методи для керування профілем, аутентифікації, додавання рецептів у “вибране”.
- Recipe — клас, що інкапсулює в собі інформацію про конкретний рецепт (назва, опис, інгредієнти, категорія, інструкції приготування, фото, автор,

рейтинг, тощо), а також відповідні методи (додавання, редагування, перегляд, оцінювання).

- Ingredient — клас, який містить властивості інгредієнта (назва, тип, перелік алергенів, одиниця виміру) та операції, пов’язані з додаванням або пошуком інгредієнтів.
- Category — представлення категорії страв (назва, опис), використовується для групування рецептів.
- Comment — відповідає за зберігання відгуків користувачів, включаючи текст коментаря, рейтинг, автора і дату створення.
- Allergen — клас, що зберігає інформацію про можливі алергени у продуктах.

Між класами встановлюються відповідні зв’язки. Наприклад, клас User має зв’язок “один до багатьох” з класом Recipe, оскільки один користувач може створювати декілька рецептів. Клас Recipe, у свою чергу, пов'язаний з класами: Ingredient, Category та Comment.

На рисунку 2.2 зображено діаграму класів, де відображені основні класи, їх атрибути та зв’язки між ними.

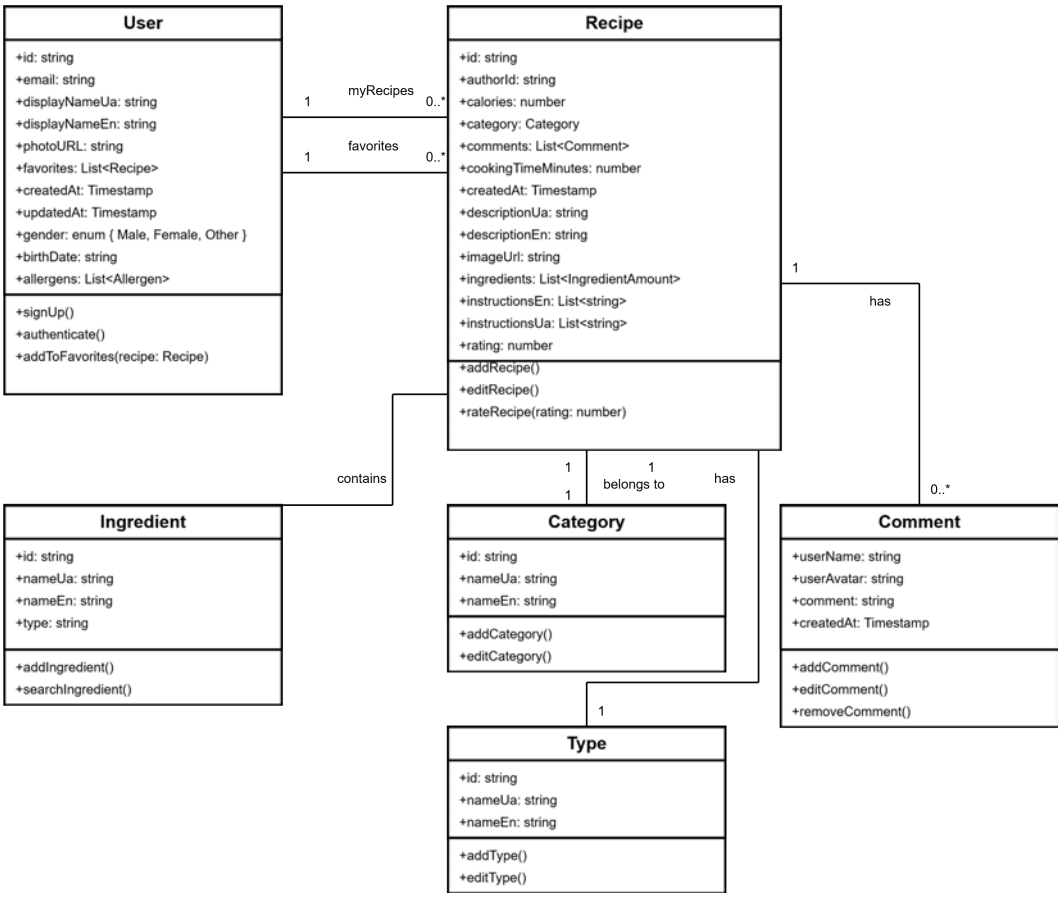

Рис. 2.2. Діаграма класів мобільного застосунку для генерації рецептів

2.2.2 Діаграма кооперацій. Діаграма кооперацій (або колаборацій) – це ще один інструмент, який застосовується для моделювання взаємодії між об’єктами у межах певного сценарію. Вона фокусується не лише на структурних зв’язках, а й на динаміці співпраці між об’єктами у процесі виконання конкретних операцій.

У випадку проектування БД, діаграма кооперацій побудована для ілюстрації, як саме взаємодіють між собою користувач, система та інші об’єкти під час пошуку та генерації рецепту:

1. Користувач ініціює запит на підбір рецепту.
2. Система звертається до класу Ingredient для перевірки наявних інгредієнтів.

3. Відбувається пошук у Recipe з урахуванням вибраних інгредієнтів і категорій.
4. Результат формується та повертається користувачу для перегляду й подальших дій (збереження у “вибране”, коментування, тощо).
5. Такий підхід дозволяє детально простежити шлях виконання ключових сценаріїв і оцінити, наскільки ефективно вибудувана архітектура додатку.

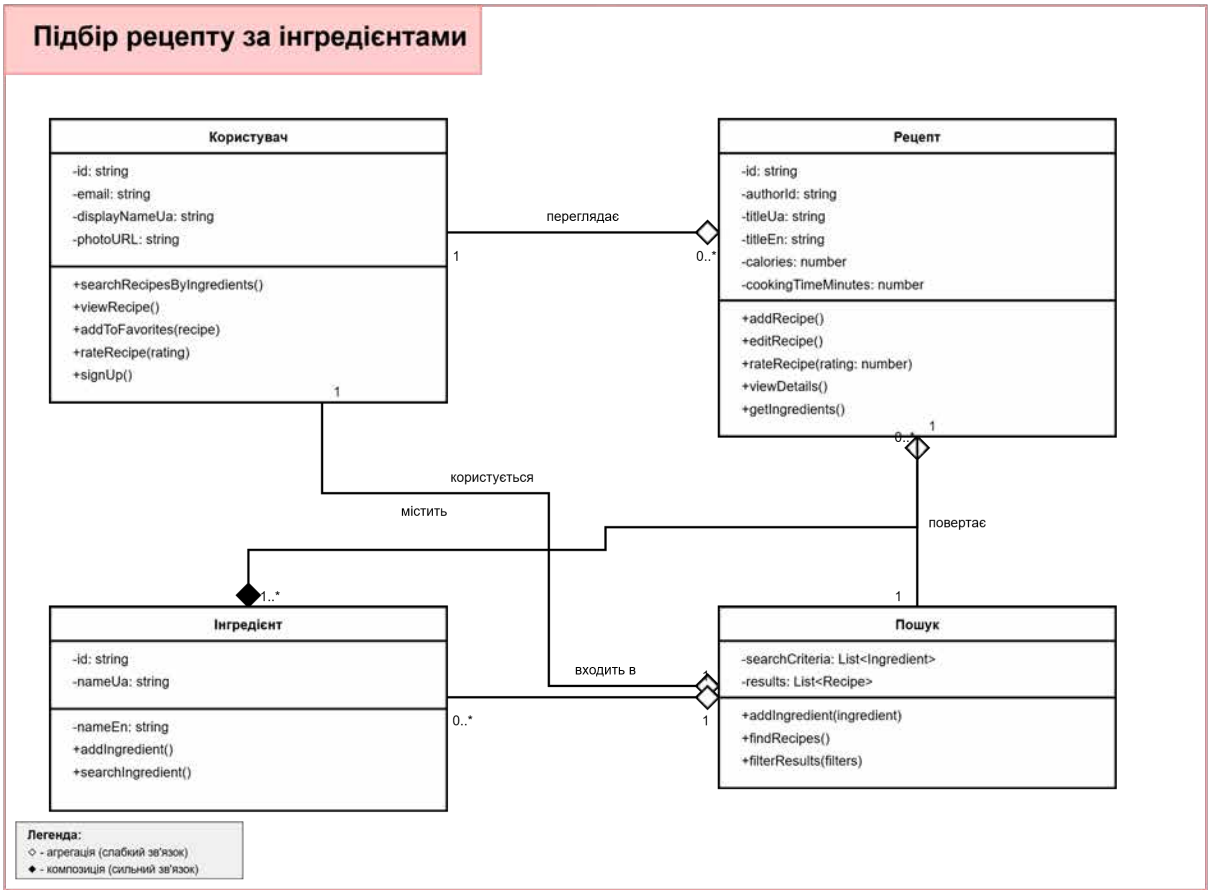

Рис. 2.3. Діаграма кооперацій для сценарію підбору рецепту за інгредієнтами

Отже, використання діаграм класів і кооперацій відіграє значну роль у процесі проектування програмних систем. Завдяки цим діаграмам можна наочно відобразити ключові структурні та поведінкові особливості розроблюваної системи, що дозволяє попередити багато потенційних помилок ще до початку написання коду. Крім того, вони сприяють тому, щоб

усі частини проекту були логічно пов’язані між собою та відповідали загальній концепції розробки.

2.3 Діаграма пакетів

Проектування програмного забезпечення сучасних інформаційних систем неможливо уявити без чіткої структуризації функціональних блоків. Для цього етапу традиційно використовують діаграми пакетів, які дозволяють відобразити поділ складної системи на окремі незалежні модулі, а також зрозуміти характер зв’язків між ними. Такий підхід дає змогу зробити архітектуру більш прозорою, зрозумілою для розробників і зручною для подальшого масштабування.

На рисунку 2.4 представлено діаграму пакетів, яка демонструє організацію ключових логічних частин мобільного застосунку для генерації рецептів. Відповідно до цієї моделі, всю систему умовно поділено на два основні компоненти: модуль користувача (User) та модуль серверної частини (Data Base Server).

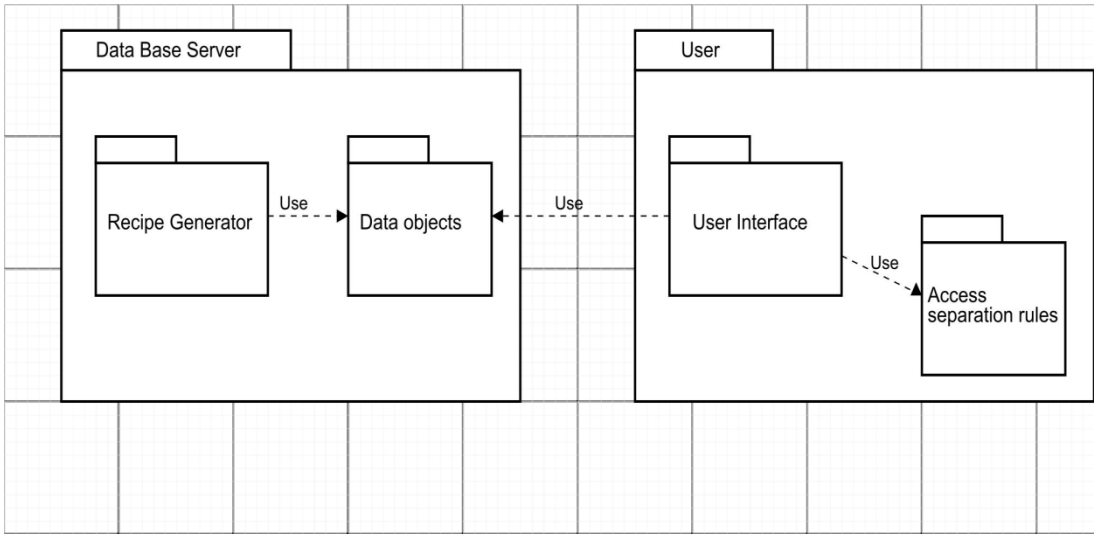

Рис. 2.4. Діаграма пакетів для мобільного додатку з генерації рецептів за введеними інгредієнтами

2.3.1 Опис основних пакетів:

1. User (Користувач) — цей пакет зосереджує всі елементи, пов’язані з інтерфейсом користувача та правилами доступу. Головний блок тут – User Interface (користувацький інтерфейс), через який людина взаємодіє із системою: вводить інгредієнти, переглядає рецепти, здійснює пошук і керує власними налаштуваннями. Окремо виділено підпакет Access separation rules, що відповідає за розмежування прав доступу — наприклад, визначає, які дії дозволено виконувати неавторизованим користувачам, а які — лише зареєстрованим.
2. Data Base Server (Серверна частина) — у цьому пакеті об’єднані основні модулі, відповідальні за роботу із даними та бізнес-логіку.
3. Recipe Generator — центральний компонент, який на основі запиту користувача та введених інгредієнтів формує підбірку відповідних рецептів.
4. Data objects — модуль, що містить об’єкти даних: зберігає інформацію про рецепти, інгредієнти, категорії тощо. Він забезпечує цілісність та актуальність даних у системі.

2.3.2 Характер взаємодії пакетів:

1. Діаграма чітко ілюструє напрямки використання (Use) між основними частинами системи.
2. User Interface користувача взаємодіє з Access separation rules, перевіряючи, чи має користувач право виконати ту чи іншу дію.
3. При роботі з рецептами User Interface використовує можливості Data objects серверної частини, тобто надсилає запити до бази даних.

