Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Мобільний застосунок для управління кулінарними рецептами та інгредієнтами
Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.220197.01.03.ПЗ

Виконав студент IV курсу, група ІПЗ-22-1 _____ Руслан ВАВУЛЬСЬКИЙ-ЗАПАСНИК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник асистентка _____ Анастасія ДЬОМІНА
Науковий ступінь, вчене звання Підпис Ім'я, ПРІЗВИЩЕ

Нормоконтролер канд. техн. наук, доцент _____ Юрій ФОРКУН
Підпис Ім'я, ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я, ПРІЗВИЩЕ

1 червня 2026 р.

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
_____ Л. П. Бедратюк
_____ 02.01.2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Вавульському-Запаснику Руслану Артемовичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Мобільний застосунок для управління кулінарними рецептами та інгредієнтами

Керівник роботи Дьоміна Анастасія Іванівна, асистентка
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01. 2026 р. № 7
2. Строк подання студентом роботи на кафедру 01.06.2026 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задач, проєктування, програмна реалізація та тестування застосунку.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Два креслення
1. Діаграма варіантів використання
2. Діаграма «сутність-зв'язок»
3. Презентаційні слайди – 17 шт.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Форкун Ю. В., доцент	05.05.26	25.05.26
Антиплагіат	Форкун Ю. В., доцент	05.05.26	25.05.26

7. Дата видачі завдання «_2_» __січня__ 2026 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Ознайомлення з тематикою дипломного проектування, визначення та узгодження індивідуальних тем кваліфікаційних робіт (КвР)	01.12– 31.12.2025	
2 Збір матеріалу за темою КвР; дослідження предметної області, в якій планується використ- тання програмного забезпечення (ПЗ), визначення задач та вимог, розробка технічного завдання	01.01 – 20.02.2026	
3 Проектування програмного забезпечення	21.02 – 20.03 2026	
4 Програмна реалізація з використанням відповідних засобів розробки. Тестування ПЗ	21.03 – 30.04.2026	
5 Написання вступу, загальних висновків, оформлення переліку джерел посилання та додатків. Оформлення пояснювальної записки КвР згідно вимог	01.05 – 25.05.2026	
6 Попередній захист КвР	15.05.2026	
7 Перевірка КвР на плагіат, нормоконтроль, отримання відгуків, рецензій та інших супровідних документів. Брошурування (зшиття) пояснювальної записки.	26.05 – 30.06.2026	
8 Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії ХНУ; підготовка до захисту та захист КвР	з 01.06.2026	

Студент _____ Руслан Вавильський-Заволинський
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ Анастасія Дьоміна
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Мобільний застосунок для управління кулінарними рецептами та інгредієнтами».

Автор роботи: Вавульський-Запасник Руслан Артемович.

Керівник роботи: Дьоміна Анастасія Іванівна.

Пояснювальна записка: 84 с., 34 рис., 12 табл., 2 дод., 34 **джерел.**

Графічна частина: 2 креслення ф. А3, презентаційні слайди 17 шт.

БАЗА ДАНИХ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, КУЛІНАРНІ РЕЦЕПТИ, МОБІЛЬНИЙ ЗАСТОСУНОК, ФРЕЙМВОРК SWIFTUI, ХМАРНІ СЕРВІСИ, ШТУЧНИЙ ІНТЕЛЕКТ.

Мета кваліфікаційної роботи: розробити мобільний застосунок для зберігання та управління кулінарними рецептами, який включає штучний інтелект для генерації нових рецептів

У кваліфікаційній роботі проведено аналіз предметної області та її інформаційного забезпечення, визначено функціональні та нефункціональні вимоги до програмної системи, розроблено загальну архітектуру застосунку, спроєктовано базу даних та структуру застосунку.

Для реалізації програмної системи використано мову програмування Swift, декларативний фреймворк SwiftUI, хмарну платформу Firebase для збереження даних і синхронізації в реальному часі, а також інтегровано зовнішній програмний інтерфейс OpenAI API.

В результаті виконання кваліфікаційної роботи здійснена програмна реалізація мобільного застосунку, який автоматизує роботу з рецептами та списками покупок, а також проведено практичну апробацію та тестування розробленого програмного забезпечення.

29.05.26
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А	КвРІПЗ.220197.01.03.ПЗ	Пояснювальна записка	84		
2	А		Завдання на кваліфіка-	1		
			ційну роботу			
3	А		Анотація	1		
			Графічні документи			
4	А	КвРІПЗ.220197.01.03.Е8	UML-діаграма варіантів	1		
			використання			
5	А	КвРІПЗ.220197.01.03.Е8	Діаграма «сутність-	1		
			зв'язок»			
6	А	КвРІПЗ.220197.01.03.ПЗ	Презентаційні матеріали	17		

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ.220197.01.03.ВД		
Виконав		Вавульський-Запасник Р. А		01.06	Мобільний застосунок для управління кулінарними рецептами та інгредієнтами Відомість документів	Літ.	Арк. 1 / Аркушів 1
Керівник		Дьоміна А. І.		01.06			
Рецензент						ХНУ, ІПЗ-22-1	
Н. контр.		Форкун Ю. В.		05.06			
Зав. каф.		Бедратюк Л.П.					

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API	–	Application programming interface
HIG	–	Human Interface Guidelines
IDEF0	–	Integration Definition for Function Modeling
JSON	–	JavaScript Object Notation
MVVM	–	Model-View-ViewModel
NoSQL	–	Not Only SQL
UML	–	Unified Modeling Language
БД	–	база даних
ООП	–	об’єктно-орієнтоване програмування
ПЗ	–	програмне забезпечення
ШІ	–	штучний інтелект

ВСТУП

Більшість людей вже рідко користується паперовими кулінарними книгами, віддаючи перевагу пошуку рецептів в Інтернеті. Звичним стало використання мобільного телефону під час приготування їжі. На ринку існує багато таких програм: великі портали з користувацьким контентом часто містять рецепти низької якості, помилки у пропорціях і неякісні фотографії, тоді як професійні застосунки від відомих шеф-кухарів зазвичай є закритими та не дозволяють додавати власні рецепти чи зберігати фотографії страв. Серед популярних рішень варто виділити Culinara (акцент на відеоконтенті від шефів), Kitchen Stories (покрокові інструкції з автоматичним перерахунком), Cookpad (соціальна мережа рецептів). Детальний порівняльний аналіз цих продуктів проведено у підрозділі 1.2.

Предметна область охоплює зберігання та управління структурованими кулінарнимирецептами від простих домашніх страв до складних шеф рецептів із точними вагамита технологічними картами Функціонально система має справу з об єктами рецептназва інгредієнти кроки час складність інгредієнт назва одиниця виміру кількість список покупок тематична колекція профіль користувача

Відтак, виникає проблема відсутності єдиного та зручного інструменту, який би поєднував у собі доступ до якісного, перевіреного редакторами контенту та можливість вести власну, повністю приватну книгу рецептів. На сьогодні користувачі змушені використовувати одразу кілька програм: браузер для пошуку, галерею телефону для збереження знімків екрана з рецептами, а також окремі застосунки для створення списку продуктів перед походом до магазину. Це незручно та забирає зайвий час.

Вирішення цієї проблеми потребує розроблення застосунку з гнучкою клієнт-серверною архітектурою та надійною системою збереження даних для обробки текстової інформації й фотографій страв. Використання хмарних рішень забезпечує ефективну взаємодію між користувачами та синхронізацію контенту.

Актуальність теми кваліфікаційної роботи полягає у потребі створення мобільного застосунку для вирішення повсякденних проблем користувачів у сфері організації побуту. Застосунок надаватиме комплексне рішення: перегляд професійних рецептів, формування списків покупок і генерацію нових рецептів за допомогою штучного інтелекту. За даними Technavio [1], ринок мобільних додатків для кулінарних рецептів щороку зростає більш ніж на 10%, а розвиток великих мовних моделей зробив можливою автоматичну генерацію рецептів на основі наявних продуктів.

Застосунок орієнтований на три категорії користувачів: кінцеві споживачі (домашні кухарі та гурмани), які хочуть зберігати та знаходити рецепти; гостьові користувачі, що переглядають публічний контент без реєстрації; адміністратори, які наповнюють платформу перевіреним контентом від шеф-кухарів.

Мета кваліфікаційної роботи: розробити мобільний застосунок для зберігання та управління кулінарними рецептами, який включає штучний інтелект для генерації нових рецептів.

Досягнення поставленої мети здійснюється шляхом виконання завдань:

– виконати аналіз існуючих програмних рішень на ринку кулінарних застосунків та встановити функціональні особливості предметної області;

– визначити вимоги до програмного забезпечення з урахуванням поділу на рівні доступу;

– розробити архітектуру та спроєктувати структуру бази даних;

– спроєктувати інтерфейс користувача;

– виконати програмну реалізацію застосунку, зокрема реалізувати алгоритми динамічного перерахунку інгредієнтів, систему рейтингу, збереження приватних записів та інтеграцію помічника;

– провести тестування розробленого програмного забезпечення для перевірки його працездатності, стабільності та відповідності поставленим вимогам технічного завдання.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовий аналіз предметної області

Предметною областю, для якої планується розроблення програмного забезпечення, є сфера кулінарії. Процес приготування їжі є частиною щоденного життя людини. Раніше для збереження інформації про страви використовувалися виключно друковані кулінарні книги або зошити з рукописними нотатками. Сьогодні користувачам зручніше тримати телефон під рукою на кухні, ніж шукати потрібну сторінку у книзі.

Знаходячи цікавий рецепт у мережі Інтернет, людина зазвичай робить знімок екрана або зберігає посилання в месенджері. З часом таких записів стає занадто багато, вони не мають зручного пошуку, не структуровані, а фотографії губляться серед інших зображень у галереї телефону. За даними аналітичної компанії Technavio, ринок мобільних додатків для рецептів продовжує динамічно зростати [1]. На Рисунку 1.1 наведено прогноз динаміки зростання обсягу світового ринку додатків для рецептів у період 2025–2029 років.

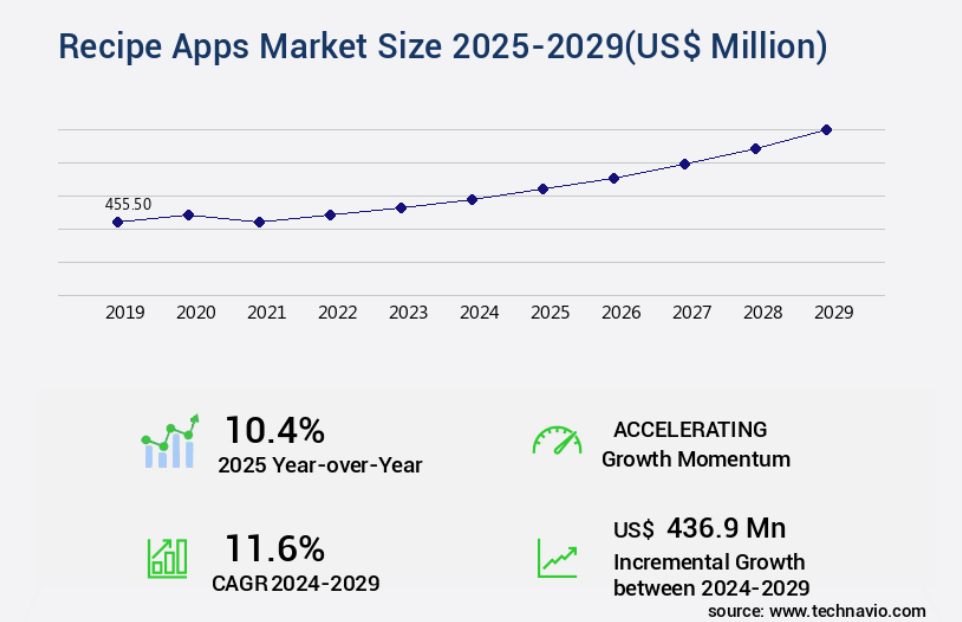


Рисунок 1.1 – Прогноз за даними Technavio [1]

Окрема проблема – відсутність автоматизації рутинних задач на домашній кухні. Наприклад, коли користувач знаходить рецепт, розрахований на дві порції, а йому необхідно приготувати страву на п'ятьох осіб, виникає потреба перераховувати вручну.

Застосунок розробляється для широкої аудиторії, тому кінцевих користувачів необхідно розділити на три ролі. Кожна з ролей має власні інформаційні потреби та права доступу до системи (Таблиця 1.1).

Таблиця 1.1 – Характеристика ролей користувачів застосунку

Роль	Короткий опис інформаційних потреб та прав доступу
Гість	Перегляд загальнодоступних рецептів. Не може зберігати страви, залишати оцінки чи створювати списки продуктів
Користувач	Повний доступ до всього функціоналу: збереження улюблених страв, створення записів, ШІ-помічник та оцінювання контенту
Адміністратор	Особа, що відповідає за наповнення застосунку контентом. Створює та редагує публічні рецепти, додає профілі шеф-кухарів у базу даних, керує колекціями та має доступ до переліку зареєстрованих користувачів.

Візуалізацію ролей та доступних їм функцій представлено на діаграмі варіантів використання, яка наведена на Рисунку 1.2.

Рисунок 1.2 – Діаграма варіантів використання для акторів системи

Для формалізованого опису функціональних особливостей предметної області та візуалізації процесу перетворення інформації обрано методологію моделювання IDEF0 [2]. Згідно зі стандартом FIPS PUB 183, ця методологія дозволяє представити систему як сукупність ієрархічно впорядкованих функцій, що взаємодіють між собою [3]. Вибір IDEF0 зумовлений її здатністю чітко відобразити вхідні дані, кінцеві результати, керуючі впливи та механізми виконання процесів. Контекстна діаграма (рівень A-0) представляє систему як єдиний функціональний блок. На Рисунку 1.3 наведено загальну схему функціонування мобільного застосунку.