4. Водночас, Recipe Generator у серверному пакеті також звертається до Data objects для отримання та оновлення інформації про рецепти.

Такий розподіл дозволяє уникати дублювання функцій, забезпечує чітке розмежування відповідальності між частинами системи та значно спрощує як тестування, так і подальший розвиток додатку.

Побудована діаграма пакетів стала основою для подальшої деталізації внутрішньої структури модулів, а також дала змогу ще на ранньому етапі проектування виявити можливі “вузькі місця” у майбутній архітектурі. Відтак така структуризація є надзвичайно важливою для підтримки і масштабування сучасних програмних продуктів.

2.4 Діаграма компонентів

На етапі проектування архітектури мобільного застосунку важливо не лише визначити основні сутності та їхні взаємозв’язки, а й чітко окреслити логічні й фізичні складові майбутньої системи. Для цього використовується діаграма компонентів, яка дозволяє відобразити структуру програмного забезпечення у вигляді незалежних функціональних блоків, що спрощує подальшу розробку, тестування та підтримку додатку.

На рисунку 2.5 представлено діаграму компонентів для мобільного додатку генерації рецептів на основі введених інгредієнтів. Тут система умовно поділена на дві великі частини — клієнтську (Client Side) і серверну (Firebase Cloud).

2.4.1 Клієнтська частина (Client Side) [4]. У клієнтській частині виділено дві основні підсистеми:

1.User Interface (Інтерфейс користувача) — складається з різних екранів, кожен із яких виконує окрему функцію в додатку:

- MainScreen — головний екран із загальним списком рецептів;
- FavoritesScreen — екран обраних страв;

- RecipeScreen — детальний перегляд окремого рецепту;
- SearchScreen — пошук рецептів;
- ProfileScreen — профіль користувача;
- LoginScreen і RegistrationScreen — екрани входу й реєстрації;
- IntroScreen — екран ознайомлення з можливостями застосунку;
- IngredientsScreen — введення інгредієнтів;
- FilterScreen — застосування фільтрів до списку рецептів;
- SettingScreen — налаштування користувача.

Такий поділ дозволяє ізолювати логіку кожної частини інтерфейсу, полегшує підтримку й розширення функціоналу в майбутньому.

- Access & Permissions (Доступ і права користувача):
- Authentication — відповідальний за ідентифікацію користувача в системі;
- AuthorizationRules — визначає рівні доступу, перевіряє дозволи на певні дії в додатку.

2.4.2 Серверна частина (Firebase Cloud) [5]**.** Серверна частина містить основний функціонал роботи з даними й логікою рецептури. Головним блоком тут є FirestoreAPI, що об’єднує низку важливих компонентів:

- IngredientParser — модуль розбору інгредієнтів, введених користувачем;
- RecipeGenerator — відповідає за підбір і генерацію рецептів;
- SaveRecipe / SaveUser — компоненти для збереження рецептів і користувацьких даних;
- FetchSavedRecipes / GetUserFavorites / AddFavoriteRecipe / RemoveFavoriteRecipe — управління списком збережених та обраних страв;
- GetRecipeById / DeleteRecipe — отримання та видалення рецептів;
- GetImageURL — модуль для отримання посилань на зображення;
- SignIn / SignUp / SignOut — операції автентифікації та реєстрації.

Кожен із цих компонентів є ізольованим функціональним блоком, який можна окремо розробляти, тестувати й масштабувати.

2.4.3 Взаємодія компонентів. Діаграма наочно ілюструє комунікацію між клієнтською та серверною частинами через чітко визначені інтерфейси. Користувач взаємодіє із застосунком через набір екранів, система здійснює автентифікацію й перевірку прав доступу, а всі операції з рецептами, улюбленими стравами, збереженням та отриманням даних здійснюються через спеціалізовані компоненти серверної частини.

Такий підхід дозволяє мінімізувати залежності між модулями, підвищити надійність системи та забезпечити її гнучкість при майбутньому розширенні функціоналу чи зміні архітектури.

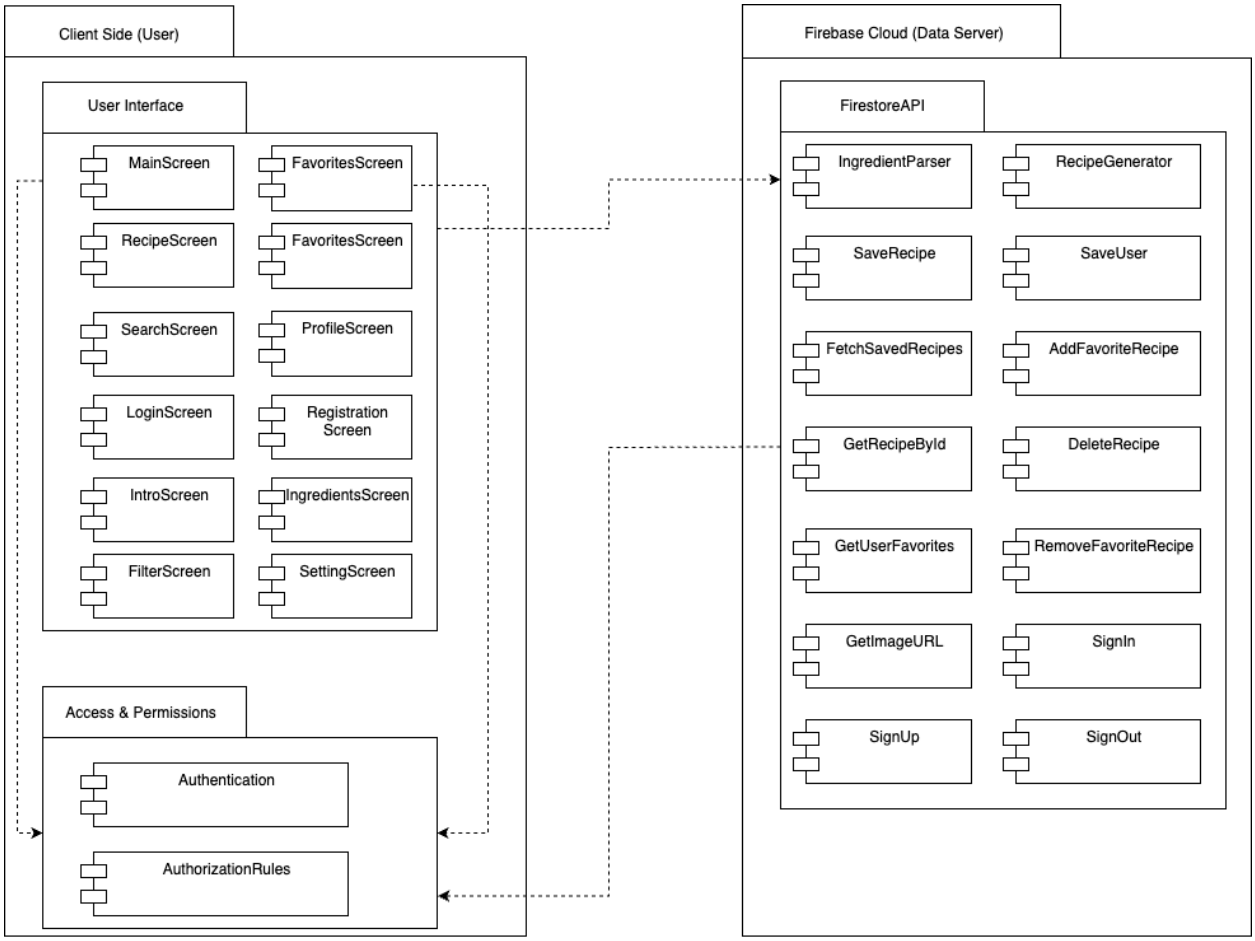

Рис. 2.5. Діаграма компонентів мобільного додатку для генерації рецептів

Загалом, детальна структуризація у вигляді діаграми компонентів стала основою для якісної реалізації програмного продукту, дала змогу чітко визначити функціональні блоки та забезпечити прозору взаємодію між усіма частинами системи.

2.5 Висновки до розділу 2

У другому розділі було проведено комплексне проектування інформаційного та програмного забезпечення майбутньої системи. Першим кроком стало створення логічної моделі даних у вигляді ER-діаграми, що дозволило структуровано подати основні сутності, визначити їх атрибути та взаємозв’язки. Такий підхід забезпечив чітке бачення інформаційної основи додатку та заклав надійний фундамент для розробки ефективної структури зберігання і обробки даних.

Наступним етапом було побудовано діаграми класів і кооперацій, які детально відобразили архітектурні рішення щодо об’єктної моделі. Це дало можливість не лише виявити основні класи системи, а й окреслити способи їхньої взаємодії, що особливо важливо для складних функціональних сценаріїв роботи застосунку.

Важливу роль відіграло і формування діаграми пакетів. Вона дала змогу поділити систему на окремі, логічно обґрунтовані модулі, чітко визначити, за що відповідає кожен із них, і показати, як вони взаємодіють між собою. Такий підхід значно полегшує масштабування програми, додає гнучкості для подальших оновлень і спрощує підтримку на різних етапах розробки.

Окрему увагу було приділено й діаграмі компонентів. Вона допомагає зрозуміти, з яких частин складається клієнтська і серверна логіка, як ці частини розподілені за функціональними блоками, і як між собою співпрацюють. Така деталізація підкреслює важливість чіткої взаємодії між інтерфейсом користувача, механізмами автентифікації, бізнес-логікою та роботою з базою даних.

Отже, можна стверджувати, що продумане і всебічне проектування архітектури стало визначальним фактором для створення надійної, масштабованої та гнучкої системи. Побудовані моделі і діаграми стали дороговказом для подальших етапів розробки й дають змогу уникнути багатьох проблем ще на початкових стадіях роботи. Саме завдяки цьому проектується програмний продукт, здатний відповідати сучасним вимогам і потребам користувачів.

3 Розробка інформаційного та програмного забезпечення

3.1 Система управління інформаційною базою

3.1.1 Загальні положення. Одним із ключових компонентів мобільного додатку для генерації рецептів є система управління базою даних (СУБД), яка забезпечує надійне збереження, швидкий пошук, оновлення та обробку інформації. Для реалізації проекту було обрано документоорієнтовану хмарну базу даних Firestore. Вона оптимально підходить для роботи з динамічними, напівструктурованими даними, що є типовим для мобільних додатків з гнучкою логікою та частими оновленнями.

Firestore також підтримує зберігання даних у форматі документів JSON-подібної структури, що дозволяє ефективно працювати з різноманітними рецептами, інгредієнтами, категоріями та іншими сутностями додатку. Завдяки інтеграції з Firebase, Firestore забезпечує просту автентифікацію користувачів, синхронізацію даних у реальному часі та офлайн-доступ, що суттєво підвищує зручність користування мобільним додатком [8].

3.1.2 Причини вибору Firestore. Вибір Firestore як основної СУБД було обумовлено наступними перевагами:

- Гнучка структура даних. Відсутність жорсткої схеми дозволяє зберігати дані з різною структурою, що особливо важливо при роботі з різноманітними рецептами та інгредієнтами, які можуть мати змінні атрибути.
- Хмарне зберігання та масштабованість. Firestore є повністю керованим сервісом із автоматичним масштабуванням, що дозволяє без додаткових зусиль підтримувати продуктивність при збільшенні кількості користувачів і обсягу даних.
- Реальний час і офлайн режим. Підтримка синхронізації змін у реальному часі та робота в офлайн-режимі покращують користувацький досвід мобільного додатку.

- Безпека. Firestore має гнучку систему правил доступу, що дозволяє детально контролювати права читання і запису залежно від аутентифікації та ролі користувача.
- Інтеграція з іншими сервісами Firebase. Використання Firestore полегшує впровадження аутентифікації, хмарних функцій, аналітики, що скорочує час розробки та підвищує якість продукту.

3.1.3 Порівняння з іншими СУБД.

Таблиця 3.1 — Порівняння з іншими СУБД.

Критерій	**Firestore (NoSQL)**	**PostgreSQL (Реляційна)**	**MongoDB (Документоорієнтована)**
Тип даних	Документи JSON	Таблиці з суворою схемою	Документи JSON
Масштабованість	Автоматична, хмарна	Вертикальна, потребує налаштувань	Висока, горизонтальна
Підтримка транзакцій	Обмежена, підтримка багатодокументних транзакцій	Повна, ACID	Обмежена, з підтримкою ACID у нових версіях
Схема даних	Гнучка, динамічна	Фіксована, нормалізована	Гнучка, динамічна
Підтримка офлайн	Так	Ні	Ні
Інтеграція з Firebase	Повна	Відсутня	Обмежена

Безпека	Гнучкі правила доступу	Рольова модель, тригери	Гнучкі правила доступу
Ідеальне застосування	Мобільні додатки, реальний час, гнучка структура	Класичні бізнес-додатки, складні зв’язки	Веб-додатки, документоорієнтовані системи

3.1.4 Основна структура бази даних у Firestore. У межах проекту основні колекції Firestore містять:

1. users — документи з інформацією про користувачів: email, ім’я, список улюблених рецептів, індивідуальні налаштування.
2. recipes — колекція рецептів із полями назви, опису, інгредієнтів, категорії, типу, часу приготування, фото тощо.
3. ingredients — перелік інгредієнтів, які можуть використовуватись у рецептах, з можливістю позначати алергени.
4. categories — категорії рецептів (супи, салати, десерти тощо).
5. allergens — список алергенів для позначення відповідних інгредієнтів.