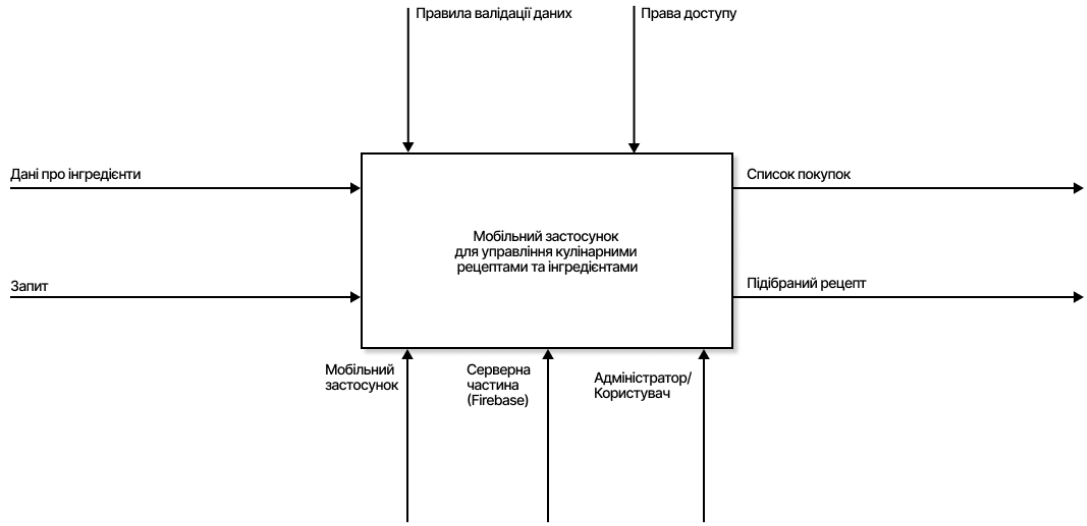


Рисунок 1.3 – Контекстна діаграма функціонування системи (рівень A-0)

Згідно з діаграмою (Рисунок 1.3), взаємодія із системою описується через чотири типи стрілок:

– вхід – це дані про інгредієнти, що вносяться користувачем, пошукові запити та фільтри;

– управління – це правила валідації даних, права доступу тощо;

– вихід – це сформований список покупок, підібраний рецепт та оновлений стан складу продуктів. Процес перетворення інформації полягає в автоматизованому опрацюванні запитів користувача: на основі обраних інгредієнтів

та фільтрів система генерує релевантну видачу контенту та виконує динамічний перерахунок ваги продуктів залежно від заданої кількості порцій;

– механізми, такі як мобільний застосунок (iOS), серверна частина Firebase, а також самі актори (Користувач, Адміністратор), які ініціюють процеси;

Для забезпечення повноцінної роботи зареєстрованого користувача передбачено навігацію, яка складається з п'яти основних розділів:

– «Головна сторінка», де відображаються сповіщення, загальні рекомендації та спеціальна рубрика від редакторів;

– «Пошук», який забезпечує можливість фільтрування та сортування як окремих рецептів, так і цілих колекцій;

– «Мої рецепти», що є основним робочим простором користувача і включає підрозділи для улюблених страв, збережених колекцій, створених користувачем збірок, фотографій з підписами та приватних записів (які доступні лише автору з можливістю подальшої модерації для публікації);

– «Список покупок», куди можна безпосередньо з картки рецепта додавати необхідні інгредієнти, або вводити їх вручну;

– «Профіль», де зберігаються налаштування облікового запису.

Для формування структури даних було проведено аналіз друкованих збірок рецептів та рукописних кулінарних нотаток. На основі аналізу виділено перелік реквізитів, необхідних для перенесення в таблиці баз даних. Основним об’єктом в таблицях бази даних є «Рецепт». Перелік його обов'язкових атрибутів включає: унікальний ідентифікатор, текстову назву страви, посилання на графічне зображення (фотографію), масив текстових тегів для пошуку, детальний опис етапів приготування. Окрім цього, рецепт містить реквізити списку продуктів, ідентифікатор автора (користувача або шеф-кухаря), посилання на батьківську колекцію, числове значення рейтингу (від нуля до п'яти) та спеціальне логічне поле, яке вказує, чи був рецепт згенерований за допомогою штучного інтелекту.

Другим важливим документом є «Профіль шеф-кухаря». Слід зазначити, що шеф-кухарі не мають власних облікових записів у системі для авторизації. Їхні профілі – це інформаційні сутності, які створює адміністратор для

відображення авторства публічних рецептів. Реквізити цього документа містять: ім'я автора, його фотографію, унікальний ідентифікатор та посилання на соціальні мережі.

Найбільш доцільним підходом до збереження інформації є нереляційний. Згідно з визначенням AWS, бази даних NoSQL розроблені спеціально для створення застосунків з гнучкими моделями даних [4]. На Рисунку 1.4 продемонстровано порівняння класичної реляційної моделі та документо-орієнтованої моделі NoSQL, що використовується в проєкті. Відповідно до міжнародного стандарту ECMA-404, JSON – це легкий текстовий формат обміну даними, який дозволяє структурувати інформацію у вигляді наборів пар «ключ-значення» [5].

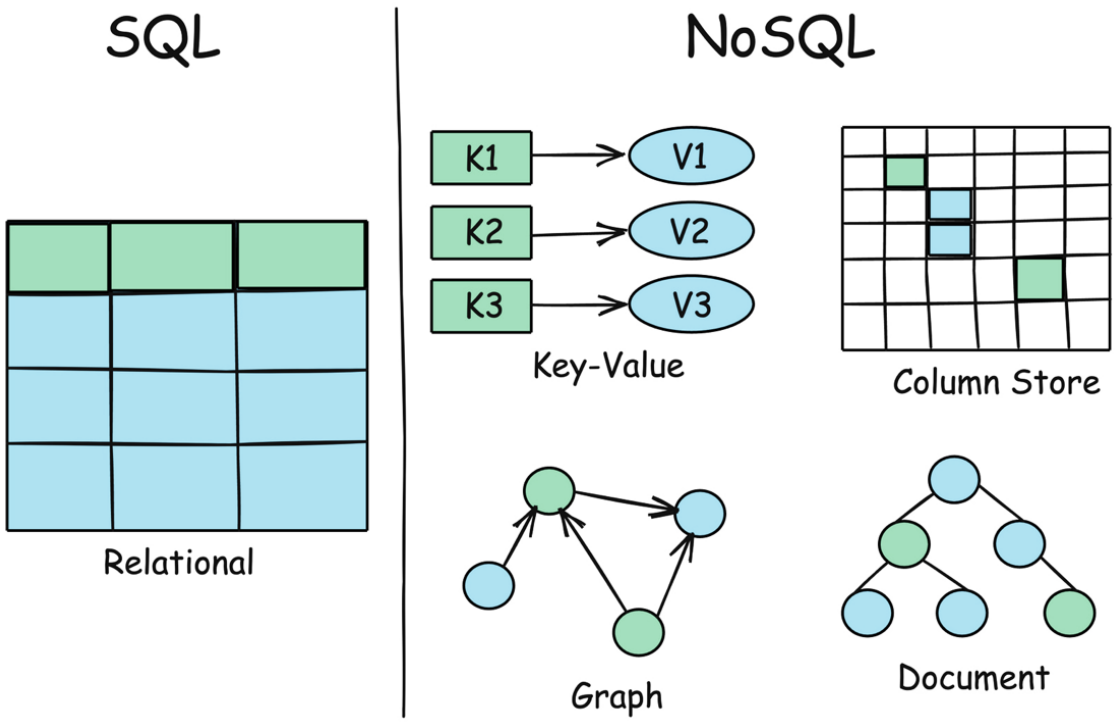


Рисунок 1.4 – Порівняння SQL та NoSQL

Для реалізації такого підходу в межах даного проєкту обрано платформу Firebase. Відповідно до офіційної документації Google, Firebase надає хмарну базу даних NoSQL, яка забезпечує синхронізацію даних між клієнтами в режимі реального часу та надає потужні інструменти для валідації запитів на рівні

сервера [6]. Це дозволяє автоматизувати рутинні процеси опрацювання інформації, які раніше виконувалися користувачами вручну.

На основі результатів цього аналізу можна чітко описати проблему, яка буде вирішена за допомогою майбутнього програмного забезпечення: існує об'єктивна необхідність створення єдиного мобільного застосунку, який дозволить користувачам не лише отримувати доступ до якісного кулінарного контенту від професійних редакторів, але й матиме функціонал для ведення власної приватної книги рецептів, зручного управління списками покупок та автоматичного перерахунку пропорцій.

Окремо необхідно проаналізувати безпекові аспекти розробки. Оскільки застосунок оперує як загальнодоступними, так і приватними даними, процеси ідентифікації та аутентифікації користувачів є критично важливими. Будь-яка особа, яка завантажує програму, спочатку отримує статус гостя. Якщо гість намагається виконати дію, що потребує запису в базу даних (наприклад, зберегти колекцію), система блокує цей запит і виводить на екран екранну форму з вимогою увійти в систему або зареєструватися.

Безпека приватних рецептів забезпечується на рівні правил бази даних, тобто кожен збережений або згенерований користувачем документ отримує маркер з унікальним ідентифікатором його власника. Система перевіряє цей маркер при кожному запиті на зчитування. Таким чином, гарантується, що ніхто, крім самого автора, не зможе переглянути його приватні кулінарні нотатки, поки він сам не вирішить відправити їх на модерацію для публікації у загальний доступ.

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Для визначення оптимальних рішень під час розроблення системи слід виконати аналіз програмного забезпечення, яке вже застосовується у предметній

області. Метою такого огляду є вивчення досвіду провідних фірм-розробників та використання їхніх рішень для формування вимог до власного проєкту. Це сприятиме тому, що розроблюване програмне забезпечення буде відповідати сучасним потребам ринку. У процесі аналізу було детально розглянуто три популярні мобільні застосунки: Culinara, Kitchen Stories та Cookpad. Кожен з них має функціонал, схожий на запланований, але реалізує за різними підходами.

Першим розглянутим програмним продуктом є застосунок «Culinara: Easy Cooking Recipes». Його основним призначенням є надання користувачам доступу до перевірених кулінарних рецептів з акцентом на якісний відеоконтент та естетику подачі страв. Фірмою-розробником є українська компанія Culinara LLC. Основні інтерфейсні вікна застосунку побудовані навколо трьох розділів: головний екран, вікно пошуку та профіль користувача. Навігація орієнтована на перегляд страв від конкретних авторів-професіоналів, є розширена система фільтрації за інгредієнтами та часом приготування (Рисунок 1.5).

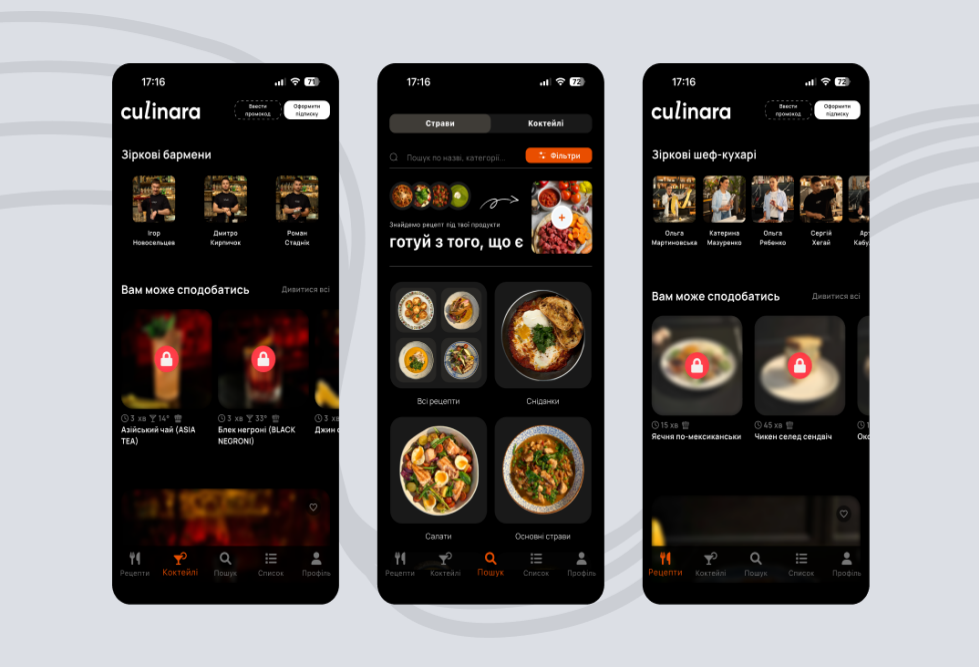


Рисунок 1.5 – Інтерфейс мобільного застосунку Culinara

До переваг цього аналогу можна віднести високу якість контенту, оскільки використовується модель закритої публікації від професійних редакторів, а також

наявність покрокового відеосупроводу. Недоліками застосунку є обмежений безкоштовний функціонал та відсутність можливості для звичайного користувача створювати та зберігати власні, повністю приватні рецепти.

Другим аналогом є застосунок «Kitchen Stories» (Рисунок 1.6). Його призначення полягає у наданні покрокових інструкцій для приготування їжі через високоякісні фотографії та відеоматеріали. Фірма-розробник – AJNS New Media GmbH (Німеччина). Інтерфейсні вікна включають детальну картку рецепта, де реалізовано важливу функцію автоматичного перерахунку кількості інгредієнтів при зміні числа порцій. Також застосунок має окреме вікно інтегрованого списку покупок та спеціальний режим готування, який не дозволяє екрану мобільного пристрою згаснути.

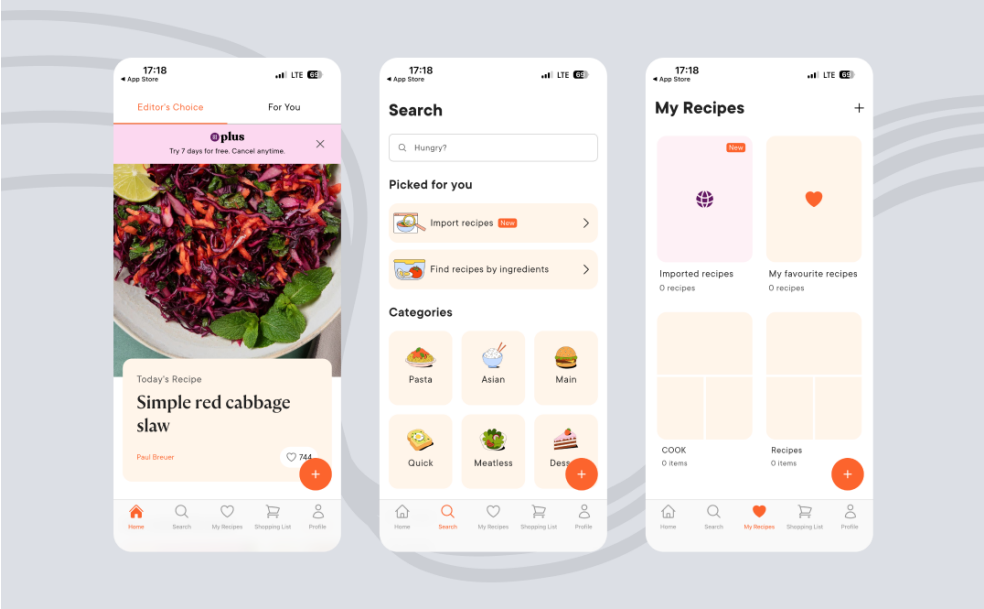


Рисунок 1.6 – Інтерфейс мобільного застосунку Kitchen Stories

Головними перевагами Kitchen Stories є еталонна реалізація користувацького інтерфейсу, зручні каруселі рекомендацій та продумана система тематичних колекцій від редакторів. Серед недоліків необхідно виділити перевантаженість інтерфейсу медіафайлами, що знижує швидкість роботи на старих пристроях, а також заскладну навігацію у розділі створення власних кулінарних записів.