Кожна колекція містить документи з відповідними полями, а зв’язки реалізовані за допомогою посилань або масивів ідентифікаторів.

3.2 Розробка інформаційної бази

Проєктування та реалізація інформаційної бази є важливим етапом розробки мобільного додатку для генерації рецептів. Інформаційна база виконує роль централізованого сховища, де зберігаються дані про

користувачів, рецепти, інгредієнти, категорії, алергени та інші сутності, необхідні для повноцінного функціонування додатку.

Розроблена база даних має забезпечувати:

1. Структуроване і цілісне зберігання інформації.
2. Підтримку узгодженості і достовірності даних.
3. Швидкий та ефективний доступ до інформації.
4. Можливість масштабування та розширення без порушення існуючої структури.

3.2.1 Структура інформаційної бази. Інформаційна база побудована з урахуванням особливостей документоорієнтованої СУБД Firestore. На відміну від реляційних баз даних, тут дані зберігаються у вигляді колекцій документів, що дає змогу гнучко адаптувати структуру під різні типи рецептів та варіанти інгредієнтів.

Таблиця 3.2 Основні колекції, що використовуються у базі

Колекція	**Призначення**
users	Інформація про користувачів: профіль, уподобання, алергени, історія пошуку
recipes	Дані про рецепти: назва, опис, інгредієнти, інструкції, категорії, рейтинг
ingredients	Перелік інгредієнтів, включаючи позначення алергенів
categories	Категорії рецептів (супи, салати, десерти тощо)
allergens	Список алергенів для позначення небажаних інгредієнтів
types	Типи рецептів (вегетаріанські, безглютенові тощо)

Кожен документ у колекції містить відповідні поля, наприклад:

- У документі user — email, ім’я, список улюблених рецептів, алергени, дата створення та оновлення.

- У документі recipe — назва, опис, список інгредієнтів (із кількістю), інструкції приготування, час готування, категорія, тип, рейтинг, фото.

3.2.2 Взаємозв'язки між даними У базі даних реалізовані зв'язки між сутностями через посилання або масиви ідентифікаторів. Наприклад:

- Рецепт містить масив ідентифікаторів інгредієнтів та кількістю інгредієнтів.
- Рецепт містить інформацію про ідентифікатор автора рецепту.
- Користувач має список улюблених рецептів за допомогою масиву ідентифікаторів.
- Користувач має список алергенів рецептів за допомогою масиву ідентифікаторів.
- Рецепти має категорію та тип.

Хоча Firestore не підтримує класичні зовнішні ключі як у реляційних базах, підтримка цілісності даних реалізується на рівні логіки додатку та правил безпеки Firebase.

3.2.3 Основні функції роботи з базою даних. Для взаємодії з базою розроблено набір функцій, що реалізують створення, оновлення, видалення і пошук документів у відповідних колекціях.

```
export const createUser = async (
  userId: string,
  userData: User,
): Promise<string> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await setDoc(userRef, {
    ...userData,
    createdAt: Date.now(),
    updatedAt: Date.now(),
  });
  return userId;
};
```

Рис. 3.1. Створення користувача

На Рис. 3.1 зображена функція, яка відповідає за створення нового користувача у базі даних. Вона приймає унікальний ідентифікатор

користувача та об’єкт із його даними, після чого створює відповідний документ у колекції users. Поля createdAt та updatedAt автоматично заповнюються поточним часом для відстеження часу створення і останнього оновлення профілю.

```
export const updateUserProfile = async (
  userId: string,
  data: User,
): Promise<void> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await updateDoc(userRef, {
    ...data,
    updatedAt: Date.now(),
  });
};
```

Рис. 3.2. Оновлення профілю користувача

На Рис. 3.2 зображена функція, яка дозволяє частково оновити інформацію про користувача — наприклад, змінити ім’я, дату народження або фото профілю. Вона приймає ідентифікатор користувача та об’єкт з полями, які необхідно оновити, після чого виконує оновлення відповідного документа.

```
export const createRecipe = async (recipe: Recipe): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.RECIPES);
  const docRef = await addDoc(colRef, {
    ...recipe,
    reviewCount: 0,
    createdAt: Date.now(),
    updatedAt: Date.now(),
  });
  return docRef.id;
};
```

Рис. 3.3. Створення рецепту

На Рис. 3.3 зображена функція, яка створює новий рецепт у колекції recipes. Вона приймає об’єкт рецепту без полів ідентифікатора, рейтингу, кількості відгуків та часових позначок, які автоматично встановлюються при

додаванні. Таким чином, кожен новий рецепт починається з кількістю переглядів 0 і без відгуків.

3.2.4 Пошук та фільтрація даних. Окрему увагу приділено реалізації механізмів пошуку та фільтрації рецептів за різними критеріями — інгредієнтами, алергенами, категоріями та типами. Firestore має обмеження на складність запитів, тому для складніших фільтрацій частина логіки виконується на клієнтській стороні.

```
export const searchRecipes = async (searchTerm: string): Promise<Recipe[]> => {
  if (!searchTerm?.trim()) {
    return getAllRecipes();
  }

  const term = searchTerm.trim();
  const results: Recipe[] = [];
  const processedIds = new Set<string>();
  const recipesCollection = collection(FIREBASE_DB, COLLECTIONS.RECIPES);

  const searchFields = [
    FIELDS.TITLE_EN,
    FIELDS.TITLE_UA,
    FIELDS.DESCRIPTION_EN,
    FIELDS.DESCRIPTION_UA,
  ];

  const searchPromises = searchFields.map(async field => {
    try {
      const q = query(
        recipesCollection,
        where(field, '>=', term),
        where(field, '<=', term + '\uf8ff'),
      );
      return await getDocs(q);
    } catch (error) {
      console.error(`Error searching ${field}:`, error);
      return null;
    }
  });

  const snapshots = await Promise.all(searchPromises);

  snapshots.forEach(snap => {
    snap?.docs.forEach(doc => {
      if (!processedIds.has(doc.id)) {
        results.push({id: doc.id, ...doc.data()} as Recipe);
        processedIds.add(doc.id);
      }
    });
  });

  return results;
};
```

Рис. 3.4. Пошук рецепту

На Рис. 3.3 зображена функція, яка реалізує складну фільтрацію рецептів за кількома критеріями: інгредієнтами, алергенами, категоріями та типами. Через обмеження Firestore, частина фільтрації виконується на клієнтській стороні, що дозволяє більш гнучко комбінувати умови пошуку та забезпечує точний результат.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

Ефективна розробка мобільного додатку значною мірою залежить від правильно підібраного набору технологій і інструментів. Враховуючи особливості предметної області, вимоги до функціональності, продуктивності, а також часові й ресурсні обмеження, було здійснено ретельний вибір стеку технологій, що забезпечить стабільну, масштабовану та зручну у підтримці систему.

3.3.1 Мова програмування і фреймворк. Для створення мобільного додатку мовою програмування було обрано JavaScript із використанням TypeScript, що дало змогу впровадити сувору типізацію у проект. Це рішення допомогло ще на етапі написання коду уникати багатьох помилок і значно спростило подальшу підтримку, особливо коли кодова база стала досить великою [6].

В якості основного фреймворку для розробки застосунку використано React Native. Це сучасна технологія, яка дозволяє розробляти кросплатформені мобільні додатки, тобто створювати одну кодову базу для Android і iOS одночасно. Завдяки React Native можна отримати доступ до нативних можливостей пристрою і забезпечити користувачам швидкий та зручний інтерфейс користувача [7] .

Переваги React Native для проєкту:

- Швидкий цикл розробки завдяки гарячому перезавантаженню (Hot Reloading).

- Велика екосистема бібліотек та інструментів.
- Можливість підключення нативних модулів, якщо потрібна специфічна функціональність.
- Підтримка спільноти та регулярні оновлення.
- Компіляція в нативний код, забезпечуючи продуктивність, близьку до повністю нативних додатків. Доступ до нативних API та компонентів платформи.
- Можливість розробляти додаток одразу як для Android та і для IOS.

3.3.2 Хмарні сервіси та база даних. В якості основного сховища даних було обрано Firebase Firestore. Це сучасна хмарна документоорієнтована БД з підтримкою роботи у реальному часі і офлайн-доступу [8].

Причини вибору Firestore:

- Гнучкість схеми даних дозволяє зберігати рецепти, інгредієнти та інші сутності з різною структурою.
- Автоматичне масштабування і підтримка великої кількості користувачів.
- Інтеграція з Firebase Authentication, що спрощує реалізацію безпечної авторизації.
- Підтримка правил безпеки, які контролюють доступ до даних на рівні документів.
- Легка інтеграція з React Native через офіційний SDK.

Це дає можливість швидко створювати й масштабувати додаток без потреби самостійно налаштовувати серверну інфраструктуру.

3.3.3 Навігація та взаємодія з користувачем. Для організації навігації між екранами застосовано React Navigation, оскільки це найбільш популярна бібліотеку навігації для React Native. Вона підтримує стекову, табову та drawer-навігацію, а також глибокі посилання (deep linking), що дозволяє зручно будувати багаторівневу структуру додатку.

Для побудови форм та їх валідації обрана бібліотека React Hook Form, що має простий API, високу продуктивність та підтримує інтеграцію з TypeScript.

3.3.4 Робота з мережею та API. Для взаємодії з віддаленими сервісами використовується Axios — популярна бібліотека для HTTP-запитів з підтримкою промісів, перехоплювачів запитів та відповіді, а також зручним налаштуванням.

Хоча Firestore використовується як основний бекенд, Axios дає змогу у майбутньому підключити додаткові API або мікросервіси.

3.3.5 Інструменти для забезпечення якості коду. Для підтримки якості коду застосовуються:

- ESLint — статичний аналізатор коду, що виявляє помилки та порушення стилю.
- Prettier — автоматичне форматування коду, яке забезпечує єдиний стиль оформлення у всьому проєкті.

Ці інструменти інтегруються у середовище розробки і CI/CD процеси, що сприяє підтримці чистоти і зрозумілості коду.

3.3.6 Середовище розробки та автоматизація. Розробка проекту проходила у середовищі Visual Studio Code. Він має широкий набір розширень для роботи з React Native, TypeScript, Git, Firebase та іншими технологіями.

Для збірки і тестування використовується стандартний Metro bundler, а також додаткові інструменти для емуляції пристроїв та відлагодження.

3.3.7 Переваги вибраного стеку Вибраний інструментарій дає змогу:

- Розробляти швидко і зручно, завдяки кросплатформеності React Native.
- Забезпечувати надійну і гнучку роботу з даними через Firestore.

- Легко масштабувати та розширювати додаток.
- Підтримувати чистий, типізований та безпечний код.
- Застосовувати сучасні практики управління станом та навігації.
- Забезпечувати якість коду за допомогою автоматичних інструментів аналізу і форматування.

3.4 Алгоритмізація та програмування програмних модулів

3.4.1 Загальні відомості. Алгоритмізація — це процес розробки чітких, крок за кроком інструкцій для розв’язання поставлених задач. Вона передбачає побудову логіки функцій і модулів, що реалізують необхідний функціонал, у вигляді послідовності дій.

Для того щоб процес розробки був максимально прозорим і зрозумілим, було використано блок-схеми. Вони наочно показують, які саме операції виконує система, де передбачені умови, цикли чи розгалуження. Такі схеми суттєво спрощують розробку і тестування — адже з їхньою допомогою можна легко перевірити, як працює той чи інший модуль, а за потреби можна швидко внести зміни.

3.4.2 Алгоритм аутентифікації користувача. На основі наданих блок-схем розглянемо алгоритм входу користувача в систему.

1. Користувач вводить свій email та пароль у відповідних полях на екрані входу.
2. Дані проходять валідацію з використанням бібліотек Formik та Yup, що забезпечують коректність введених даних.
3. Якщо валідація не пройдена, користувачу показується відповідна помилка, і процес завершується.
4. Якщо валідація успішна, виконується виклик функції signInWithEmailAndPassword Firebase Authentication для входу в систему.
5. У разі неуспішного входу відображається повідомлення про помилку.
6. При успішному вході запускається глобальний слухач onAuthStateChanged, який встановлює інформацію про користувача у глобальний стан додатку (контекст або стан Redux).
7. Далі відбувається перевірка наявності userId у локальному сховищі AsyncStorage.
8. Якщо ідентифікатор відсутній або відрізняється від поточного, встановлюється прапорець isFirstLogin = true, інакше — false.

9. Після цього виконується навігація в залежності від статусу авторизації користувача:
10. Якщо користувач авторизований — переходить до основних екранів додатку.
11. Якщо ні — відображаються екрани входу/реєстрації.

Цей алгоритм забезпечує плавний та безпечний вхід користувача, а також підтримує логіку першого входу, яка може використовуватись для показу вітальних чи навчальних екранів.

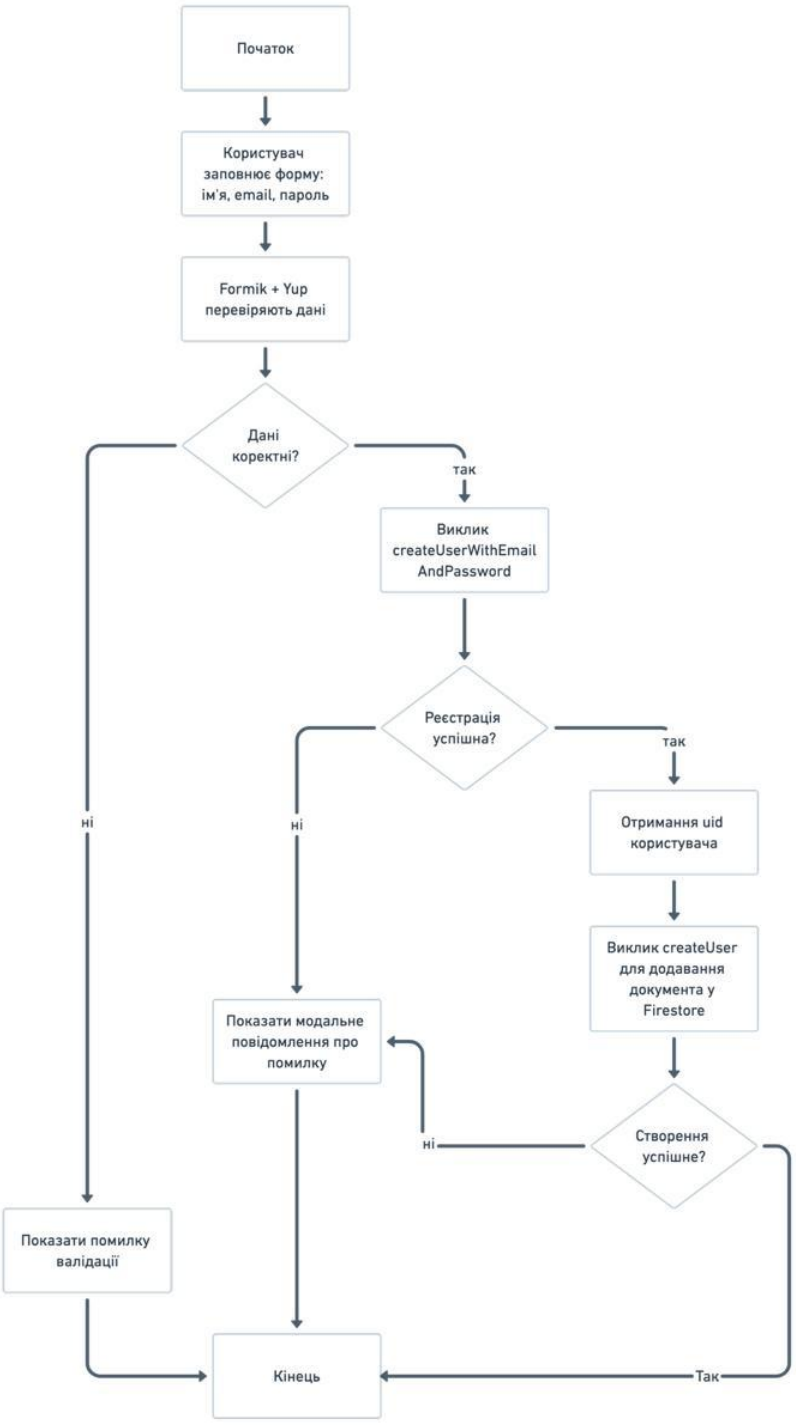

Рис. 3.5. Блок-схема аутентифікації користувача

3.4.3. Реєстрація користувача

1. Користувач заповнює форму реєстрації з основними полями: ім’я, email та пароль. Введені дані проходять валідацію за допомогою Formik та Yup.
2. Якщо дані некоректні, користувач бачить повідомлення про помилку валідації.
3. При коректних даних виконується виклик Firebase Auth методу createUserWithEmailAndPassword.
4. У разі успішної реєстрації отримується унікальний ідентифікатор користувача (UID). Після цього створюється відповідний документ користувача у Firestore через функцію createUser.
5. Якщо збереження у Firestore не відбулося успішно, користувачеві показується повідомлення про помилку.
6. Після успішного створення профілю процес реєстрації завершується, і користувач може перейти до роботи з додатком.

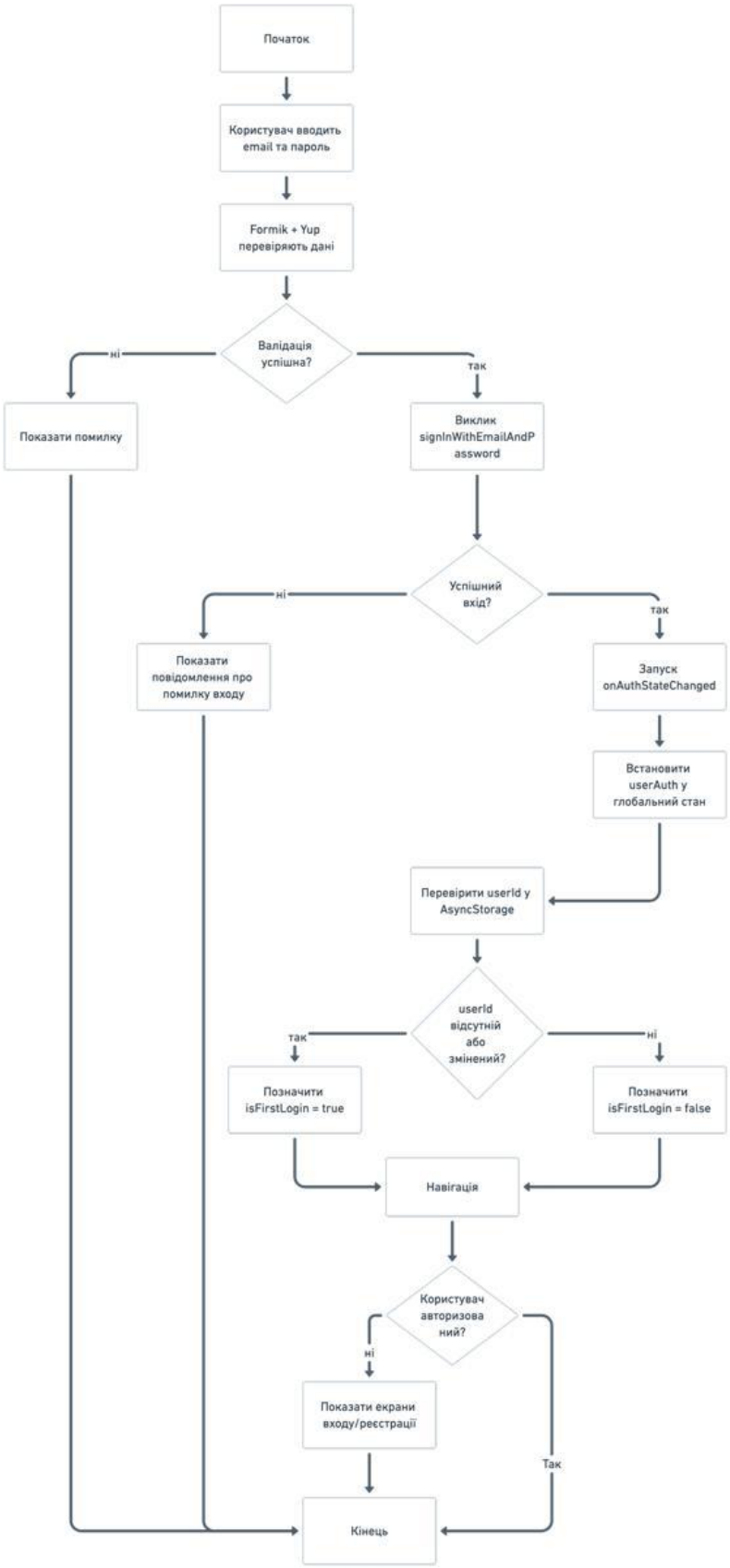

Рис. 3.6. Блок-схема реєстрації користувача

3.4.4 Алгоритм генерації рецепту.

1. Користувач ініціює процес створення рецепту, натискаючи кнопку «Створити рецепт». Відкривається сторінка генерації рецепту GeneratedRecipe з обраними фільтрами.
2. Далі виконується виклик API генерації рецепту generateRecipe з параметрами, що відповідають заданим фільтрам.
3. Отриманий рецепт із часовими мітками конвертується у формат JavaScript Date, що дозволяє коректно працювати з датами.
4. Відображаються деталі рецепту: назва, інгредієнти, кроки приготування, калорійність та інші харчові показники.
5. Користувач може змінювати кількість порцій, що автоматично коригує кількість інгредієнтів і відповідні харчові дані.
6. При збереженні рецепту виконується перевірка кожного інгредієнта через функцію getAllIngredients на наявність у базі даних:
7. Якщо інгредієнт знайдений — використовується існуючий.
8. Якщо ні — створюється новий через функцію createIngredient.
9. Аналогічним чином відбувається обробка категорій та типів рецепту.
10. Після формування повного об’єкта рецепту він зберігається у Firestore через функцію createRecipe.
11. Якщо збереження відбулося успішно, користувач переходить на сторінку перегляду деталей новоствореного рецепта. В іншому випадку відображається повідомлення про помилку.

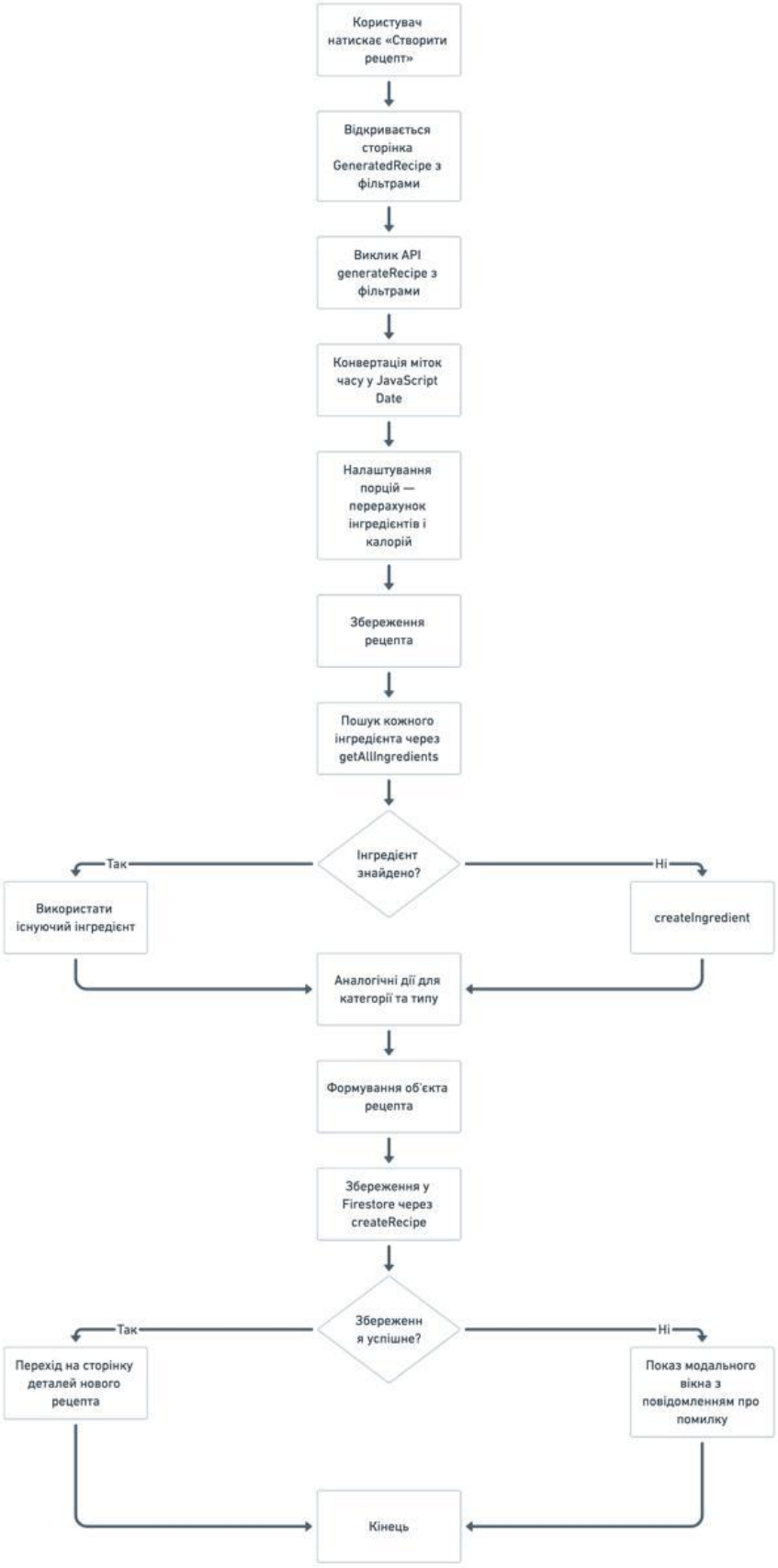

Рис. 3.7. Блок-схема генерація рецепту

Ці алгоритми лягли в основу реалізації відповідних програмних модулів додатку, які забезпечують коректну, зручну і безпечну роботу для кінцевого користувача. Послідовність дій, перевірок і взаємодій із сервісами детально опрацьована для мінімізації помилок і забезпечення позитивного користувацького досвіду.

3.4.5 Модульність та структура програмного додатку. Для забезпечення зручності розробки, підтримки і масштабування мобільного додатку була використана модульна архітектура. Код розбитий на логічні компоненти, які виконують окремі функції, що спрощує навігацію по проекту та полегшує командну роботу.