Третім розглянутим продуктом є глобальна платформа «Cookpad» (фірма-розробник – Cookpad Inc., Японія) (Рисунок 1.7). Її призначення – функціонування як соціальної мережі для обміну домашніми рецептами, контент для якої створюють виключно самі користувачі. Інтерфейс застосунку орієнтований на взаємодію спільноти. Основні вікна дозволяють швидко опублікувати власний рецепт, залишити фотовідгук про приготування страви та шукати рецепти за наявними у холодильнику продуктами. Дозволяється зберігати страви у приватному режимі тільки для особистого перегляду.

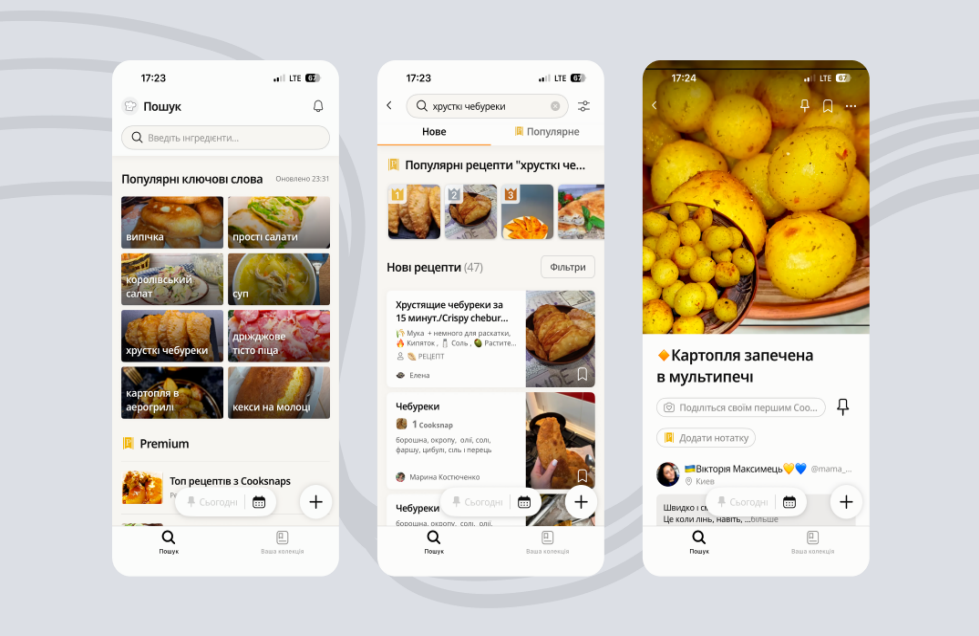


Рисунок 1.7 – Інтерфейс мобільного застосунку Cookpad

Перевагами Cookpad є простота додавання власних записів та грамотне розмежування прав: неавторизований гість може переглядати сторінки, але для збереження даних вимагається реєстрація. Суттєвим недоліком є загальний візуальний стиль, який часто страждає через низьку якість аматорських фотографій, а також перевантаженість екранів рекламними блоками.

Для більш наочного розуміння ситуації на ринку варто детально порівняти функціонал цих трьох застосунків із тим, що планується реалізувати у проєкті. Якщо розглядати тип контенту, то помітна суттєва різниця в підходах. Culinara та

Kitchen Stories роблять основну ставку на професійні рецепти від кухарів або редакції, хоча німецький застосунок також залишає простір для спільноти. Натомість Cookpad повністю базується на матеріалах, які створюють самі користувачі. Одним із найважливіших елементів інтерфейсу кулінарних програм є масштабування порцій. У Cookpad ця функція взагалі відсутня, оскільки автори просто публікують статичний текст. Culinara пропонує лише часткове вирішення цієї проблеми, і тільки Kitchen Stories має зручний автоматичний перерахунок інгредієнтів. Саме повноцінну автоматизацію цього процесу планується впровадити під час розроблення. Функція формування списків покупок тією чи іншою мірою присутня у всіх трьох розглянутих продуктах. Найкраще вона реалізована у Kitchen Stories, де продукти зручно групуються. Цей досвід доцільно використати для тісної інтеграції списків із самими рецептами у новій системі. Щодо колекцій страв, то в конкурентів вони зазвичай формуються або виключно редакторами, або тільки самими користувачами.

Використання помічника на базі інструментів штучного інтелекту є слабким місцем наявного програмного забезпечення. Kitchen Stories та Cookpad не мають такого функціоналу, а Culinara використовує лише базові алгоритми для персоналізації рекомендацій. Тому розроблення спеціального модуля для генерації контенту розглядається як одна з головних конкурентних переваг майбутньої розробки. Такий модуль дозволить не лише створювати нові рецепти на основі наявних у користувача продуктів, але й автоматично адаптувати їх під індивідуальні дієтичні обмеження та харчові вподобання. Це суттєво знизить когнітивне навантаження на етапі планування меню.

Виконаний аналіз показує, що хоча на ринку існують потужні гравці, жоден з них не покриває концепцію майбутнього програмного продукту повністю. Доцільно взяти за основу візуальні рішення та логіку публічної частини з Kitchen Stories, а підходи до приватності та організації збережених страв запозичити у Cookpad. Інтеграція цих перевірених часом рішень із новітніми хмарними технологіями дозволить створити єдине безшовне середовище. Об'єднання цих функцій у єдиному мобільному застосунку робить його розроблення доцільним та актуальним.

1.3 Визначення функціональних та нефункціональних вимог до програмного забезпечення

Для того, щоб майбутній застосунок повністю відповідав очікуванням користувачів та вирішував описані раніше проблеми, необхідно сформулювати точний перелік вимог до його роботи [7]. На цьому етапі проєктування увага зосереджується на тому, які саме функції має виконувати система, а питання їх технічного чи програмного втілення залишається для наступних розділів. Щоб формалізувати ці вимоги, в інженерії програмного забезпечення застосовують візуальне моделювання за допомогою уніфікованої мови UML [8, 9] згідно зі специфікацією консорціуму Object Management Group (OMG), UML є стандартизованою мовою моделювання загального призначення, що надає набір графічних позначень для створення візуальних моделей програмних систем [10].

Проєктування зазвичай починається з побудови моделі варіантів використання. Ця модель допомагає зрозуміти, хто саме буде користуватися програмою і що ці люди зможуть у ній робити. Зважаючи на специфіку кулінарного застосунку, логічно розділити всю аудиторію на три окремі ролі. Детальний опис цих ролей (акторів) наведено у таблиці 1.2.

Таблиця 1.2 – Детальний опис користувачів системи

Роль	Короткий опис інформаційних потреб та прав доступу
Гість	Це людина, яка щойно завантажила застосунок або ще не створила свій обліковий запис. Такий підхід, відомий як відкладена реєстрація, дозволяє значно знизити поріг входу та залучити більшу кількість нової аудиторії Її взаємодія з програмою зводиться до режиму читання. Гість може гортати стрічку рецептів на головній сторінці, читати рекомендації від редакторів та користуватися пошуком. Проте, як тільки гість спробує зберегти рецепт, поставити оцінку або відкрити вкладку своїх записів, система блокує цю дію і пропонує авторизуватися.

Продовження таблиці 1.2

Користувач	Основний учасник системи, для якого створюється більшість функцій. Авторизована особа отримує доступ до всіх п'яти розділів меню. Користувач може формувати власну бібліотеку рецептів, об'єднувати їх у колекції, зберігати фотографії страв із підписами. Окрім того, він може користуватися списком покупок, куди інгредієнти додаються автоматично, та генерувати нові приватні рецепти за допомогою вбудованого помічника.
Адміністратор	Службова особа, відповідальна за наповнення застосунку якісним матеріалом. Адміністратор має доступ до спеціальної панелі, де він створює та публікує загальнодоступні рецепти. Також він веде базу профілів шеф-кухарів (додає їхні імена, фото та посилання). Важливо, що самі шеф-кухарі не мають доступу до системи як окремі користувачі – їхніми сторінками та рецептами повністю керує адміністратор.

Після визначення кола користувачів потрібно окреслити перелік сервісів, які їм надаватимуться. Кожен такий сервіс називається варіантом використання. Основні з них зібрані у таблиці 1.3.

Таблиця 1.3 – Опис основних варіантів використання застосунку

Актор	Найменування варіанта використання	Опис варіанта використання
Гість, Користувач	Реєстрація та авторизація	Дозволяє гостю створити профіль, щоб отримати доступ до персональних функцій збереження та оцінювання
Користувач	Збереження публічного рецепта	Користувач зберігає рецепт від редактора до свого списку улюблених страв для швидкого доступу в майбутньому.

Продовження таблиці 1.3

Користувач	Робота з приватними рецептами	Створення власних кулінарних нотаток, які ніхто інший не бачить. Сюди ж входить генерація рецепта помічником.
Користувач	Управління списком покупок	Перенесення інгредієнтів з рецепта до зручного переліку для походу в магазин та групування куплених продуктів.
Гість, Користувач	Перегляд та пошук страв	Робота з публічною базою: застосування фільтрів, перегляд карток страв та профілів їхніх авторів (шеф-кухарів).
Адміністратор	Керування контентом бази	Наповнення системи новими статтями, створення колекцій та управління інформацією про зареєстрованих осіб.

Щоб наочно показати, як саме актори взаємодіють із системою під час виконання цих завдань, розроблено відповідну діаграму (Рисунок 1.8). Відповідно до технічної документації IBM, діаграма варіантів використання описує сукупність дій (варіантів використання), які система може виконувати у взаємодії з одним або декількома зовнішніми користувачами (акторами), і є ключовим інструментом для визначення функціональних вимог до системи [11].

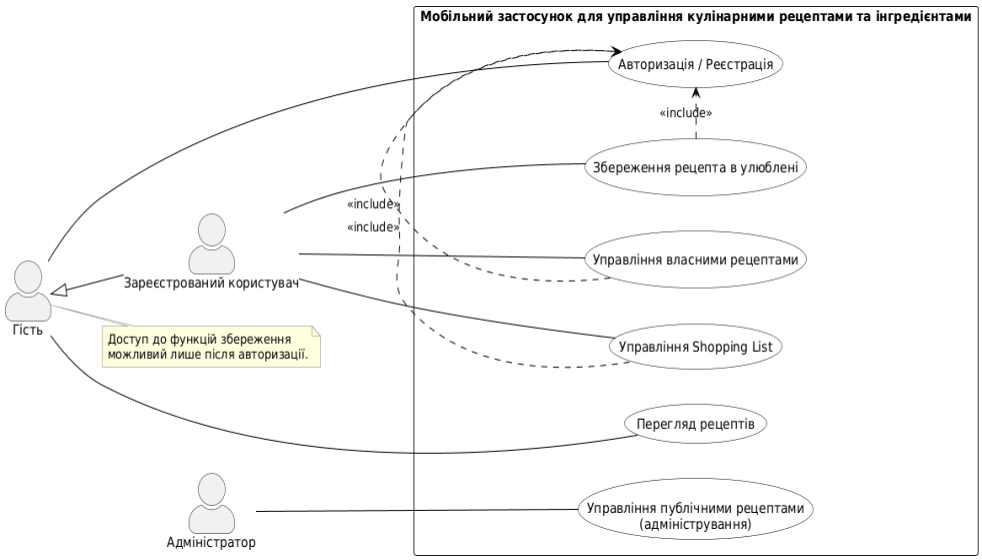


Рисунок 1.8 – Діаграма варіантів використання

Для того, щоб під час програмування чітко розуміти логіку роботи програми в різних ситуаціях, необхідно деталізувати ключові варіанти використання у вигляді текстових специфікацій. Вони описують, що відбувається до, під час та після виконання певної дії.

Специфікація прецеденту «Реєстрація та авторизація». Передумовою виконання даного прецеденту є стан, коли особа відкрила застосунок і має статус гостя. Дія, що запускає сценарій, полягає у натисканні на кнопку профілю або спробі додати вподобану страву до розділу улюбленого. Основний сценарій виконання передбачає наступні кроки:

– система зупиняє поточну дію і виводить на екран форму для входу;

– гість вводить свої дані (електронну пошту та пароль) або обирає реєстрацію;

– застосунок відправляє запит на сервер бази даних для перевірки;

– сервер підтверджує дані і повертає застосунку дозвіл на доступ;

– система оновлює статус з гостя на авторизованого користувача. В межах альтернативного сценарію, якщо гість вводить неправильний пароль, система не пропускає його далі, виводить текстове повідомлення про помилку та очищає поле вводу для повторної спроби.

Специфікація прецеденту «Управління списком покупок». Для реалізації цього функціоналу користувач має бути авторизований і знаходитися на екрані детального опису рецепта. Сценарій запускається натисканням кнопки «Додати інгредієнти у список». Процес виконання включає такі етапи:

– користувач на екрані рецепта збільшує кількість порцій (наприклад, з двох до чотирьох);

– система автоматично перераховує обсяг кожного продукту пропорційно до нових порцій;

– користувач натискає кнопку додавання;

– програма копіює перераховані продукти і переносить їх у розділ Список покупок», групуючи їх за відповідними категоріями. Як альтернативний сценарій передбачено можливість не заходити в рецепт, а відкрити розділ

покупок і вручну ввести назву необхідного продукту. Післяумовою є поява нових позицій у списку покупок, які користувач може відмічати як придбані.

Специфікація прецеденту «Робота з приватними рецептами». Передумовою є перехід авторизованого користувача у розділ «Мої рецепти» до підрозділу «Приватні». Сценарій активується вибором опції створення нового запису. Послідовність дій системи:

– відкривається форма, де користувач додає власні фотографії, назву, інгредієнти та кроки приготування;

– після натискання кнопки збереження система створює в базі даних новий документ;

– документу присвоюється спеціальний маркер приватності, який гарантує доступ до нього лише автору. Альтернативний сценарій передбачає залучення вбудованого помічника для генерації рецепта. Програма створює текст страви і автоматично присвоює запису логічну змінну (isAI), що відображається спеціальним значком в інтерфейсі. Післяумовою є відображення нового запису у приватній бібліотеці користувача.

Після визначення сценаріїв роботи можна скласти зведений перелік вимог до програмного забезпечення. Їх поділяють на функціональні та нефункціональні. Функціональні вимоги до програмного забезпечення. Вся навігація застосунку будується навколо п'яти основних розділів, кожен з яких має свої чіткі функціональні вимоги:

а) Розділ «Головна сторінка»:
   1) система повинна виводити нотифікації для користувача;
   2) програма має відображати блоки з рекомендаціями від редакторів (наприклад, рубрика «Рецепт дня»);
   3) повинна бути реалізована стрічка з переліком доступних рецептів у вигляді карток.

б) Розділ «Пошук»:
   1) забезпечення можливості текстового пошуку як за окремими стравами, так і за цілими колекціями;

2) наявність інструментів фільтрування результатів (за часом, складністю тощо) та їх сортування.