На наведеному нижче зображені з редактора коду показана структура основних папок проєкту:

1. services — містить допоміжні сервіси та API-шари для роботи з базою даних, константами, локалізацією (i18n), темами та типами. Це ядро бізнес-логіки, що відокремлене від UI.
2. src — основна папка з вихідним кодом додатку, де розміщені:
3. assets — ресурси (зображення, іконки, шрифти)
4. components — повторно використовувані UI-компоненти (кнопки, форми, елементи списку)
5. config — конфігурації додатку, налаштування сервісів
6. layout — компоненти, що формують загальний каркас UI (хедера, футера, обгортки)
7. navigation — реалізація маршрутизації і навігації між екранами
8. screens — екранні компоненти — логічні сторінки додатку
9. types — типи TypeScript для уніфікації структури даних
10. utils — утилітні функції і хелпери, що використовуються в різних місцях

Такий розподіл дозволяє легко знаходити потрібний функціонал, розділяти відповідальність між модулями і підвищує гнучкість при додаванні нових функцій або зміні існуючих.

Кожен модуль і компонент має чітко визначену роль, що сприяє більш чистій і зрозумілій архітектурі. Наприклад, всі виклики до серверних API сконцентровані у папці services/apis, що полегшує тестування і заміну бекенд-сервісів.

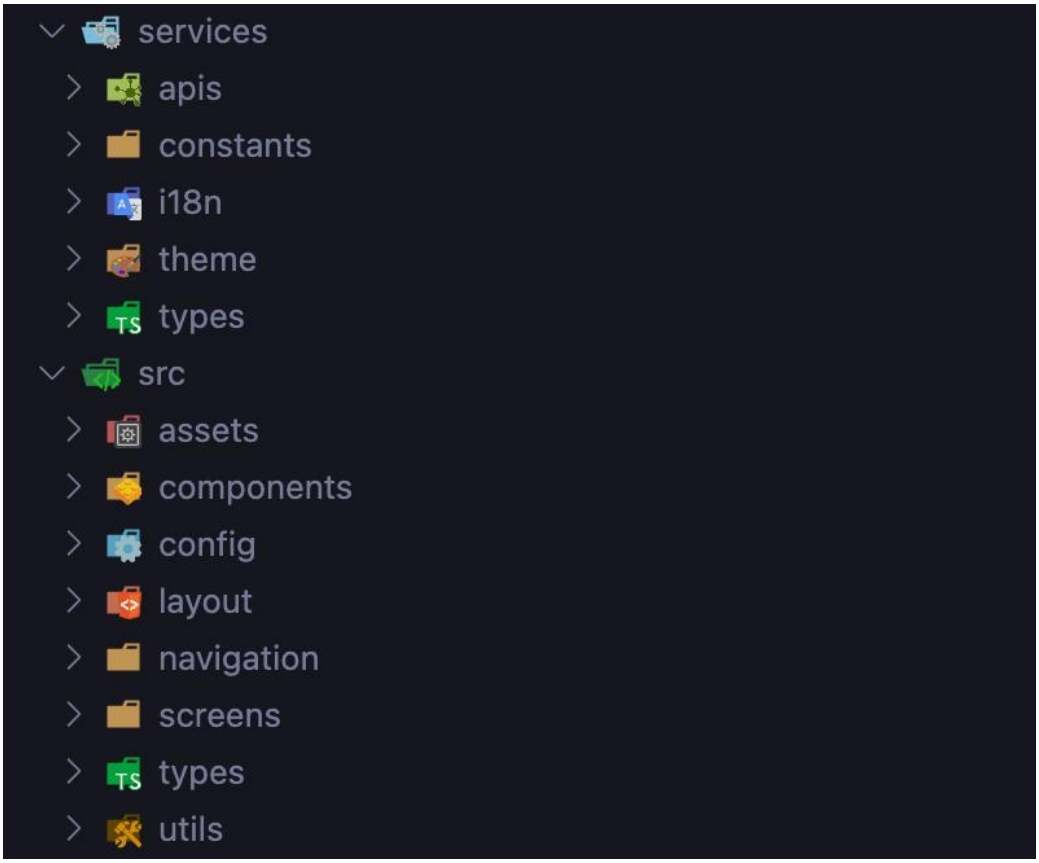

Рис. 3.8. Структура додатку

3.5 Висновки до розділу 3

У третьому розділі було розглянуто основні етапи розробки програмної системи для мобільного додатку генерації рецептів на основі введених інгредієнтів. Визначено та обґрунтовано вибір системи управління базою даних — хмарної документоорієнтованої Firestore, яка забезпечує гнучкість структури даних, масштабованість і високу швидкість обробки інформації.

Розроблена інформаційна база даних спроектована з урахуванням предметної області додатку та підтримує всі необхідні колекції й зв’язки між сутностями. Наведені приклади основних функцій для роботи з базою

демонструють надійність і ефективність збереження, оновлення та пошуку даних.

Вибір інструментарію для створення прикладного програмного забезпечення базується на сучасних і перевірених технологіях — React Native, TypeScript, Redux Toolkit, Firebase SDK, які дозволяють забезпечити кросплатформенність, типізованість і продуктивність розробки.

Алгоритмізація та програмування основних модулів додатку, представлені у вигляді покрокових алгоритмів та блок-схем, чітко відображають логіку функціонування ключових користувацьких сценаріїв — реєстрації, входу, налаштування профілю, пошуку та створення рецептів. Це створює міцну основу для подальшої реалізації та тестування системи.

Таким чином, у розділі 3 виконано всебічний аналіз і опис архітектури програмної системи, вибору технологій і розробки основних алгоритмів, що забезпечують коректну і стабільну роботу мобільного додатку.

4 Рекомендації щодо впровадження та експлуатації системи

4.1 Тестування системи. Тестування є невід'ємною частиною розробки програмного забезпечення, що дозволяє забезпечити якість, надійність та відповідність системи вимогам. Метою тестування є виявлення та усунення помилок на ранніх етапах розробки, а також підтвердження коректності роботи функціональних і нефункціональних компонентів додатку **[3].**

4.1.1 Види тестування. У процесі розробки застосовується кілька видів тестування:

- Автоматизоване тестування — виконання тестів за допомогою спеціальних інструментів та фреймворків, що дозволяє швидко та систематично перевіряти функціональність додатку.
- Ручне тестування — перевірка функціоналу користувачем або тестувальником без використання автоматичних скриптів, що дає змогу оцінити поведінку системи в реальних умовах.

Обидва підходи доповнюють один одного і застосовуються для досягнення максимального покриття тестування [10].

4.1.2 Цілі тестування. Основні цілі тестування системи:

1. Перевірка коректності роботи окремих модулів та всього додатку.
2. Виявлення помилок та несправностей на різних рівнях.
3. Підтвердження відповідності функціоналу вимогам.
4. Забезпечення стабільної роботи при зміні коду (рефакторинг, додавання нових функцій).
5. Підвищення довіри користувачів до якості програмного продукту.

4.1.3 Автоматичне тестування за допомогою Jest. Для автоматизації тестування у проекті використовується фреймворк Jest, який добре підходить для тестування JavaScript/TypeScript проектів, зокрема API-інтерфейсів. Jest дозволяє писати Unit-тести, створювати мок-об’єкти, перевіряти асинхронні функції та забезпечує зручний інтерфейс для запуску та моніторингу тестів.

У проекті реалізовано тести для сервісних функцій, які працюють із базою даних Firestore. Для тестування функцій взаємодії з API застосовується мокування функцій Firebase, що дозволяє імітувати виклики без реального підключення до бази даних.

Таблиця 4.1 Сценарії тестування

Тестова умова	**Очікуваний результат**
Виклик createUser з валідним userId	Повертає переданий userId
Виклик createCategory з даними категорії	Повертає id нового запису (мок test-id)
Виклик getAllCategories при відсутності категорій	Повертає порожній масив
Виклик getCategoryById з існуючим ID	Повертає об’єкт категорії з коректними даними
Оновлення категорії через updateCategory	Виконується без помилок
Виклик createRecipe з валідними даними рецепту	Повертає id нового рецепту (мок test-id)
Видалення рецепту через deleteRecipe	Виконується без помилок

Додавання рецепту в улюблені користувача	Виконується без помилок

```
describe('Firebase Recipe APIs', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('should create a recipe and return its ID', async () => {
    const id = await apis.createRecipe(recipeData);
    expect(id).toBe('test-id');
  });

  it('should get a recipe by ID', async () => {
    const recipe = await apis.getRecipeById('test-id');
    expect(recipe).toBeDefined();
    expect(recipe?.id).toBe('test-id');
  });

  it('should update a recipe', async () => {
    await apis.updateRecipe('test-id', {titleEn: 'Updated Recipe'});
    // Just verify it doesn't throw an error
    expect(true).toBe(true);
  });

  it('should delete a recipe', async () => {
    await apis.deleteRecipe('test-id');
    // Just verify it doesn't throw an error
    expect(true).toBe(true);
  });
});

describe('Firebase User APIs', () => {
  it('should add a recipe to favorites', async () => {
    await apis.addToFavorites('user-123', 'recipe-123');
    // Just verify it doesn't throw an error
    expect(true).toBe(true);
  });

  it('should remove a recipe from favorites', async () => {
    await apis.removeFromFavorites('user-123', 'recipe-123');
    // Just verify it doesn't throw an error
    expect(true).toBe(true);
  });

  it('should update user allergens', async () => {
    await apis.updateUserAllergens('user-123', ['allergen-1', 'allergen-2']);
    // Just verify it doesn't throw an error
    expect(true).toBe(true);
  });
});
```

Рис. 4.1 приклад коду тесту для API

Схожим чином пишуться й інші тести. Ці тести забезпечують перевірку основних функцій API, що гарантує коректну роботу логіки додатку.

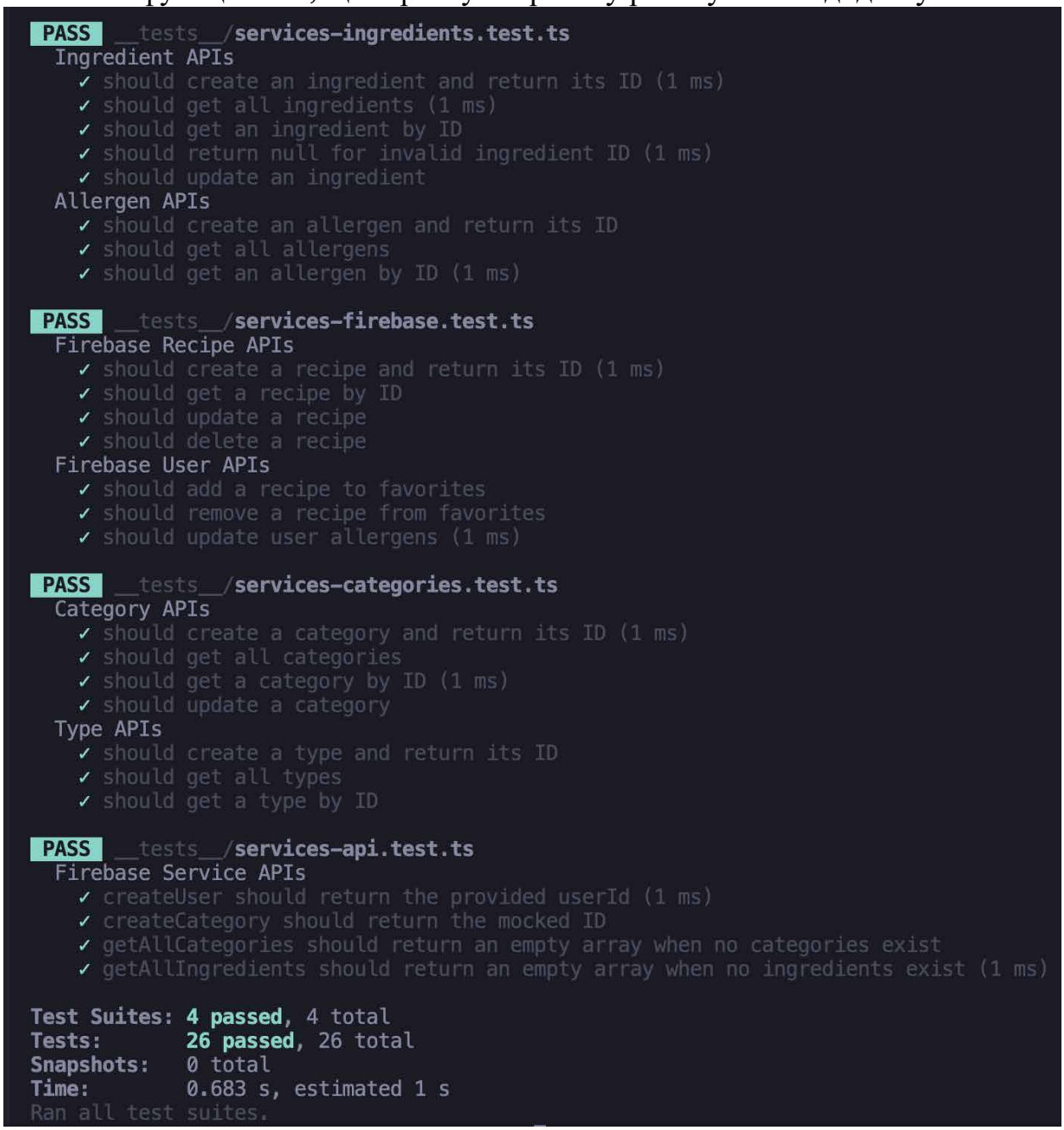

Рис. 4.2 Результати всіх тестів

4.1.4 Ручне тестування. Ручне тестування проводиться для перевірки роботи інтерфейсу користувача, а також функціоналу, який важко або неефективно автоматизувати. Ручне тестування допомагає виявити проблеми з юзабіліті, логікою бізнес-процесів та незаплановані сценарії використання.