в) Розділ «Мої рецепти». Цей розділ є найскладнішим і повинен включати п'ять окремих підрозділів. Система має зберігати публічні рецепти, які вподобав користувач, та цілі улюблені колекції. Крім того, повинна бути можливість створювати власні папки (колекції) і групувати в них збережені страви. Окремою вимогою є створення галереї для збереження фотографій або знімків екрана з можливістю додавати до них текстові підписи. Найважливіша функція цього розділу – створення та зберігання приватних рецептів, які фізично недоступні іншим людям. Тут же має бути передбачена кнопка або інтерфейс для відправки свого рецепта на модерацію адміністратору, щоб зробити його публічним.

г) Розділ «Список покупок»:

1) застосунок повинен вміти приймати список продуктів напряму з відкритого рецепта;
2) інгредієнти мають автоматично підлаштовуватися під ту кількість порцій, яку користувач виставив на екрані;
3) програма повинна групувати додані продукти за їхнім типом для зручності покупок;
4) користувач повинен мати змогу вручну вводити будь-які інші товари до цього списку.

д) Додатковий та адміністративний функціонал. Для реалізації модуля штучного інтелекту планується інтеграція стороннього API. Відповідно до технічної документації OpenAI, їхній програмний інтерфейс дозволяє розробникам вбудовувати моделі обробки природної мови безпосередньо у свої застосунки для виконання складних завдань, таких як генерація, переклад та структурування текстового контенту [12]. Для адміністратора повинна бути реалізована панель керування, де він створює добірки, переглядає список користувачів і додає шеф-кухарів. Профіль шеф-кухаря створюється як запис у базі, з якого користувачі можуть переглядати всі страви конкретного автора або

переходити на його соціальні мережі. На екрані рецепта від шефа система повинна вміти генерувати карусель з трьох інших випадкових страв із колекції.

Крім набору функцій, система повинна відповідати ряду технічних умов, щоб її використання було безпечним і комфортним. Інтерфейс програми повинен створюватися з урахуванням офіційних принципів проєктування Human Interface Guidelines (HIG) [13]. Цей нормативний документ від розробника операційної системи регламентує правила побудови інтуїтивно зрозумілої навігації, використання системних шрифтів та адаптації контенту під різні розміри екранів [14]. Для розроблення користувацького інтерфейсу доцільно застосувати сучасний фреймворк SwiftUI. Згідно з офіційною документацією Apple Developer, SwiftUI надає декларативний синтаксис та інструменти для створення інтерфейсів, які автоматично підтримують системні функції, такі як динамічне масштабування тексту та темна тема оформлення [15].

Увесь масив інформації, включаючи бази даних користувачів, рецептів та фотографій, повинен надійно зберігатися у хмарному сховищі (на базі сервісів Firebase). Механізми ідентифікації повинні надійно захищати облікові записи від несанкціонованого входу. Для цього в архітектурі системи застосовується криптографічний захист поточних сесій на базі безпечних JWT-токенів хмарного сервісу Firebase Auth. Це повністю унеможливлює пряме перехоплення автентифікаційних даних. Застосунок має працювати стабільно навіть при завантаженні великої кількості зображень високої якості. Крім того, програма повинна коректно реагувати на раптову втрату з'єднання з мережею Інтернет, зберігаючи введені користувачем дані та повідомляючи його про відсутність зв'язку замість раптового завершення роботи.

1.4 Висновки. Постановка задачі

У цьому розділі було детально проаналізовано предметну область. Аналіз показав, що сфера організації домашнього харчування та збереження кулінарних

рецептів потребує сучасних та зручних програмних рішень. Наявні на ринку популярні продукти (Culinara, Kitchen Stories, Cookpad) пропонують потужний функціонал, але жоден з них не поєднує в собі одночасний доступ до професійного кураторського контенту від шеф-кухарів із середовищем для ведення повністю приватних кулінарних записів. Крім того, більшість наявних рішень не повною мірою автоматизують такі повсякденні потреби, як динамічний перерахунок кількості інгредієнтів під час зміни порцій або швидке формування списків покупок прямо з екрана рецепта.

На основі виявлених недоліків існуючих систем було сформовано чіткі вимоги до нового програмного забезпечення. Визначено три основні ролі (гість, зареєстрований користувач та адміністратор) і побудовано модель варіантів використання для опису їхньої взаємодії з програмою. Також формалізовано ключові функціональні та нефункціональні вимоги, які враховують специфіку операційної системи iOS та використання хмарних баз даних.

Головною метою кваліфікаційної роботи є розроблення мобільного застосунку для управління кулінарними рецептами та інгредієнтами, яке дозволяє підвищити зручність організації рецептів, автоматизувати створення списків покупок та забезпечити користувачам доступ до якісного контенту.

Для досягнення поставленої мети в наступних розділах необхідно вирішити такі задачі проєктування:

– спроєктувати загальну архітектуру програмного забезпечення та логічну структуру нереляційної бази даних;

– розробити користувацький інтерфейс із дотриманням офіційних гайдлайнів проєктування під мобільні пристрої Apple;

– виконати програмну реалізацію алгоритмів та логіки;

– інтегрувати спеціальний модуль штучного інтелекту;

– налаштувати систему ідентифікації та розробити правила бази даних;

– провести тестування готового застосунку для перевірки його стабільності, швидкості завантаження мультимедійного контенту та коректності роботи базового функціоналу.

2 ПРОЄКТУВАННЯ

2.1 Архітектура та функціональна структура

Архітектура мобільного застосунку побудована за клієнт-серверним принципом [16, 17]. Для серверної частини обрано модель хмарного бекенду. Клієнтська частина реалізована за патерном MVVM. Цей патерн розділяє код на три незалежні частини: графічний інтерфейс, бізнес-логіку управління станом і структури даних. Таке розділення спрощує масштабування коду та дозволяє тестувати логіку окремо від інтерфейсу.

Систему поділено на три рівні:

– перший рівень відповідає за роботу клієнтського додатка. Він збирає вхідні дані, формує запити до сторонніх сервісів і відображає результати на екрані пристрою. Вся бізнес-логіка цього рівня знаходиться у відповідних класах управління станом;

– другий рівень становить хмарна інфраструктура Firebase. Вона виконує роль серверної частини та зберігає поточний стан системи. Для реєстрації та авторизації застосовується сервіс Firebase Authentication. Дані про згенеровані рецепти, списки покупок та налаштування профілів зберігаються у нереляційній базі даних Cloud Firestore. Завантажені користувачами зображення та аватари розміщуються у хмарному сховищі Firebase Storage:

– третій рівень це зовнішній сервіс OpenAI API. Він отримує текстові запити від мобільного клієнта, обробляє список інгредієнтів за допомогою мовної моделі та повертає готовий рецепт у структурованому форматі.

Логіку застосунку розбито на кілька незалежних модулів.

– модуль авторизації відповідає за екрани логіну та реєстрації. Він перевіряє введені користувачем дані та передає їх до бази даних для створення безпечної сесії;

– модуль управління рецептами є основною частиною програми. Він виводить список збережених страв, показує їхні деталі та містить форму для

створення нових запитів. Цей модуль безпосередньо звертається до зовнішнього API, отримує згенерований рецепт і зберігає його в базу. Також тут налаштовано поділ рецептів на приватні та публічні;

– модуль пошуку та рекомендацій відповідає за фільтрацію контенту. Він дозволяє шукати страви за інгредієнтами і формує стрічку публічних рецептів на головному екрані;

– модуль списків покупок передбачено для роботи з інвентарем. Він автоматично додає відсутні продукти зі згенерованого рецепта до списку. Користувач також може вручну додавати, редагувати або видаляти продукти, а всі зміни одразу синхронізуються з хмарною базою даних;

– модуль профілю та навігації керує переходами між екранами і дозволяє змінювати налаштування облікового запису;

– модуль адміністрування існує для користувачів із розширеними правами. Він дає змогу модерувати контент, створювати публічні добірки страв і керувати профілями інших користувачів.

Процес генерації рецепта починається з введення інгредієнтів на екрані програми. Логічний блок передає ці дані сервісу генерації тексту, а отриманий результат конвертується у внутрішній формат і записується у хмарну базу даних. Після успішного збереження інтерфейс оновлюється автоматично. Схематичне подання загальної архітектури та функціональної структури мобільного застосунку наведено на рисунку 2.1.

Модель хмарного бекенду на базі Firebase усуває потребу налаштування та адміністрування власного сервера. База даних Cloud Firestore підтримує постійне з'єднання з пристроєм, тому будь-які зміни відображаються миттєво без ручного перезавантаження сторінок. Патерн MVVM ізолює графічну розмітку від логіки обробки даних, спрощуючи оновлення дизайну та пошук помилок у коді. Недоліком архітектури є повна залежність від стабільності серверів Google та OpenAI. Будь-які технічні збої чи перевищення лімітів на стороні REST API стороннього сервісу штучного інтелекту блокують процес генерації нових кулінарних інструкцій.

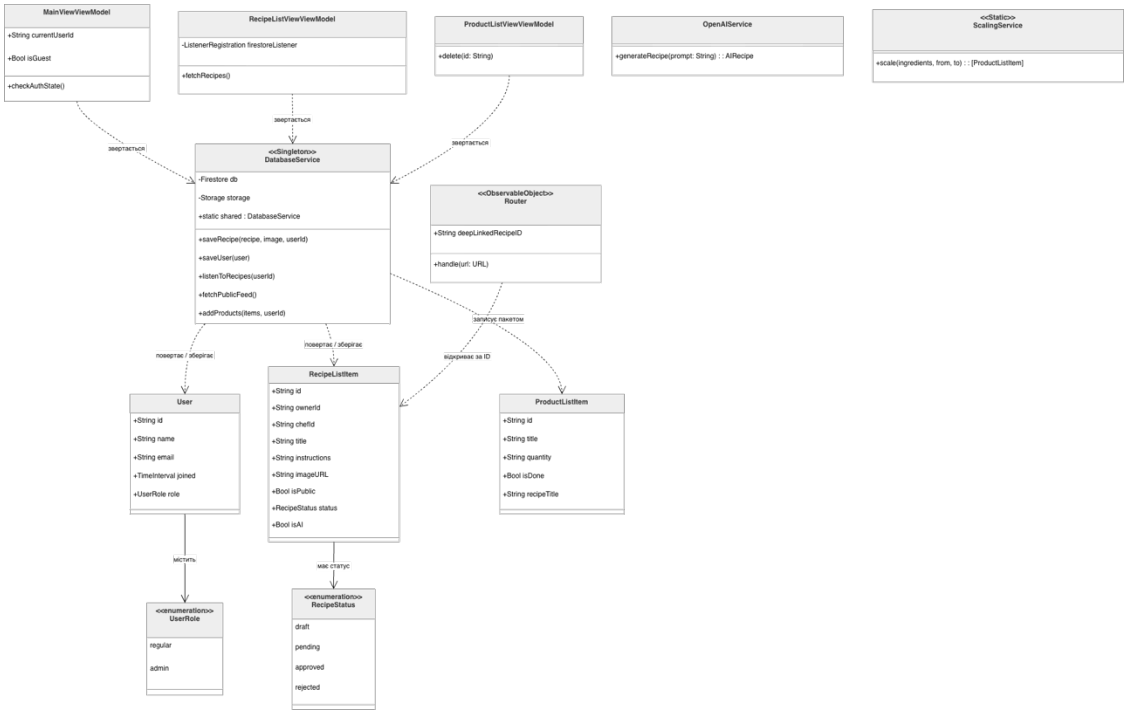


Рисунок 2.1 – Діаграма «сутність-зв'язок»

Без підключення до мережі відправка запитів на генерацію або синхронізація даних профілю стають недоступними. Загальну схему системи наведено на рисунку 2.2.

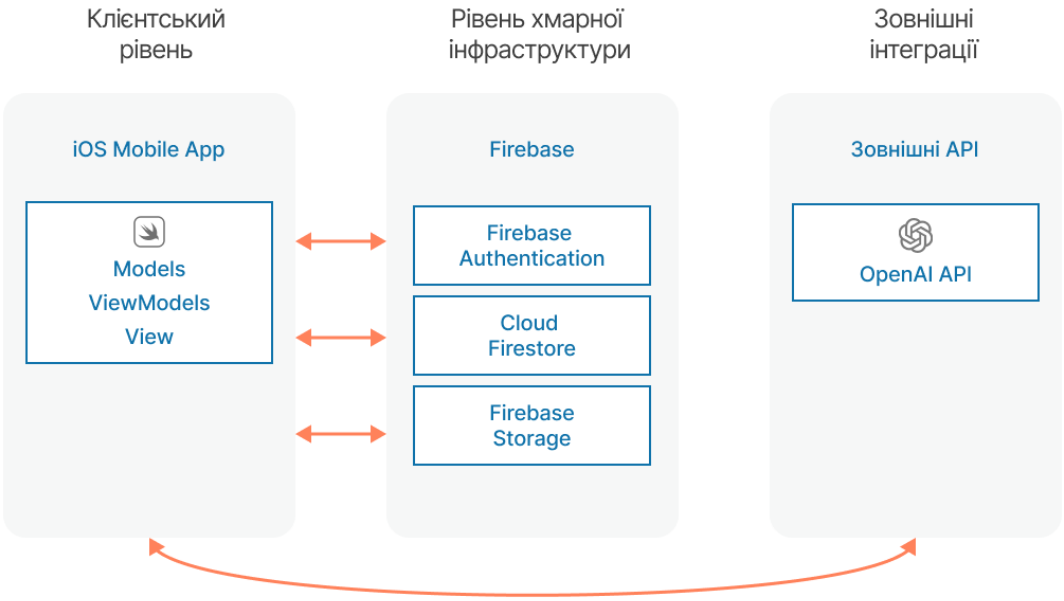


Рисунок 2.2 – Загальна архітектура та функціональна структура

2.2 Проєктування структури бази даних

База даних є інструментом для збирання та впорядкування інформації [18]. Сучасні системи управління базами даних загалом поділяються на дві основні категорії [19]. Реляційні бази даних зберігають інформацію у вигляді строгих таблиць із чітко визначеними зв'язками. Нереляційні бази даних використовують гнучкі моделі. Вони зберігають дані у форматі документів або пар ключ-значення. Для розробки серверної частини мобільного застосунку використано платформу Firebase. Це набір хмарних сервісів для створення програмного забезпечення [20]. Платформа надає безпосередньо готові інструменти для авторизації, зберігання файлів та роботи з даними.

Основним сховищем даних у проєкті є Cloud Firestore. Це гнучка нереляційна база даних, яка синхронізує інформацію між клієнтськими пристроями в реальному часі. Дані у Firestore зберігаються у вигляді документів, які об'єднуються у колекції. Клієнтський код застосунку написаний мовою Swift. Взаємодія з базою даних відбувається через протокол Codable. Цей протокол дозволяє системі автоматично перетворювати документи Firestore у нативні структури коду. Такий підхід усуває потребу ручного оброблення тексту під час завантаження даних з сервера. Загальна логічна структура бази даних складається з п'яти основних колекцій. Першою є колекція Users. Вона відповідає за збереження базової інформації про облікові записи (Таблиця 2.1).