Для забезпечення високої якості роботи мобільного додатку було проведено ручне тестування основних функціональних модулів. Це дозволило перевірити коректність роботи інтерфейсу, логіки бізнес-процесів, а також загальну стабільність системи в реальних умовах використання.

Основні напрямки ручного тестування:

- Аутентифікація користувача: перевірка реєстрації, входу, виходу, відновлення пароля та зміни профілю.
- Введення інгредієнтів: тестування додавання, видалення та редагування інгредієнтів у списку для генерації рецептів.
- Генерація рецептів: перевірка роботи алгоритму генерації рецепту на основі введених інгредієнтів, коректність відображення результатів.
- Відображення деталей рецепту: перевірка коректності виведення інформації про рецепт — назва, опис, інгредієнти, кроки приготування, час приготування, фото.
- Функції збереження: тестування можливості збереження рецептів до улюблених, а також видалення зі списку.
- Пошук та фільтрація: перевірка функціоналу пошуку рецептів за ключовими словами і застосування фільтрів за категоріями, типами страв, алергенами.
- Перевірка навігації: коректність переходів між екранами, відсутність помилок при переходах.

Ручне тестування здійснювалося за допомогою смартфона із встановленим додатком, у реальних умовах роботи. Всі дії виконувалися послідовно, з фіксацією результатів та виявлених неточностей або помилок.

Особлива увага приділялась сценаріям, які можуть бути складними для автоматизації, таким як зміна параметрів рецепту в процесі перегляду, налаштування порцій, а також взаємодія з інтерфейсом.

4.1.5 Результати тестування. Під час тестування було підтверджено стабільну роботу основних функцій додатку, коректне відображення інформації та коректну обробку користувацьких дій. Виявлені незначні помилки були оперативно виправлені в процесі розробки.

4.2 Вимоги до апаратного та програмного забезпечення

Для успішної експлуатації мобільного додатку генерації рецептів необхідно чітко визначити вимоги до апаратного та програмного забезпечення як користувача, так і розробника. Це дозволяє забезпечити оптимальну продуктивність, стабільність та якість роботи системи, уникнути проблем із сумісністю та підвищити загальний користувацький досвід.

4.2.1 Вимоги до апаратного забезпечення користувача. Мобільний додаток орієнтований на широке коло користувачів, тому підтримує популярні моделі смартфонів з операційними системами Android і iOS. Для забезпечення плавної роботи інтерфейсу, швидкої обробки запитів до серверу, а також стабільного відображення мультимедійного контенту, пристрій користувача має відповідати певним технічним характеристикам.

Низька продуктивність пристрою може призводити до затримок у відображенні інтерфейсу, помилок при обробці даних, а також до зниження якості користувацького досвіду. Тому рекомендовано використовувати пристрої з не менш ніж 2 ГБ оперативної пам’яті, а також з процесорами не нижче двоядерних із частотою 1.5 ГГц. Це забезпечить оптимальну роботу додатку навіть при одночасній роботі з кількома екранами або при генерації складних рецептів.

Таблиця 4.2 вимоги до апаратного забезпечення

Категорія	Мінімальні вимоги	Рекомендовані вимоги
Процесор	Двоядерний процесор 1.5 ГГц	Чотириядерний процесор 2.0 ГГц і вище
Оперативна пам'ять (RAM)	2 ГБ	4 ГБ і більше
Внутрішня пам'ять	100 МБ вільного місця	200 МБ і більше
Екран	Роздільна здатність від 720x1280 пікселів	Full HD (1080x1920 пікселів) або більше
З'єднання з інтернетом	3G або Wi-Fi	4G/5G або стабільний Wi-Fi

4.2.2 Вимоги до програмного забезпечення користувача. Для коректної роботи додатку важливо, щоб операційна система мобільного пристрою підтримувала сучасні функції, необхідні для запуску React Native додатків, а також сумісність із Firebase та іншими сервісами.

Мінімальною підтримуваною версією для Android є 7.0, що охоплює більшість активних пристроїв, а для iOS – 12 версія. Однак для кращої продуктивності, безпеки і підтримки нових функцій додатка рекомендовано використовувати для Android версію 11 та вище, а для iOS 14 версію і вище.

Також користувачам необхідно мати стабільне підключення до інтернету, оскільки багато операцій (запити до сервера, завантаження рецептів, оновлення бази даних) відбуваються онлайн.

Таблиця 4.3 вимоги до платформи

Платформа	**Мінімальна версія ОС**	**Рекомендована версія ОС**
Android	Android 7.0 (Nougat)	Android 11 і новіші
iOS	iOS 12	iOS 14 і новіші

Крім того, у користувачів має бути встановлено браузер або середовище, яке підтримує сучасні стандарти безпеки та сумісність з веб-компонентами додатку (наприклад, для відкриття зовнішніх посилань чи інтеграцій).

4.2.3 Вимоги до апаратного та програмного забезпечення для розробки. Для розробки, тестування і підтримки мобільного додатку потрібно відповідне обладнання і програмне забезпечення, які дозволяють ефективно створювати, налагоджувати та запускати проект.

Загальні вимоги до робочого середовища розробника включають сучасну операційну систему (Windows 10 і вище, macOS Catalina і вище), достатньо потужний процесор (рекомендовано не нижче чотириядерного з тактовою частотою від 2 ГГц), і оперативну пам’ять не менше 8 ГБ (краще — 16 ГБ) для комфортної роботи з емуляторами, редакторами коду, системами контролю версій і іншими інструментами.

Для запуску мобільних емуляторів (Android Studio Emulator, Xcode Simulator) важливо мати достатньо ресурсів, зокрема SSD-диск для швидкого доступу до файлів проекту [7].

Необхідні інструменти:

1. Node.js для запуску JavaScript оточення
2. React Native CLI для роботи з мобільним додатком

3. Firebase CLI для роботи з серверною частиною та хмарними функціями
4. Jest для автоматичного тестування
5. IDE (Visual Studio Code або WebStorm) для зручного кодування

Таблиця 4.4 вимоги для середовища розробки.

Компонент	**Мінімальні вимоги**	**Рекомендовані вимоги**
Операційна система	Windows 10 / macOS Catalina або новіша	macOS Big Sur / Windows 11
Процесор	Двоядерний процесор Intel або AMD	Чотириядерний або вище, Apple M1/M2
Оперативна пам’ять	8 ГБ	16 ГБ або більше
Дисковий простір	Мінімум 20 ГБ вільного місця	SSD-диск з 50 ГБ вільного простору
Мобільний емулятор	Android Studio Emulator або Xcode Simulator	Емулятор останніх версій Android/iOS
Інструменти розробки	Node.js, React Native CLI, Firebase CLI, Jest	IDE (VSCode, WebStorm), інструменти CI/CD

4.2.4 Вимоги до мережевого з’єднання. Оскільки мобільний додаток використовує хмарні сервіси Firebase для збереження та синхронізації даних, генерації рецептів та аутентифікації користувачів, стабільність і швидкість інтернет-з’єднання є ключовими факторами для комфортного використання.

Погане або нестабільне з’єднання може призвести до тривалого завантаження даних, помилок під час генерації рецептів, а також проблем із синхронізацією локальних і хмарних даних.

Тому рекомендовано мати інтернет-з’єднання зі швидкістю не менше 1 Мбіт/с (мінімальні умови) та низькою затримкою (пінг не більше 150 мс). Оптимальним є підключення через 4G/5G або стабільний Wi-Fi з високою швидкістю передачі даних та низькою затримкою.

Таблиця 4.5 вимоги домережі.

Параметр	**Мінімальні вимоги**	**Рекомендовані параметри**
Тип з’єднання	3G, Wi-Fi	4G/5G, стабільний Wi-Fi
Швидкість інтернету	Від 1 Мбіт/с	Від 5 Мбіт/с і більше
Затримка мережі (ping)	Не більше 150 мс	Не більше 50 мс

4.3 Склад інсталяційного пакету

Інсталяційний пакет є основним носієм програмного забезпечення, який містить усі необхідні компоненти для встановлення, запуску та безперебійної роботи мобільного додатку генерації рецептів на основі введених інгредієнтів. Правильне формування і комплектування пакету є важливим етапом підготовки до впровадження системи, адже він забезпечує зручність розгортання для кінцевих користувачів і стабільність роботи додатку у різних умовах.

4.3.1 Основні компоненти інсталяційного пакету. Інсталяційний пакет містить такі основні складові:

1.Виконавчий файл додатку.

Для платформи Android — це файл у форматі APK (Android Package), який містить всі компоненти додатку, необхідні для встановлення на пристрій користувача. APK-файл упакований, підписаний цифровим сертифікатом і готовий до прямої інсталяції безпосередньо на мобільний пристрій.

Для платформи iOS — у разі відсутності публікації у App Store, застосовується формат IPA (iOS App Archive). Цей файл також підписаний відповідним сертифікатом розробника, і для його встановлення користувачу необхідно скористатися спеціальними методами, наприклад, через TestFlight, приватні профілі розповсюдження або інші засоби безпосередньої інсталяції (side-loading).

2. Вбудовані залежності

Всі сторонні бібліотеки, фреймворки і SDK, необхідні для коректної роботи додатку, інтегровані безпосередньо у збірку. Це включає React Native, Firebase SDK, бібліотеки для роботи з UI, навігацією, зображеннями тощо. Завдяки цьому користувач не потребує додаткової установки чи конфігурації зовнішніх компонентів.

3. Файли ресурсів

- Пакет містить всі необхідні ресурси, які використовуються у додатку:
- Зображення — іконки, логотипи, фотографії рецептів, кнопки, фони.
- Шрифти — для відображення тексту відповідно до дизайну.
- Мультимедійні файли — якщо передбачено, наприклад, аудіоінструкції чи відео.
- Локалізаційні файли — словники для підтримки різних мов інтерфейсу (українська, англійська).
- Ці ресурси оптимізовані за розміром, щоб не збільшувати загальний обсяг пакету та прискорити завантаження і встановлення.

4. Конфігураційні файли

У пакеті також містяться файли конфігурації, які визначають налаштування додатку:

- Параметри підключення до серверів та Firebase.
- Ключі API (якщо застосовуються) — захищені та зашифровані.
- Параметри локалізації, теми оформлення, поведінки інтерфейсу.
- Файли налаштувань безпеки, наприклад, політики доступу.

Ці конфігураційні файли забезпечують гнучкість і можливість адаптації додатку під різні умови експлуатації без необхідності перекомпіляції.

4.3.2 Допоміжні матеріали та документація. Для полегшення процесу встановлення та подальшої експлуатації додатку інсталяційний пакет містить повний набір документації.

Інструкція з встановлення – це документ містить покроковий опис процесу інсталяції для користувачів обох платформ. Враховуючи, що додаток не розміщується у Google Play або App Store, інструкція містить детальний опис альтернативних способів встановлення:

- Для Android — як встановити APK-файл вручну, включно з налаштуваннями безпеки (дозвіл на інсталяцію з невідомих джерел).
- Для iOS — методи встановлення через TestFlight або інші методи side-loading, вимоги до сертифікатів та профілів розробника.

Це забезпечує зручність для кінцевих користувачів і знижує ризик помилок при встановленні.

Керівництво користувача надає опис основного функціоналу додатку, рекомендації щодо введення інгредієнтів, створення рецептів, роботи з улюбленими, а також способів налаштування персональних уподобань. Документ містить відповіді на поширені питання і розв’язання можливих проблем.

Технічна документація передбачена для адміністраторів та розробників, включає:

- Опис архітектури додатку.
- Інструкції з оновлення та розгортання.
- Опис формату конфігураційних файлів.

- Відомості про логування і моніторинг

4.3.3 Засоби оновлення та підтримки. Оскільки додаток не розміщується у публічних магазинах додатків, система оновлення реалізована через:

- Розсилку оновлених інсталяційних файлів безпосередньо користувачам (через електронну пошту або інші канали).
- Використання внутрішніх скриптів оновлення, які перевіряють версію додатку і завантажують нові компоненти.
- Можливість відкотити оновлення до попередньої версії у разі виявлення критичних помилок.

Це забезпечує контроль за версіями додатку та мінімізує ризики пов’язані з оновленнями.

4.3.4 Вимоги до формату і розміру пакету. Для зручності передачі та зберігання інсталяційний пакет повинен відповідати наступним вимогам:

- Формат файлів — APK для Android, IPA для iOS, упаковані у стандартний архів для розповсюдження.
- Максимальний розмір пакету — не більше 150 МБ, що дозволяє швидко завантажувати файл через мобільні мережі.
- Відсутність зайвих залежностей, які збільшують обсяг і можуть викликати конфлікти.
- Надійне підписання пакету для забезпечення безпеки і автентичності.

4.3.5 Безпека інсталяційного пакету. Під час формування пакету особливу увагу приділено захисту від несанкціонованого доступу та модифікацій:

- Використання цифрового підпису додатку.
- Шифрування конфігураційних файлів з ключами API.
- Захист від зміни ресурсів і коду шляхом застосування механізмів цілісності.

Ці заходи дозволяють уникнути встановлення підроблених або модифікованих версій додатку, що є важливим для збереження довіри користувачів.

4.4 Висновки до розділу 4

У розділі 4 було детально розглянуто ключові аспекти впровадження та експлуатації мобільного додатку генерації рецептів на основі введених інгредієнтів.

По-перше, тестування системи, що включає як автоматичні тести за допомогою Jest, так і ручне тестування, дозволило підтвердити коректність роботи основних функціональних модулів, стабільність взаємодії з серверною частиною та зручність користування інтерфейсом.