Таблиця 2.1 – Опис колекції Users

Атрибут	Опис
id	Унікальний ідентифікатор облікового запису.
email	Електронна пошта профілю.
role	Рівень доступу профілю. Може приймати значення гостя, звичайного користувача або адміністратора.

Основний масив даних застосунку зберігається у колекції Recipes. Ця колекція містить усі створені та згенеровані страви. Рецепти можуть належати звичайним користувачам або професійним кухарям (Таблиця 2.2).

Таблиця 2.2 – Опис колекції Recipes

Атрибут	Опис
id	Унікальний ідентифікатор рецепта.
ownerId	Посилання на користувача для приватних рецептів.
chefId	Посилання на шеф-кухаря для публічних страв.
title	Назва страви.
ingredients	Масив об'єктів для зберігання необхідних продуктів.
instructions	Текстовий опис процесу приготування страви.
category	Категорія страви згідно з визначеним переліком.
difficulty	Рівень складності приготування.
imageURL	Посилання на завантажену фотографію у хмарному сховищі.
createdDate	Час і дата створення запису в базі даних.
isPublic	Маркер для визначення видимості рецепта для інших осіб.
isFavorite	Маркер додавання страви до списку улюблених.

Наступною колекцією є ShoppingLists. Вона відповідає за зберігання інформації про інвентар та необхідні для приготування продукти. Кожен список прив'язується до конкретного облікового запису (Таблиця 2.3).

Таблиця 2.3 – Опис колекції ShoppingLists

Атрибут	Опис
id	Ідентифікатор списку покупок.
userId	Ідентифікатор профілю власника списку.
items	Масив продуктів та їхній поточний статус придбання.

Користувачі мають змогу зберігати рецепти та об'єднувати їх у групи. Для цього використовується колекція UserCollections. Вона допомагає організувати збережені страви у тематичні папки (Таблиця 2.4).

Таблиця 2.4 – Опис колекції UserCollections

Атрибут	Опис
id	Ідентифікатор тематичної добірки.
ownerId	Ідентифікатор власника папки.
title	Назва створеної папки.
recipeIds	Масив ідентифікаторів доданих до добірки рецептів.

Для публічних рецептів від професіоналів створено окрему колекцію Chefs. Вона дозволяє розмежувати звичайні профілі та акаунти експертів. Документи цієї колекції містять публічну інформацію про кухарів (Таблиця 2.5).

Таблиця 2.5 – Опис колекції Chefs

Атрибут	Опис
id	Ідентифікатор професійного кухаря.
name	Ім'я шеф-кухаря.
photoURL	Посилання на профільну фотографію.
bio	Коротка біографія автора страв.
socialMediaLinks	Об'єкт із посиланнями на соціальні мережі кухаря.

Окрему увагу приділено механізму поширення рецептів між користувачами. Для цього у структурі даних передбачено генерацію глибоких посилань. На клієнтській стороні обчислюється спеціальна властивість publicURL. Вона формує посилання формату cookbook://recipe?id=. На місце останнього параметра підставляється унікальний ідентифікатор конкретного рецепта. Цей механізм дозволяє відкривати потрібну страву безпосередньо у

мобільному застосунку. Перехід відбувається автоматично після натискання на посилання в інших програмах або месенджерах. Діаграма сутність-зв'язок є візуальним представленням логічної структури бази даних. Вона ілюструє сутності та відносини між ними [21]. Схематичне подання колекцій бази даних застосунку наведено на рисунку 2.3.

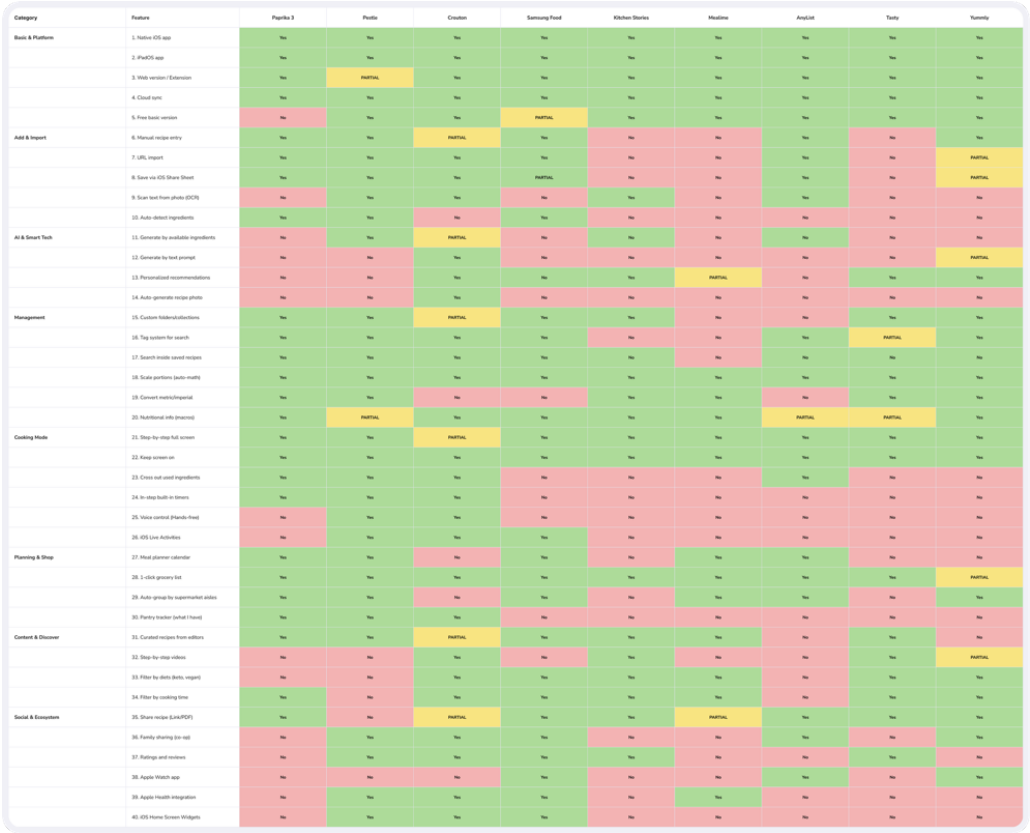
Рисунок 2.3 – ER-діаграма

2.3 Проєктування інтерфейсу користувача

Проєктування інтерфейсу користувача та досвіду взаємодії є обов'язковим етапом розроблення мобільного застосунку. Цей етап визначає структуру діалогу між кінцевим користувачем і пристроєм. Візуальні та навігаційні рішення проєкту базуються на офіційних принципах Apple Human Interface Guidelines [22]. Дотримання цих настанов гарантує нативний вигляд застосунку для

платформи операційної системи. Макети спроєктовано під можливості актуальної версії операційної системи iOS 26. У проєкті використано системні шрифти San Francisco та базові принципи динамічного масштабування тексту. Інтерфейс також підтримує системний темний режим відображення. Основним інструментом для створення візуалу обрано середовище Figma [23, 24].

Перед створенням макетів проведено дослідження користувацького досвіду. Для розуміння ринку проаналізовано стратегії дев'яти конкурентів за сорока різними параметрами. Результати порівняльного аналізу зведено у відповідну таблицю (Рисунок 2.4).



Category	Feature	Paprika 3	Pestle	Crouton	Samsung Food	Kitchen Stories	Mealime	AnyList	Tasty	Yummly
Basic & Platform	1. Native iOS app	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	2. iPadOS app	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	3. Web version / Extension	Yes	PARTIAL	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	4. Cloud sync	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	5. Free basic version	No	Yes	Yes	PARTIAL	Yes	Yes	Yes	Yes	Yes
Add & Import	6. Manual recipe entry	Yes	Yes	PARTIAL	Yes	No	No	Yes	No	Yes
	7. URL import	Yes	Yes	Yes	Yes	No	No	Yes	No	PARTIAL
	8. Save via iOS Share Sheet	Yes	Yes	Yes	PARTIAL	No	No	Yes	No	PARTIAL
	9. Scan text from photo (OCR)	No	Yes	Yes	No	Yes	No	Yes	No	No
	10. Auto-detect ingredients	Yes	Yes	No	Yes	No	No	No	No	No
AI & Smart Tech	11. Generate by available ingredients	No	Yes	PARTIAL	No	No	No	No	No	No
	12. Generate by text prompt	No	No	Yes	No	No	No	No	No	PARTIAL
	13. Personalized recommendations	No	No	Yes	No	Yes	PARTIAL	No	Yes	Yes
	14. Auto-generate recipe photo	No	No	Yes	No	No	No	No	No	No
Management	15. Custom folders/collections	Yes	Yes	PARTIAL	Yes	Yes	No	No	Yes	Yes
	16. Tag system for search	Yes	Yes	Yes	Yes	No	No	Yes	PARTIAL	Yes
	17. Search inside saved recipes	Yes	Yes	Yes	Yes	No	No	No	No	No
	18. Scale portions (auto-math)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	19. Convert metric/imperial	Yes	Yes	No	No	Yes	Yes	No	No	Yes
	20. Nutritional info (macros)	Yes	PARTIAL	Yes	Yes	Yes	Yes	PARTIAL	PARTIAL	Yes
Cooking Mode	21. Step-by-step full screen	Yes	Yes	PARTIAL	Yes	Yes	Yes	Yes	Yes	Yes
	22. Keep screen on	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	23. Cross out used ingredients	Yes	Yes	Yes	No	No	No	Yes	No	No
	24. In-step built-in timers	Yes	Yes	Yes	No	No	No	No	No	No
	25. Voice control (Hands-free)	No	Yes	Yes	No	No	No	No	No	No
	26. iOS Live Activities	No	Yes	Yes	Yes	No	No	No	No	No
Planning & Shop	27. Meal planner calendar	Yes	Yes	No	Yes	No	Yes	Yes	No	No
	28. 1-click grocery list	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	PARTIAL
	29. Auto-group by supermarket aisles	Yes	Yes	No	Yes	No	Yes	Yes	No	Yes
	30. Pantry tracker (what I have)	Yes	Yes	Yes	No	No	No	No	No	No
Content & Discover	31. Curated recipes from editors	Yes	Yes	PARTIAL	Yes	Yes	Yes	No	Yes	No
	32. Step-by-step videos	No	No	Yes	No	Yes	No	No	Yes	PARTIAL
	33. Filter by diets (keto, vegan)	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes
	34. Filter by cooking time	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes
Social & Ecosystem	35. Share recipe (Link/PDF)	Yes	No	PARTIAL	Yes	Yes	PARTIAL	Yes	Yes	Yes
	36. Family sharing (co-op)	No	Yes	Yes	Yes	No	No	Yes	No	Yes
	37. Ratings and reviews	No	Yes	Yes	Yes	Yes	No	No	No	No
	38. Apple Watch app	No	No	No	Yes	No	No	Yes	No	Yes
	39. Apple Health integration	No	Yes	Yes	Yes	No	Yes	No	No	No
	40. iOS Home Screen Widgets	No	Yes	Yes	Yes	No	No	No	No	No

Рисунок 2.4 – Порівняльний аналіз конкурентів

Для оцінки результатів використано методологію дослідницької компанії Nielsen Norman Group [25]. На основі аналізу визначено точки подібності та точки відмінності. Точки подібності дозволили виявити обов'язкові функції та загальноприйняті навігаційні схеми. Впровадження знайомих схем знижує

когнітивне навантаження на користувача. Усі макети пройшли перевірку на зручність використання однією рукою. Екран авторизації та реєстрації має мінімалістичний вигляд. На екрані розташовані поля для введення імені, електронної пошти та пароля. Для зареєстрованих користувачів передбачено перемикач запам'ятовування сесії (Рисунок 2.5).

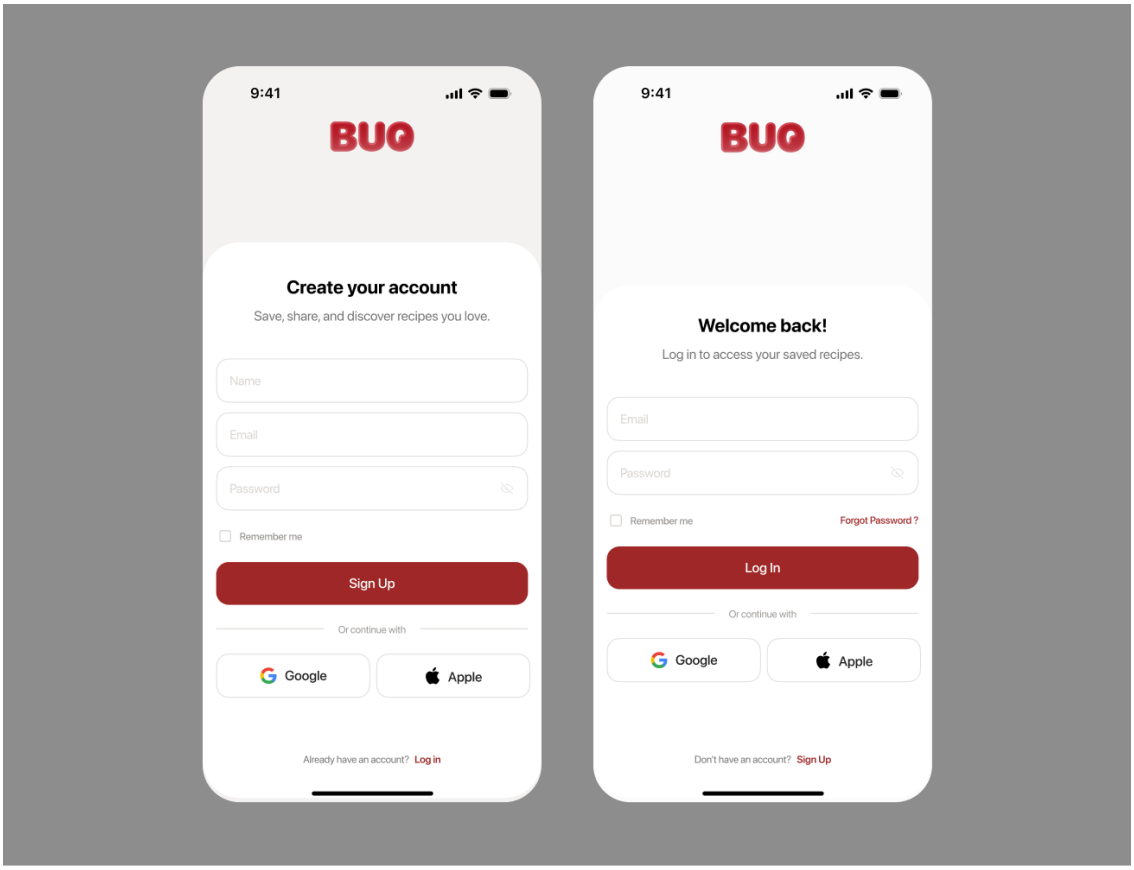


Рисунок 2.5 – Екран авторизації та реєстрації користувача

Головний екран виконує роль вітрини застосунку. Його побудовано за принципом вертикального прокручування. Верхня частина містить персоналізоване привітання та кнопку сповіщень. Далі розташовано велику фотографію рекомендованої страви з кнопкою початку приготування та іконкою збереження. Нижче розміщено розділ персональних рекомендацій. Картки страв містять візуальні теги з інформацією про час приготування та дієтичні особливості. У нижній частині екрана закріплено панель навігації для переходу між основними розділами програми (Рисунок 2.6).

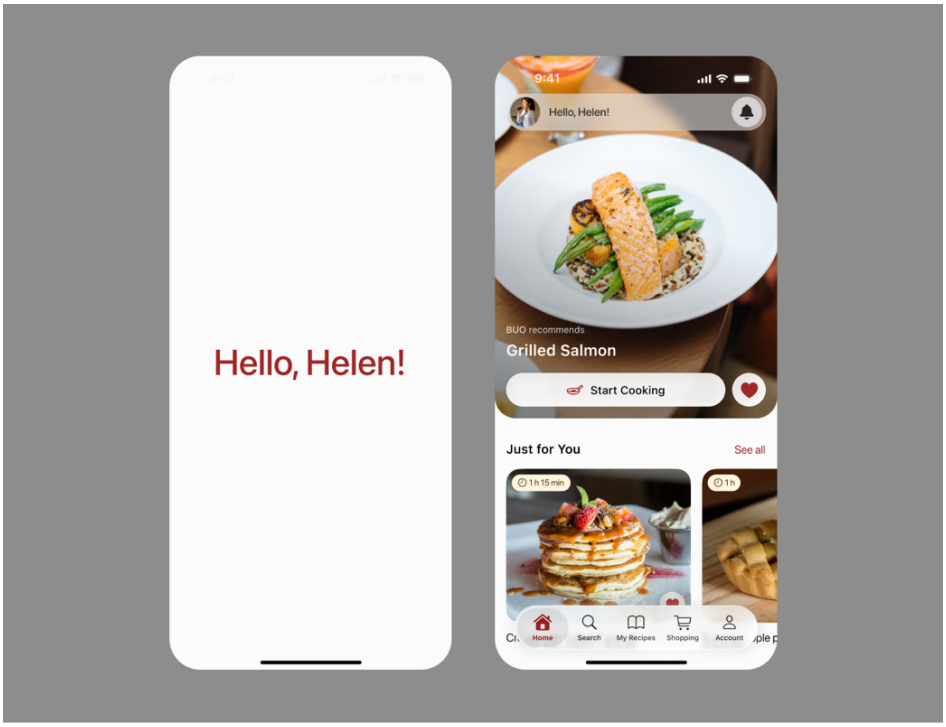


Рисунок 2.6 – Макет заставки та головного екрана застосунку

Інтерфейс екрана пошуку створено для швидкого знаходження контенту. У верхній частині знаходиться текстовий рядок із можливістю голосового вводу та кнопкою виклику панелі фільтрів (Рисунок 2.7).

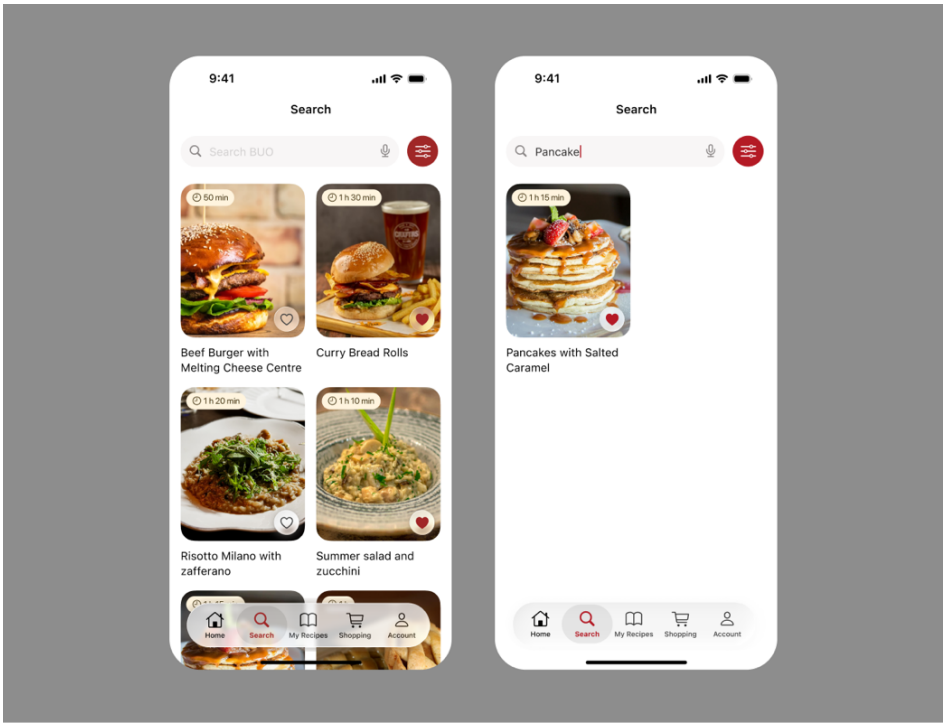


Рисунок 2.7 – Екран пошуку та фільтрації рецептів

Сторінка деталей рецепта відображає всю необхідну інформацію для приготування. У верхній частині розміщено головне зображення страви та кнопки управління контентом. Під фотографією вказано назву, рейтинг, загальний час приготування, рівень складності та кількість порцій. Блок інгредієнтів має перемикачі для динамічної зміни кількості порцій. Вага інгредієнтів оновлюється під час зміни цього параметра (Рисунок 2.8).

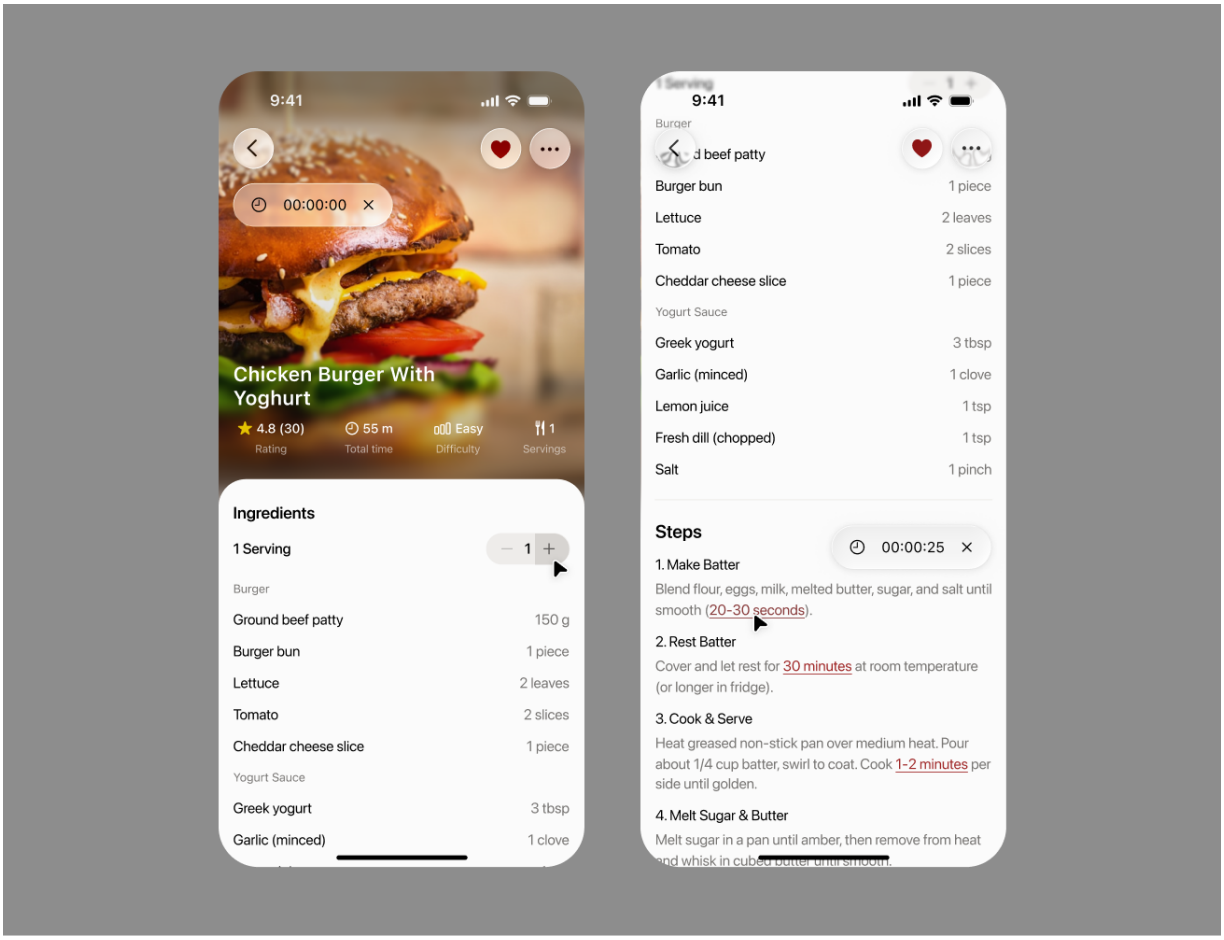


Рисунок 2.8 – Інтерфейс сторінки деталей рецепта

Розділ власних рецептів побудовано з використанням великих кольорових блоків. Кожен елемент адаптований під нативні жести iOS для швидкого переходу між категоріями. З цього екрана користувач може перейти до збережених закладок, загального списку власних рецептів, створених колекцій або рецептів у форматі фотографій. Екран колекцій відповідає за організацію контенту. Користувач бачить створені тематичні папки зі стравами (Рисунок 2.9).

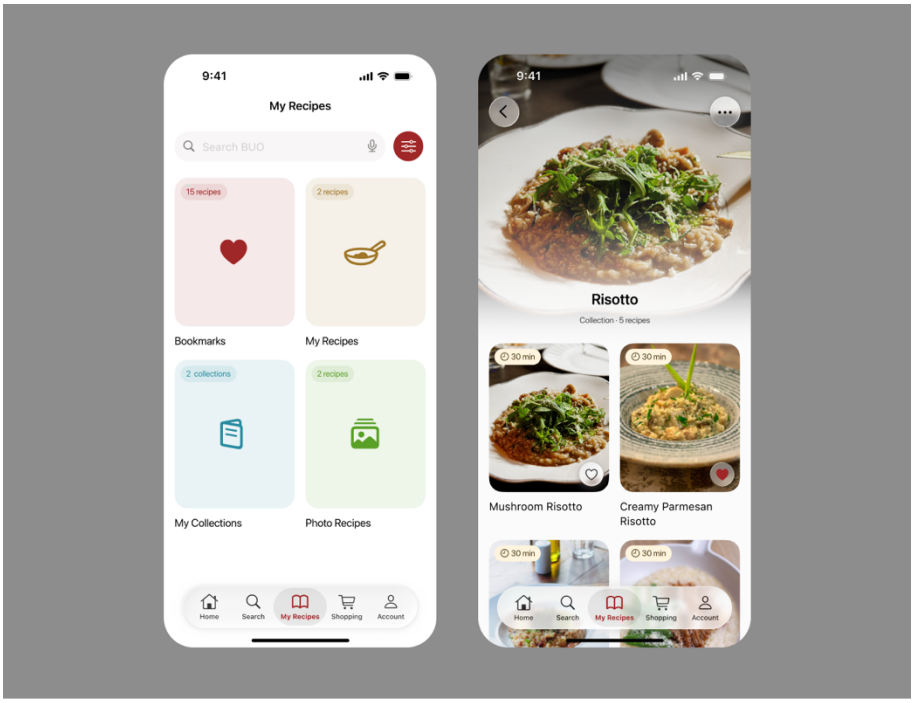


Рисунок 2.9 – Екран організації користувача та екран колекції

Екран списку покупок допомагає керувати необхідними для приготування продуктами. Інгредієнти групуються за рецептами для зручності орієнтування у списку. Додавання продуктів до списку можливе з картки рецепта (Рисунок 2.10).

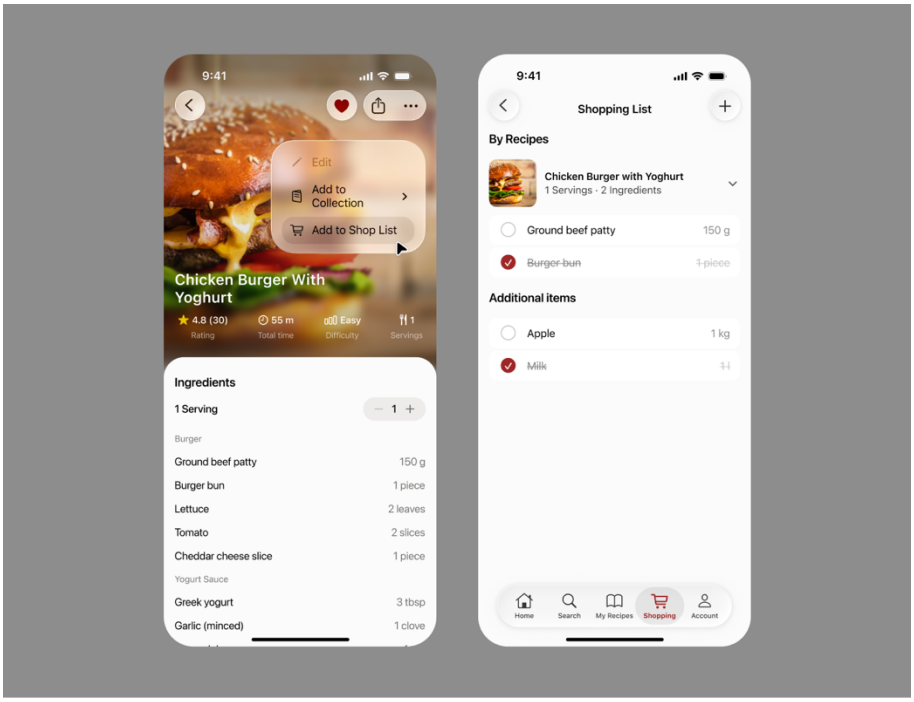


Рисунок 2.10 – Інтерфейс інтегрованого списку покупок

Завершальним етапом проєктування графічного інтерфейсу є створення іконки застосунку. Іконка слугує головним елементом візуальної ідентифікації програмного продукту на екрані пристрою. Процес створення включав розроблення візуальної концепції, підбір контрастної кольорової палітри та тестування читабельності логотипа. Основний візуальний акцент зроблено на фірмовому кольорі застосунку та чіткій типографіці. Для проєктування та фінального експорту графічних активів використано спеціалізоване програмне середовище Icon composer. Цей інструмент призначений спеціально для створення іконок згідно з вимогами операційної системи iOS 26. Програма дозволяє автоматично генерувати всі необхідні розміри графічних елементів та перевіряти їхнє відображення на різних системних фонах [26, 27] (Рисунок 2.11).