По-друге, було визначено комплексні вимоги до апаратного та програмного забезпечення як кінцевих користувачів, так і розробників. Мінімальні та рекомендовані технічні параметри дозволяють забезпечити стабільну, швидку і безперебійну роботу додатку на більшості сучасних пристроїв, а також ефективне середовище розробки та тестування.

По-третє, описано склад інсталяційного пакету, що включає виконавчі файли, вбудовані залежності, ресурси, конфігураційні файли та повний набір документації. Враховуючи відсутність публікації у магазинах додатків, особлива увага приділяється методам інсталяції, оновлення і безпеки пакету, що забезпечує надійність і зручність у використанні.

Загалом, реалізовані підходи до впровадження та експлуатації системи гарантують високий рівень якості, безпеки та комфорт користування мобільним додатком, що є важливим фактором для його успішного поширення та подальшого розвитку.

ВИСНОВКИ

Під час виконання дипломної роботи було розроблено мобільний додаток для генерації рецептів на основі введених користувачем інгредієнтів. Цей додаток є актуальним рішенням для сучасних користувачів, які прагнуть оптимізувати процес приготування їжі, заощадити час і використати наявні продукти максимально ефективно.

Перший розділ роботи був присвячений аналізу предметної області та обґрунтуванню актуальності дослідження. Проведений аналіз дозволив виявити основні потреби та проблеми цільової аудиторії, серед яких найважливішими є нестача часу для вибору і планування меню, обмеженість кулінарних навичок та бажання скоротити витрати на продукти. Виходячи з цього, було сформульовано головну мету дослідження – створення зручного та функціонального мобільного додатку, що дозволяє ефективно вирішувати ці завдання.

Другий розділ був присвячений проектуванню інформаційної системи. Тут визначалися сутності для бази даних Firebase, продумувалися взаємозв’язки між ними, а також створювалися діаграми, які наочно демонстрували структуру програми та принципи взаємодії її складових.

У третьому розділі було здійснено безпосередню розробку програмних модулів додатку, включаючи алгоритми автентифікації користувачів, формування персоналізованих профілів, пошуку рецептів за заданими інгредієнтами та генерації рецептів з використанням штучного інтелекту. Реалізація алгоритмів забезпечила високу модульність та простоту в обслуговуванні програмного продукту.

Четвертий розділ роботи містив детальну інформацію щодо тестування розробленого програмного продукту. Було проведено як автоматизоване, так і ручне тестування. Під час тестування було перевірено роботу основних функцій додатку, таких як реєстрація, автентифікація, додавання та редагування рецептів, а також генерація нових рецептів.

Також у четвертому розділі були описані вимоги до апаратного і програмного забезпечення, необхідного для розгортання та експлуатації додатку, і склад інсталяційного пакету. Це дозволило чітко визначити умови, в яких програмний продукт може бути ефективно впроваджений і використовуватися без додаткових труднощів.

Отже, можна зробити висновок, що розроблений мобільний додаток повністю відповідає початковим цілям і задачам дипломного проекту. Продукт має сучасний інтерфейс, зручний для користувача дизайн і широку функціональність, яка дозволяє задовольнити потреби цільової аудиторії у швидкому і зручному плануванні щоденного харчування.

У майбутньому застосунок можна ще вдосконалити: наприклад, додати інтеграцію з іншими сервісами (для замовлення продуктів онлайн), розширити можливості персоналізації під різні дієти або алергії, а також поліпшити алгоритми для ще більш точного підбору рецептів. Це дасть змогу зробити продукт ще привабливішим для користувачів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Козак О. Л. Опорний конспект лекцій з курсу «Аналіз вимог до програмного забезпечення» для студентів напрямку підготовки «Програмна інженерія» / О. Л. Козак. – Тернопіль, 2011. – 56 с.
2. Гаркуша І. М. Конспект лекцій з дисципліни «Проектування інформаційних систем» для студентів галузі знань 12 «Інформаційні технології» спеціальності 126 «Інформаційні системи та технології» / І. М. Гаркуша. – Дніпро : НТУ «ДП», 2020. – 75 с.
3. Авраменко А. С., Авраменко В. С., Косенюк Г. В. Тестування програмного забезпечення : навч. посіб. – Черкаси : ЧНУ ім. Б. Хмельницького, 2017. – 284 с.
4. Стоян С. І. Мобільна розробка на React Native: практичний посібник. – Київ: Видавництво Ліра-К, 2020. – 328 с.
5. Гончарук В. Ю. Бази даних Firebase: теорія та практика. – Харків: Основа, 2021. – 212 с.
6. Григорчук І. О. JavaScript та TypeScript у сучасному програмуванні: навчальний посібник. – Львів: Видавництво Львівської політехніки, 2022. – 340 с.
7. Official React Native Documentation. – URL: https://reactnative.dev/docs/getting-started (дата звернення: 22.04.2025).
8. Firebase Documentation. – URL: https://firebase.google.com/docs (дата звернення: 22.04.2025).
9. Бейлі А., Бейлі М. UX/UI дизайн мобільних додатків: основні принципи та методики. – Київ: Артек, 2021. – 264 с.
10. Павловська І. О. Тестування програмного забезпечення: практичний посібник. – Харків: Фактор, 2022. – 320 с.

Додаток А. Конфігурація Firebase

```
import {initializeApp} from 'firebase/app';
import {getFirestore} from 'firebase/firestore';
import firebase from 'firebase/compat/app';
import 'firebase/compat/auth';
import {initializeAuth, getReactNativePersistence} from 'firebase/auth';
import ReactNativeAsyncStorage from '@react-native-async-storage/async-storage';
import {
FIREBASE_API_KEY, FIREBASE_AUTH_DOMAIN, FIREBASE_PROJECT_ID,
FIREBASE_STORAGE_BUCKET, FIREBASE_MESSAGING_SENDER_ID, FIREBASE_APP_ID, }
from '@env';

const firebaseConfig = {
apiKey: FIREBASE_API_KEY,
authDomain: FIREBASE_AUTH_DOMAIN,
projectId: FIREBASE_PROJECT_ID,
storageBucket: FIREBASE_STORAGE_BUCKET,
messagingSenderId: FIREBASE_MESSAGING_SENDER_ID,
appId: FIREBASE_APP_ID,
};

if (!firebase.apps.length) {
firebase.initializeApp(firebaseConfig);
}

export const FIREBASE_APP = initializeApp(firebaseConfig);
export const FIREBASE_DB = getFirestore(FIREBASE_APP);
export const FIREBASE_AUTH = initializeAuth(FIREBASE_APP, {
persistence: getReactNativePersistence(ReactNativeAsyncStorage),
});
```