Рисунок 2.11 – Іконка застосунку

Отже, розроблений графічний інтерфейс повністю відповідає сучасним вимогам до мобільних застосунків, враховує кращі практики лідерів ринку та забезпечує користувачам зручне середовище для роботи з кулінарним контентом.

2.4 Розроблення алгоритму роботи мобільного застосунку

Алгоритм є точною інструкцією, яка описує порядок дій виконавця для досягнення конкретного результату за визначене число кроків. Згідно з вимогами до програмного забезпечення, під час проєктування визначається повний набір

вихідних даних, кінцевий стан об'єкта та система команд обробки інформації. Для наочного представлення логіки роботи застосовуються графічні моделі. Локальні обчислювальні процеси подаються у вигляді блок-схем. Блок-схема є графічним представленням алгоритму, де кроки зображуються у вигляді геометричних фігур, з'єднаних лініями напрямку для показу послідовності дій [28]. Для опису мережевих асинхронних процесів використовуються моделі уніфікованої мови моделювання. Діаграма послідовності ілюструє взаємодію об'єктів системи у часі і показує порядок обміну повідомленнями між клієнтською та серверною частинами [29]. Нижче наведено описи п'яти алгоритмів роботи застосунку.

Алгоритм авторизації та реєстрації визначає процес ідентифікації профілю в системі через хмарний сервіс автентифікації. Початковий стан: введені облікові дані електронної пошти та пароля. Кінцевий стан: створена активна сесія з правами доступу або виведене повідомлення про помилку (Рисунок 2.12).

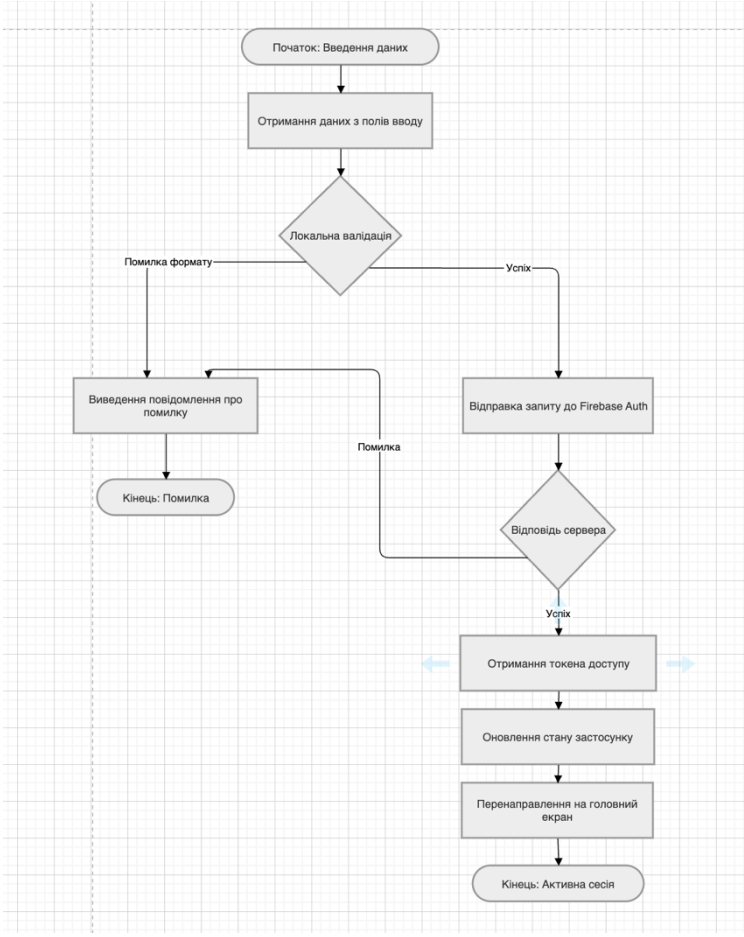


Рисунок 2.12 – Блок-схема алгоритму авторизації користувача

Порядок дій:

– система отримує введені дані з текстових полів;
– виконується локальна перевірка формату пошти та довжини пароля;
– формується і відправляється мережевий запит до сервера;
– перевіряється відповідь сервера;
– у разі помилки алгоритм зупиняється і виводить попередження;
– у разі успіху система отримує маркер доступу;
– оновлюється стан програми і відбувається перехід на головний екран.

Алгоритм створення та збереження рецепта визначає порядок збереження даних у нереляційній базі. Процес вимагає взаємодії мобільного клієнта з кількома хмарними сервісами. Початковий стан: текстові та мультимедійні дані введені у форму створення рецепта. Кінцевий стан: збережений новий запис у базі даних. Порядок дій:

– реєструється подія заповнення форми та прикріплення зображення;
– фотографія асинхронно завантажується у хмарне сховище;
– система отримує мережеве посилання на завантажений файл;
– зібрані дані формуються у структурований об'єкт;
– об'єкт перетворюється у текстовий формат для передачі по мережі;
– відправляється запит на запис документа у базу даних;
– після підтвердження запису інтерфейс показує рецепт (Рисунок 2.13).

Алгоритм пошуку та фільтрації описує механізм знаходження потрібного контенту. Початковий стан: текстовий запит та обрані параметри фільтрації. Кінцевий стан: відфільтрований масив даних виведений на екран. Порядок дій:

– система зчитує введений текст і вибрані параметри фільтрів;
– формується комбінований запит до колекції бази даних;
– отриманий масив результатів проходить локальне сортування;
– відбуваються перевірка на наявність елементів у масиві;
– якщо результати відсутні, на екран виводиться повідомлення;
– якщо результати знайдені, система формує список (Рисунок 2.14).

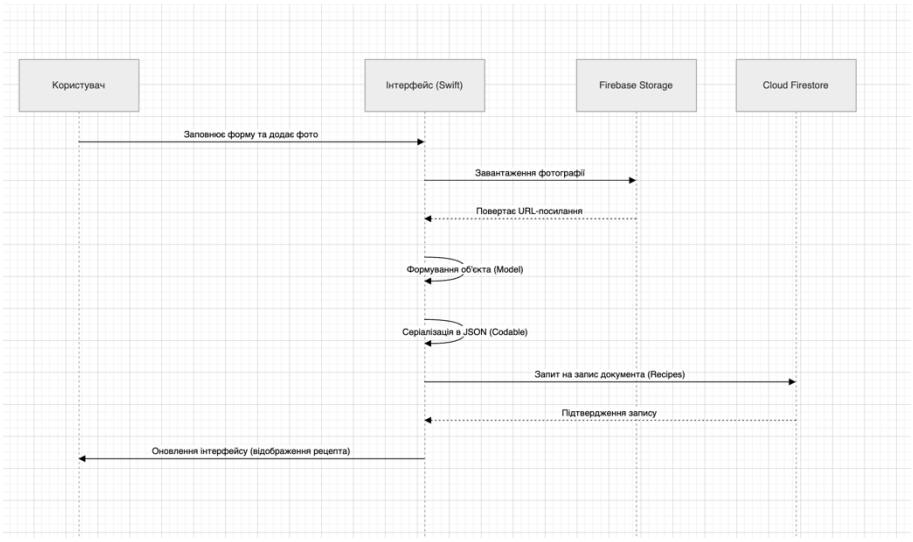


Рисунок 2.13 – UML-модель процесу створення та збереження рецепта

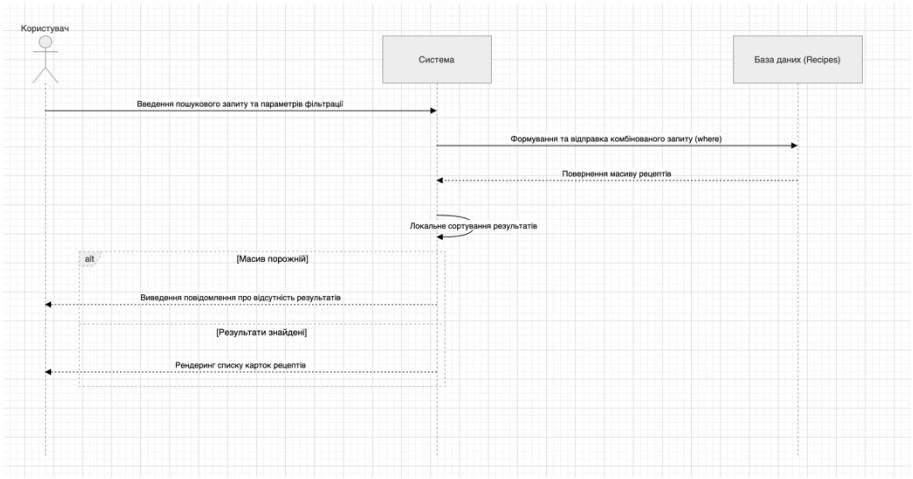


Рисунок 2.14 – UML-модель алгоритму пошуку та фільтрації

Алгоритм генерації рецепта за допомогою штучного інтелекту відображає процес взаємодії мобільного клієнта із зовнішнім сервісом на базі великих мовних моделей. Початковий стан: перелік продуктів введений у форму генерації. Кінцевий стан: згенерований рецепт збережений у базі з відповідним маркером приналежності. Порядок дій:

– програма формує текстовий запит на основі введених інгредієнтів;

– виконується асинхронний мережевий запит до стороннього програмного інтерфейсу;

– інтерфейс переходить у стан очікування і показує завантаження;

– отриманий результат декодується у внутрішню структуру даних;

– створеному об'єкту програмно присвоюється поточна дата та маркер штучного інтелекту;

– об'єкт записується у базу даних;

– інтерфейс оновлюється і відображає готову страву (Рисунок 2.15).

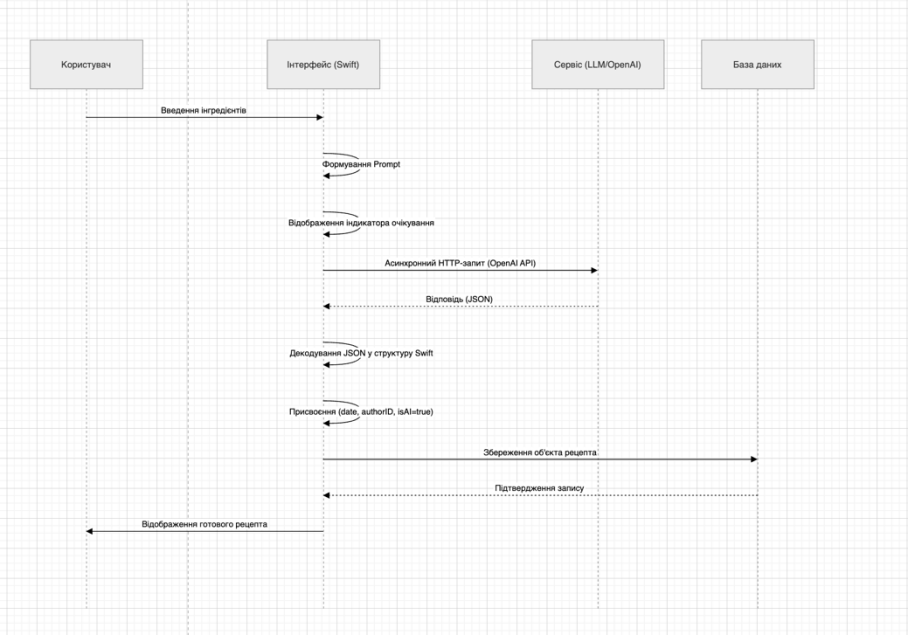


Рисунок 2.15 – UML-модель алгоритму генерації рецепта за допомогою штучного інтелекту

Алгоритм динамічного перерахунку інгредієнтів автоматизує математичний перерахунок пропорцій продуктів для вибраної страви. Процес виконується локально на пристрої. Початковий стан: базова кількість порцій, поточна вага кожного продукту та цільова кількість порцій. Кінцевий стан: оновлений масив продуктів із перерахованою вагою. Порядок дій:

– програма реєструє подію зміни значення порцій; перевіряється умова валідності даних;

– система обчислює коефіцієнт масштабування через ділення цільових порцій на базові;

– запускається цикл перевірки для масиву інгредієнтів;

– базова вага окремого продукту множиться на розрахований коефіцієнт;

– програма перевіряє наявність наступних елементів у масиві;

– після перерахунку останнього елемента цикл завершується;

– оновлені дані передаються для миттєвого оновлення інтерфейсу.

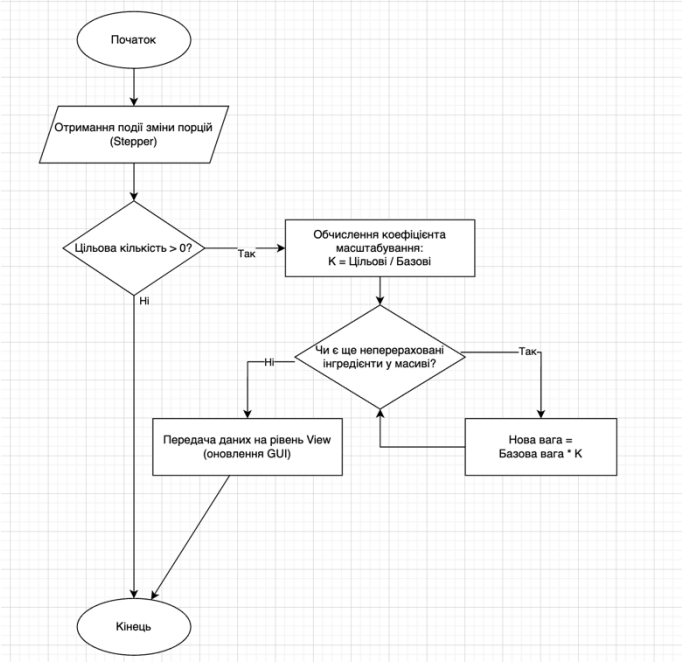


Рисунок 2.16 – Блок-схема алгоритму перерахунку інгредієнтів

2.5 Створення прототипу мобільного застосунку

Після проєктування графічного інтерфейсу здійснюється прототипування головних частин застосунку. Цей процес об'єднує розрізнені макети екранів у єдину логічну систему. Він допомагає перевірити зручність переходів користувача і сформувати остаточну архітектуру навігації. Для побудови схеми роботи та зв'язків застосунку використано середовище розробки Figma. Глобальна навігація системи базується на використанні нижньої панелі вкладок. Цей елемент є стандартом для операційної системи iOS. Він забезпечує швидкий доступ до п'яти паралельних гілок ієрархії. До цих гілок належать головна

сторінка, пошук, розділ власних рецептів, список покупок та профіль користувача. Кожна вкладка працює як незалежний стек навігації.

Архітектура переходів між екранами поділяється на два основні типи:

– ієрархічні переходи;

– модальні переходи.