Додаток Б. Запити до Firebase

```
export const createUser = async (
  userId: string,
  userData: Omit<User, 'id' | 'createdAt' | 'updatedAt'>,
): Promise<string> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await setDoc(userRef, {
    ...userData,
    createdAt: Date.now(),
    updatedAt: Date.now(),
  });
  return userId;
};

export const getUserById = async (userId: string): Promise<User | null> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  const snap = await getDoc(userRef);
  if (!snap.exists()) {
    return null;
  }
  return {id: snap.id, ...snap.data()} as User;
};

export const updateUserProfile = async (
  userId: string,
  data: Partial<
    Pick<
      User,
```

```
      'displayNameUa' | 'displayNameEn' | 'gender' | 'birthDate' | 'photoURL'
    >
  >,
): Promise<void> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await updateDoc(userRef, {
    ...data,
    updatedAt: Date.now(),
  });
};

export const updateUserAllergens = async (
  userId: string,
  allergens: string[],
): Promise<void> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await updateDoc(userRef, {
    allergens,
    updatedAt: Date.now(),
  });
};

export const addToFavorites = async (
  userId: string,
  recipeId: string,
): Promise<void> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await updateDoc(userRef, {
    favorites: arrayUnion(recipeId),
    updatedAt: Date.now(),
```

```
  });
};

export const removeFromFavorites = async (
  userId: string,
  recipeId: string,
): Promise<void> => {
  const userRef = doc(FIREBASE_DB, COLLECTIONS.USERS, userId);
  await updateDoc(userRef, {
    favorites: arrayRemove(recipeId),
    updatedAt: Date.now(),
  });
};

export const createCategory = async (
  category: Omit<Category, 'id'>,
): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.CATEGORIES);
  const docRef = await addDoc(colRef, category);
  return docRef.id;
};

export const getAllCategories = async (): Promise<Category[]> => {
  const q = query(
    collection(FIREBASE_DB, COLLECTIONS.CATEGORIES),
    orderBy('order', 'asc'),
  );
  const snap = await getDocs(q);
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Category));
};
```

```
export const updateCategory = async (
  categoryId: string,
  data: Partial<Category>,
): Promise<void> => {
  const categoryRef = doc(FIREBASE_DB, COLLECTIONS.CATEGORIES, categoryId);
  await updateDoc(categoryRef, data);
};

export const getCategoryById = async (
  categoryId: string,
): Promise<Category | null> => {
  const categoryRef = doc(FIREBASE_DB, COLLECTIONS.CATEGORIES, categoryId);
  const snap = await getDoc(categoryRef);
  return snap.exists() ? ({id: snap.id, ...snap.data()} as Category) : null;
};

export const createIngredient = async (
  ingredient: Omit<Ingredient, 'id'>,
): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.INGREDIENTS);
  const docRef = await addDoc(colRef, ingredient);
  return docRef.id;
};

export const getAllIngredients = async (): Promise<Ingredient[]> => {
  const snap = await getDocs(collection(FIREBASE_DB, COLLECTIONS.INGREDIENTS));
```

```
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Ingredient));
};

export const getIngredientById = async (
  ingredientId: string,
): Promise<Ingredient | null> => {
  try {
    if (!ingredientId) {
      console.error('Invalid ingredient ID:', ingredientId);
      return null;
    }

    const ref = doc(FIREBASE_DB, COLLECTIONS.INGREDIENTS, ingredientId);
    const snap = await getDoc(ref);

    if (!snap.exists()) {
      return null;
    }

    const ingredient = {id: snap.id, ...snap.data()} as Ingredient;
    return ingredient;
  } catch (error) {
    console.error(`Error fetching ingredient ${ingredientId}:`, error);
    return null;
  }
};

export const updateIngredient = async (
  ingredientId: string,
```

```
  data: Partial<Ingredient>,
): Promise<void> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.INGREDIENTS,
ingredientId);
  await updateDoc(ref, data);
};

export const addComment = async (
  recipeId: string,
  comment: Comment,
): Promise<void> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.RECIPES, recipeId);
  await updateDoc(ref, {comments: arrayUnion(comment)});
};

export const createRecipe = async (
  recipe: Omit<
    Recipe,
    'id' | 'rating' | 'reviewCount' | 'createdAt' | 'updatedAt'
  >,
): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.RECIPES);
  const docRef = await addDoc(colRef, {
    ...recipe,
    rating: 0,
    reviewCount: 0,
    createdAt: Date.now(),
    updatedAt: Date.now(),
  });
  return docRef.id;
```

```
};

export const getRecipesByCategory = async (
  categoryId: string,
): Promise<Recipe[]> => {
  const q = query(
    collection(FIREBASE_DB, COLLECTIONS.RECIPES),
    where('categoryIds', 'array-contains', categoryId),
    orderBy('createdAt', 'desc'),
  );
  const snap = await getDocs(q);
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Recipe));
};

export const getRecipesByAuthor = async (
  authorId: string,
): Promise<Recipe[]> => {
  const q = query(
    collection(FIREBASE_DB, COLLECTIONS.RECIPES),
    where('authorId', '==', authorId),
    orderBy('createdAt', 'desc'),
  );
  const snap = await getDocs(q);
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Recipe));
};

export const getRecipeById = async (
  recipeId: string,
): Promise<Recipe | null> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.RECIPES, recipeId);
```

```
  const snap = await getDoc(ref);
  if (!snap.exists()) {
    return null;
  }
  return {id: snap.id, ...snap.data()} as Recipe;
};

export const updateRecipe = async (
  recipeId: string,
  data: Partial<Recipe>,
): Promise<void> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.RECIPES, recipeId);
  await updateDoc(ref, {...data, updatedAt: Date.now()});
};

export const deleteRecipe = async (recipeId: string): Promise<void> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.RECIPES, recipeId);
  await deleteDoc(ref);
};

export const getAllRecipes = async (): Promise<Recipe[]> => {
  const snap = await getDocs(collection(FIREBASE_DB,
COLLECTIONS.RECIPES));
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Recipe));
};

export const createAllergen = async (
  allergen: Omit<Allergen, 'id'>,
): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.ALLERGENS);
```

```
  const docRef = await addDoc(colRef, allergen);
  return docRef.id;
};

export const getAllAllergens = async (): Promise<Allergen[]> => {
  const snap = await getDocs(collection(FIREBASE_DB,
COLLECTIONS.ALLERGENS));
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Allergen));
};

export const getAllergenById = async (
  allergenId: string,
): Promise<Allergen | null> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.ALLERGENS, allergenId);
  const snap = await getDoc(ref);
  if (!snap.exists()) {
    return null;
  }
  return {id: snap.id, ...snap.data()} as Allergen;
};

export const getFeaturedRecipes = async (maxResults: number = 10) => {
  try {
    const colRef = collection(FIREBASE_DB, COLLECTIONS.RECIPES);
    const q = query(
      colRef,
      orderBy(FIELDS.CREATED_AT, 'desc'),
      limit(maxResults),
    );
    const snap = await getDocs(q);
```

```
    const data: Recipe[] = snap.docs.map(
      d => ({id: d.id, ...d.data()} as Recipe),
    );
    return {success: true, data};
  } catch (error) {
    console.error('Error getting featured recipes:', error);
    return {success: false, error};
  }
};

export const createType = async (type: Omit<Type, 'id'>): Promise<string> => {
  const colRef = collection(FIREBASE_DB, COLLECTIONS.TYPES);
  const docRef = await addDoc(colRef, type);
  return docRef.id;
};

export const getAllTypes = async (): Promise<Type[]> => {
  const snap = await getDocs(collection(FIREBASE_DB,
COLLECTIONS.TYPES));
  return snap.docs.map(d => ({id: d.id, ...d.data()} as Type));
};

export const getTypeById = async (typeId: string): Promise<Type | null> => {
  const ref = doc(FIREBASE_DB, COLLECTIONS.TYPES, typeId);
  const snap = await getDoc(ref);
  if (!snap.exists()) {
    return null;
  }
  return {id: snap.id, ...snap.data()} as Type;
};
```

```
export const getSearchHistory = async (userId: string): Promise<string[]> => {
  const historyRef = collection(
    FIREBASE_DB,
    COLLECTIONS.USERS,
    userId,
    'search_history',
  );
  const q = query(historyRef, orderBy('createdAt', 'desc'), limit(20));
  const snap = await getDocs(q);
  const terms: string[] = [];
  snap.forEach(docSnap => {
    const data = docSnap.data() as {term: string; createdAt: number};
    if (!terms.includes(data.term)) {
      terms.push(data.term);
    }
  });
  return terms.slice(0, 5);
};

export const saveSearchTerm = async (
  userId: string,
  searchTerm: string,
): Promise<void> => {
  const historyRef = collection(
    FIREBASE_DB,
    COLLECTIONS.USERS,
    userId,
    'search_history',
  );
```

```
  await addDoc(historyRef, {
    term: searchTerm,
    createdAt: Date.now(),
  });
};

export const searchRecipes = async (searchTerm: string): Promise<Recipe[]> => {
  if (!searchTerm || searchTerm.trim() === '') {
    return getAllRecipes();
  }

  const normalizedTerm = searchTerm.trim();
  const results: Recipe[] = [];
  const processedIds = new Set<string>();

  const recipesCollection = collection(FIREBASE_DB, COLLECTIONS.RECIPES);

  try {
    const titleEnQuery = query(
      recipesCollection,
      where(FIELDS.TITLE_EN, '>=', normalizedTerm),
      where(FIELDS.TITLE_EN, '<=', normalizedTerm + '\uf8ff'),
    );

    const titleEnSnap = await getDocs(titleEnQuery);
    titleEnSnap.docs.forEach(doc => {
      if (!processedIds.has(doc.id)) {
        results.push({id: doc.id, ...doc.data()} as Recipe);
        processedIds.add(doc.id);
```

```
    }
  });
} catch (error) {
  console.error('Error searching recipes:', error);
}

try {
  const titleUaQuery = query(
    recipesCollection,
    where(FIELDS.TITLE_UA, '>=', normalizedTerm),
    where(FIELDS.TITLE_UA, '<=', normalizedTerm + '\uf8ff'),
  );

  const titleUaSnap = await getDocs(titleUaQuery);
  titleUaSnap.docs.forEach(doc => {
    if (!processedIds.has(doc.id)) {
      results.push({id: doc.id, ...doc.data()} as Recipe);
      processedIds.add(doc.id);
    }
  });
} catch (error) {
  console.error('Error searching recipes:', error);
}

try {
  const descEnQuery = query(
    recipesCollection,
    where(FIELDS.DESCRIPTION_EN, '>=', normalizedTerm),
    where(FIELDS.DESCRIPTION_EN, '<=', normalizedTerm + '\uf8ff'),
  );
```

```
  const descEnSnap = await getDocs(descEnQuery);
  descEnSnap.docs.forEach(doc => {
    if (!processedIds.has(doc.id)) {
      results.push({id: doc.id, ...doc.data()} as Recipe);
      processedIds.add(doc.id);
    }
  });
} catch (error) {
  console.error('Error searching recipes:', error);
}

try {
  const descUaQuery = query(
    recipesCollection,
    where(FIELDS.DESCRIPTION_UA, '>=', normalizedTerm),
    where(FIELDS.DESCRIPTION_UA, '<=', normalizedTerm + '\uf8ff'),
  );

  const descUaSnap = await getDocs(descUaQuery);
  descUaSnap.docs.forEach(doc => {
    if (!processedIds.has(doc.id)) {
      results.push({id: doc.id, ...doc.data()} as Recipe);
      processedIds.add(doc.id);
    }
  });
} catch (error) {
  console.error('Error searching recipes:', error);
}
```

```
  return results;
};

export const getFilteredRecipes = async (
  ingredients: string[] = [],
  allergens: string[] = [],
  types: string[] = [],
  categories: string[] = [],
): Promise<Recipe[]> => {
  try {
    if (!Array.isArray(categories)) {
      categories = [];
    }
    if (!Array.isArray(types)) {
      types = [];
    }
    if (!Array.isArray(ingredients)) {
      ingredients = [];
    }
    if (!Array.isArray(allergens)) {
      allergens = [];
    }

    const hasFilters =
      categories.length > 0 ||
      types.length > 0 ||
      ingredients.length > 0 ||
      allergens.length > 0;

    if (!hasFilters) {
```

```
    return await getAllRecipes();
  }

  let recipesQuery = query(collection(FIREBASE_DB, COLLECTIONS.RECIPES));

  if (categories.length === 1 && types.length === 0) {
    recipesQuery = query(
      recipesQuery,
      where('category', '==', categories[0]),
    );
  } else if (categories.length === 0 && types.length === 1) {
    recipesQuery = query(recipesQuery, where(FIELDS.TYPE, '==', types[0]));
  } else if (categories.length === 0 && types.length > 1) {
    recipesQuery = query(recipesQuery, where(FIELDS.TYPE, 'in', types));
  } else {
    const snap = await getDocs(collection(FIREBASE_DB, COLLECTIONS.RECIPES));
    let allRecipes = snap.docs.map(d => ({id: d.id, ...d.data()} as Recipe));

    if (categories.length > 0) {
      allRecipes = allRecipes.filter(
        recipe =>
          recipe && recipe.category && categories.includes(recipe.category),
      );
    }

    if (types.length > 0) {
      allRecipes = allRecipes.filter(
        recipe => recipe && recipe.type && types.includes(recipe.type),
```

```
        );
      }

      return await filterByIngredientsAndAllergens(
        allRecipes,
        ingredients,
        allergens,
      );
    }

    const snap = await getDocs(recipesQuery);
    let filteredRecipes = snap.docs.map(
      d => ({id: d.id, ...d.data()} as Recipe),
    );

    return await filterByIngredientsAndAllergens(
      filteredRecipes,
      ingredients,
      allergens,
    );
  } catch (error) {
    console.error('Error filtering recipes:', error);
    return [];
  }
};
```

Додаток В. Навігація додатку

```
import React, {useEffect, useState} from 'react';
import {createNativeStackNavigator} from '@react-navigation/native-stack';
import {
  PrivateStackParamList,
  PublicStackParamList,
  RootStackParamList,
} from './types';
import FilterScreen from '@Screens/filter';
import IngredientsScreen from '@Screens/ingredients';
import {NavigationContainer} from '@react-navigation/native';
import LoginScreen from '@Screens/login';
import EmailConfirmationScreen from '@Screens/email-confirmation';
import FavoriteRecipes from '@Screens/favorite-recipes';
import ForgotPassword from '@Screens/forgot-password';
import Intro from '@Screens/intro';
import Recipe from '@Screens/recipe';
import GeneratedRecipe from '@Screens/generated-recipe';
import Register from '@Screens/register';
import MyRecipes from '@Screens/my-recipes';
import Language from '@Screens/language';
import Theme from '@Screens/theme';
import AllergensScreen from '@Screens/allergens';
import {onAuthStateChanged, User} from 'firebase/auth';
import AsyncStorage from '@react-native-async-storage/async-storage';
import {FIREBASE_AUTH} from 'src/config/FirebaseConfig';
import BottomTabNavigator from './bottom-tab-navigator';
import Loading from '@Screens/loading';
import UserInfo from '@Screens/user-info';
```

```
const Stack = createNativeStackNavigator<RootStackParamList>();
const PublicStack = createNativeStackNavigator<PublicStackParamList>();
const PrivateStack = createNativeStackNavigator<PrivateStackParamList>();

type PrivateStackNavigatorProps = {
  initialRouteName: keyof PrivateStackParamList;
};

const PublicStackNavigator = () => {
  return (
    <PublicStack.Navigator
      initialRouteName="Login"
      screenOptions={{
        headerShown: false,
      }}>
      <PublicStack.Screen name="Login" component={LoginScreen} />
      <PublicStack.Screen name="Register" component={Register} />
      <PublicStack.Screen
        name="EmailConfirmation"
        component={EmailConfirmationScreen}
      />
      <PublicStack.Screen  name="ForgotPassword"  component={ForgotPassword}
/>
    </PublicStack.Navigator>
  );
};

const PrivateStackNavigator = ({
  initialRouteName,
```

```
}: PrivateStackNavigatorProps) => {
  return (
    <PrivateStack.Navigator
      initialRouteName={initialRouteName}
      screenOptions={{
        headerShown: false,
      }}>
      <PrivateStack.Screen  name="MainTabs"  component={BottomTabNavigator}
/>
      <PrivateStack.Screen name="Filter" component={FilterScreen} />
      <PrivateStack.Screen name="Ingredients" component={IngredientsScreen} />
      <PrivateStack.Screen name="Allergens" component={AllergensScreen} />
      <PrivateStack.Screen  name="FavoriteRecipes"  component={FavoriteRecipes}
/>
      <PrivateStack.Screen name="Intro" component={Intro} />
      <PrivateStack.Screen name="UserInfo" component={UserInfo} />
      <PrivateStack.Screen name="RecipeDetails" component={Recipe} />
      <PrivateStack.Screen                                    name="GeneratedRecipe"
component={GeneratedRecipe} />
      <PrivateStack.Screen name="MyRecipes" component={MyRecipes} />
      <PrivateStack.Screen name="Language" component={Language} />
      <PrivateStack.Screen name="Theme" component={Theme} />
    </PrivateStack.Navigator>
  );
};

const AppNavigator = () => {
  const [userAuth, setUserAuth] = useState<User | null>(null);
  const [isLoading, setIsLoading] = useState(true);
  const [isFirstLogin, setIsFirstLogin] = useState<boolean | null>(null);
```

```
useEffect(() => {
  setIsLoading(true);
  const unsubscribe = onAuthStateChanged(FIREBASE_AUTH, user => {
    setUserAuth(user);
    setIsLoading(false);
    if (user) {
      (async () => {
        try {
          const storedUserId = await AsyncStorage.getItem('userId');
          if (storedUserId !== user.uid) {
            await AsyncStorage.setItem('userId', user.uid);
            setIsFirstLogin(true);
          } else {
            setIsFirstLogin(false);
          }
        } catch (err) {
          setIsFirstLogin(false);
        }
      })();
    } else {
      setIsFirstLogin(null);
    }
  });
  return () => unsubscribe();
}, []);

if (isLoading || (userAuth && isFirstLogin === null)) {
  return (
    <NavigationContainer fallback={<Loading />}>
```

```
        <Stack.Navigator screenOptions={{headerShown: false}}>
          <Stack.Screen name="Loading" component={Loading} />
        </Stack.Navigator>
      </NavigationContainer>
    );
  }

  return (
    <NavigationContainer fallback={<Loading />}>
      <Stack.Navigator screenOptions={{headerShown: false}}>
        {!userAuth ? (
          <Stack.Screen name="Public" component={PublicStackNavigator} />
        ) : (
          <Stack.Screen name="Private">
            {() => (
              <PrivateStackNavigator
                initialRouteName={isFirstLogin ? 'UserInfo' : 'MainTabs'}
              />
            )}
          </Stack.Screen>
        )}
      </Stack.Navigator>
    </NavigationContainer>
  );
};

export default AppNavigator;
```

Додаток Д. Код налаштування локалізації

```
import i18n from 'i18next';
import {initReactI18next} from 'react-i18next';
import en from './locales/en.json';
import ua from './locales/ua.json';

const resources = {
  en: {
    translation: en,
  },
  ua: {
    translation: ua,
  },
};

i18n.use(initReactI18next).init({
  resources,
  lng: 'en',
  fallbackLng: 'en',
  interpolation: {
    escapeValue: false,
  },
  compatibilityJSON: 'v4',
});

export type LanguageType = 'en' | 'ua';
export enum Language {
  EN = 'en',
  UA = 'ua',
}
```

```
export const useLanguage = () => {
  const {i18n} = useTranslation();

  const changeLanguage = useCallback(
    (language: LanguageType) => {
      i18n.changeLanguage(language);
    },
    [i18n],
  );

  const currentLanguage = i18n.language as LanguageType;

  const getFieldBasedOnLanguage = useCallback(
    (object: any, field: string) => {
      return currentLanguage === 'en'
        ? object[field + 'En']
        : object[field + 'Ua'];
    },
    [currentLanguage],
  );

  return {
    currentLanguage,
    changeLanguage,
    getFieldBasedOnLanguage,
  };
};
export default i18n;
```