Ієрархічні переходи використовуються для заглиблення у контент. При натисканні на картку рецепта на головному екрані або в результатах пошуку система виконує перехід на сторінку деталей страви. Користувач завжди має можливість повернутися на крок назад. Для цього передбачена системна кнопка у верхньому лівому куті екрана та стандартний жест гортання від лівого краю пристрою. Аналогічно реалізовано перехід від загального списку колекцій до вмісту конкретної тематичної папки. Модальні переходи застосовуються для виконання тимчасових ізольованих завдань. Вони візуально переривають основний потік роботи.

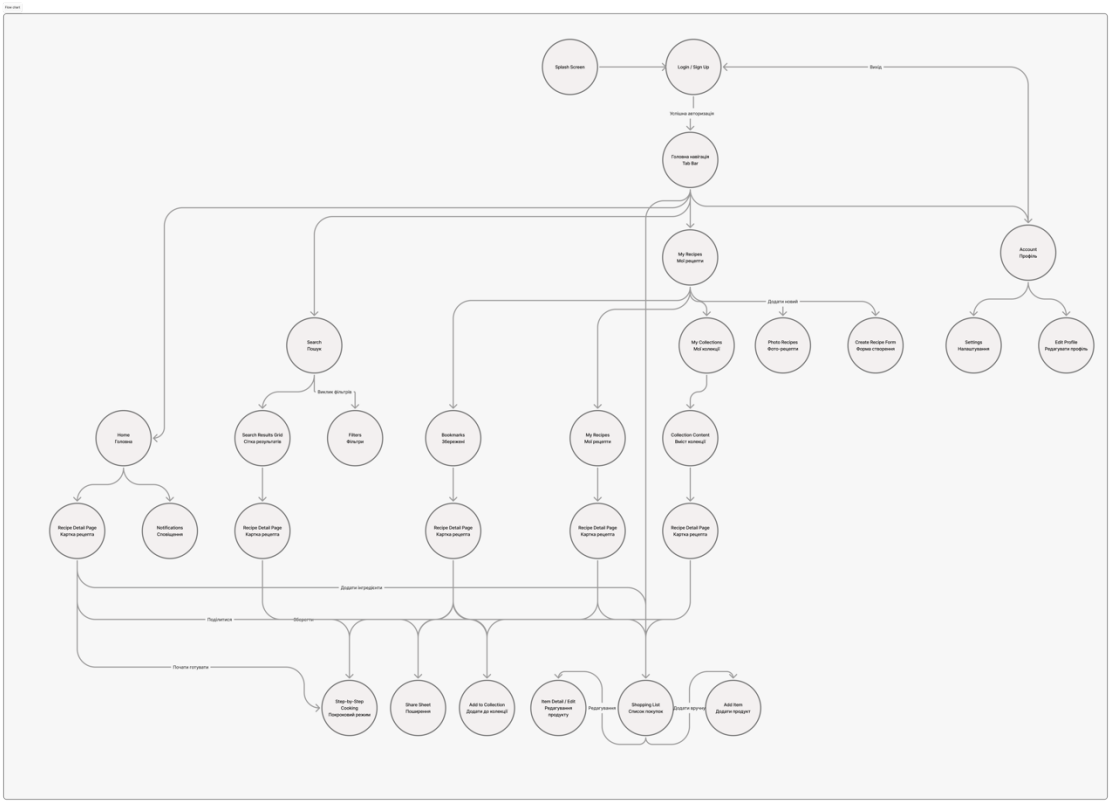


Рисунок 2.17 – Схема переходів та роботи мобільного застосунку

Екран авторизації, форма створення нового рецепта або панель управління для адміністратора з'являються поверх поточного контексту програми. Після збереження рецепта або успішного входу в обліковий запис модальне вікно закривається. Користувач автоматично повертається до попереднього стану системи. На основі готових прототипів будується схема роботи мобільного застосунку. Вона відображає карту екранів та логіку навігаційних переходів між ключовими вузлами програми. Схематичне подання переходів наведено на рисунку 2.16. Створений прототип дав змогу виявити потенційні помилки у навігації до етапу написання програмного коду. Отримана схема задовольняє ергономічні вимоги. Вона покриває всі сценарії використання системи, які були визначені на етапі аналізу вимог.

2.6 Аналіз та вибір технологій і методів реалізації застосунку

Для виконання поставлених у роботі завдань та забезпечення високої якості кінцевого продукту було проведено аналіз сучасних технологій, фреймворків та інструментальних засобів розроблення програмного забезпечення. Вибір технологічного стека здійснювався з урахуванням цільової операційної системи iOS, вимог до безпеки даних та необхідності реалізації клієнт-серверного функціоналу. Оскільки цільовою платформою застосунку є операційна система мобільних пристроїв компанії Apple, основною мовою програмування було обрано Swift. На відміну від застарілої Objective-C, Swift є сучасною, безпечною та високопродуктивною мовою, яка забезпечує автоматичне управління пам'яттю ARC та строгу типізацію. Це дозволяє мінімізувати кількість критичних помилок ще на етапі компіляції коду. Основним інтегрованим середовищем розроблення IDE обрано офіційний інструмент Xcode. Він надає повний набір засобів для написання коду, профілювання продуктивності та тестування застосунку на вбудованих програмних емуляторах різних моделей пристроїв.

Для реалізації графічного інтерфейсу було проаналізовано два нативні підходи: імперативний з фреймворком UIKit та декларативний з фреймворком SwiftUI. Вибір було зроблено на користь SwiftUI. Цей сучасний фреймворк дозволяє описувати інтерфейс декларативно, що значно скорочує обсяг програмного коду та спрощує реалізацію архітектурного патерну MVVM. SwiftUI має вбудовану підтримку динамічного шрифту, автоматично адаптує кольори під темну тему та ідеально відповідає вимогам екосистеми Apple.

Розроблення власного сервера потребує значних витрат часу на налаштування серверної інфраструктури, написання API та забезпечення безпеки. Тому було прийнято рішення використати хмарну платформу Firebase – Backend-as-a-Service . З її інструментарію обрано три ключові сервіси:

– Firebase Authentication – для швидкої та безпечної реалізації механізмів реєстрації та авторизації користувачів без необхідності самостійно хешувати паролі та керувати токенами сесій;

– Cloud Firestore – документо-орієнтована NoSQL база даних. Вона підходить для збереження кулінарних рецептів у форматі JSON, дозволяє працювати з ієрархічними даними та підтримує синхронізацію стану між клієнтом і сервером у режимі реального часу;

– Firebase Storage – хмарне сховище для збереження медіафайлів.

Для реалізації функціоналу генерації рецептів за переліком інгредієнтів було обрано OpenAI API. Використання готової великої мовної моделі через REST API є найбільш доцільним підходом, оскільки розгортання власної нейромережі подібного рівня на мобільному пристрої неможливе через апаратні обмеження. OpenAI API забезпечує високу точність обробки природної мови та здатність повертати результат у чітко структурованому форматі, придатному для автоматичного парсингу в застосунку. Для розроблення візуальної концепції, побудови макетів та прототипування застосунку, на етапах, що передували написанню коду, використано хмарний графічний редактор Figma. Його інструментарій дозволяє створювати інтерактивні прототипи та експортувати графічні елементи безпосередньо у середовище Xcode.

Отже, обраний комплекс технологій, мов та платформ повністю відповідає актуальним вимогам індустрії та забезпечує оптимальні умови для реалізації всіх поставлених задач проєктування.

2.7 Висновки

У другому розділі пояснювальної записки виконано проєктування мобільного застосунку для управління кулінарними рецептами. Для побудови програмної системи обрано клієнт-серверну архітектуру. Вона базується на використанні хмарних сервісів та архітектурному патерні MVVM. Під час роботи спроєктовано логічну структуру документо-орієнтованої бази даних. Створені моделі п'яти основних колекцій забезпечують збереження профілів користувачів, списків покупок і кулінарних рецептів. Розроблено графічний інтерфейс програми. Усі макети створено з дотриманням принципів проєктування операційної системи iOS. Побудований інтерактивний прототип відображає ієрархічні та модальні переходи між екранами. Розроблена карта екранів формує логічну систему навігації застосунку.

Для формалізації роботи застосунку розроблено п'ять алгоритмів. Вони описують процеси авторизації, створення записів та пошуку даних. Окремо описано порядок дій для генерації рецептів за допомогою сторонніх сервісів та алгоритм математичного перерахунку ваги інгредієнтів. Зазначені процеси візуалізовано за допомогою блок-схем і діаграм уніфікованої мови моделювання. Проведений аналіз визначив перелік технологій для подальшого розроблення програмного продукту. Клієнтську частину буде реалізовано мовою Swift за допомогою фреймворку SwiftUI. Серверну частину забезпечать інструменти платформи Firebase. Для генерації текстового контенту обрано програмний інтерфейс OpenAI. Отримані результати проєктування відповідають поставленим завданням та утворюють основу для етапу програмної реалізації застосунку.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Реалізація логіки мобільного застосунку

Програмна реалізація мобільного застосунку передбачає створення надійного механізму управління внутрішніми процесами та станами. На етапі проєктування системи визначено базовий набір дій та функцій. Програма забезпечує реєстрацію нових профілів, авторизацію через хмарні сервіси та перегляд загальної стрічки страв. Реалізовано засоби для розширеного пошуку, фільтрації контенту за дієтичними параметрами та впорядкування списків. Передбачено можливості створення власних рецептів із завантаженням фотографій, ведення списків покупок і групування страв у тематичні колекції. Окремим модулем реалізовано генерацію кулінарних інструкцій за допомогою штучного інтелекту. Рівень логіки приймає вхідні сигнали від клієнта, звертається до сервісів бази даних і готує масиви інформації для відображення на екрані. Завдяки чіткому розділенню обов'язків між програмними модулями, будь-які зміни в алгоритмах взаємодії з хмарою Firestore не вимагають перебудови компонентів нативного фреймворку SwiftUI.

Програмний код написано мовою Swift з використанням архітектурного шаблону MVVM. Цей підхід чітко відокремлює бізнес-логіку від графічної розмітки. Такий поділ спрощує пошук помилок і дозволяє проводити ізольоване тестування окремих компонентів. Бізнес-логіка застосунку розподілена між кількома ключовими файлами. До цього переліку входять файли:

- MainViewViewModel.swift;
- LoginViewViewModel.swift;
- RecipeListViewViewModel.swift;
- RecipeFormViewModel.swift;
- Router.swift;
- CookBookApp.swift.

Управління глобальним станом сесії реалізовано в класі MainViewViewModel. Для автоматичного відстеження авторизації використовується слухач AuthStateDidChangeListener із запобіганням витокам пам'яті через слабке посилання.

```
self.handler = Auth.auth().addStateDidChangeListener { [weak self] _, user in
    DispatchQueue.main.async {
        self?.currentUserId = user?.uid ?? ""
        if user != nil { self?.isGuest = false }
    }
    // подальший запит ролі користувача з бази даних
}
```

Точкою входу в програму є структура CookBookApp, позначена атрибутом @main. Структура відповідає протоколу App і визначає життєвий цикл застосунку. Основний контент розміщується всередині сцени WindowGroup. Для обробки зовнішніх посилань розроблено клас Router, який також підтримує протокол ObservableObject. Екземпляр маршрутизатора створюється один раз під час запуску системи за допомогою обгортки @StateObject. Цей об'єкт передається в глобальне середовище інтерфейсу через модифікатор environmentObject.

Завдяки цьому будь-який дочірній екран може зчитувати стан маршрутизації. Механізм прямих переходів реалізовано через реєстрацію власної схеми посилань cookbook://. Модифікатор onOpenURL перехоплює зовнішні виклики та передає отриману адресу до функції обробки. Клас Router аналізує структуру адреси за допомогою компонента URLComponents. Логіка безпечно витягує значення параметра ідентифікатора рецепта та записує його в опубліковану змінну. Це викликає автоматичний перехід на відповідну сторінку деталей страви.

```
class Router: ObservableObject {
    @Published var deepLinkedRecipeID: String?

    func handle(url: URL) {
        guard url.scheme == "cookbook",
              url.host == "recipe",
              let components = URLComponents(url: url, resolvingAgainstBaseURL:
false),
```

```
            let id = components.queryItems?.first(where: { $0.name == "id"
})?.value
        else { return }
        deepLinkedRecipeID = id
    }
}

// CookBookApp.swift –точка входу
@main
struct CookBookApp: App {
    @StateObject var router = Router()

    init() { FirebaseApp.configure() }

    var body: some Scene {
        WindowGroup {
            MainView()
                .environmentObject(router)
                .onOpenURL { url in router.handle(url: url) }
        }
    }
```

Клас RecipeListViewViewModel готує масиви даних для відображення, послідовно фільтруючи записи за збереженими маркерами та текстовими запитами. Реактивна обробка подій введення в класах моделі представлення реалізована за допомогою фреймворку Combine. Роботу штучного інтелекту забезпечує клас OpenAIService. Запит надсилається з параметром response_format: json_object для отримання чистого коду без зайвої текстової розмітки. Декодування відповіді відбувається двоетапно: після зняття зовнішньої обгортки ChatResponse з поля content вилучається цільова структура AIRecipe. Фінальний результат перетворюється на стандартний об'єкт RecipeListItem із прапорцем isAI: true для візуального виділення у загальному списку.

```
let chat = try JSONDecoder().decode(ChatResponse.self, from: data)
guard let raw = chat.choices.first?.message.content,
      let contentData = raw.data(using: .utf8) else { throw
OpenAIError.emptyResponse }
// Декодування внутрішньої структури AIRecipe
      let ai = try JSONDecoder().decode(AIRecipe.self, from: contentData)
```

Проєктування надійного програмного забезпечення вимагає чіткого розмежування внутрішніх компонентів [30]. Згідно з визначенням шаблону сервісного рівня в каталозі архітектурних патернів підприємницьких застосунків Мартіна Фаулера, виділення окремого шару сервісів дозволяє встановити чітку

межу системи [31]. Сервісний рівень інкапсулює бізнес-логіку та контролює виконання транзакцій під час спільної роботи компонентів. Пряме вбудовування логіки обміну даними в графічні елементи створює значне дублювання коду. Тому винесення взаємодії із зовнішніми ресурсами в ізольовані сервіси спрощує підтримку та подальше масштабування системи. Така декомпозиція є фундаментальним інженерним рішенням. Вона дозволяє проводити незалежну модифікацію мережевих запитів без ризику порушити стабільність візуального шару програми. Проєкт організовано за модульним принципом і поділено на чотири основні групи. Схематичний розподіл цих груп відображено на рисунку 3.1. Така організація відповідає архітектурному патерну MVVM і забезпечує чіткий розподіл відповідальностей між шарами застосунку:

– Models – структури даних з підтримкою протоколу Codable;

– ViewModels – класи бізнес-логіки, що відповідають ObservableObject;

– Views – декларативні екрани на базі SwiftUI, згруповані за розділами;

– Services – сервісний шар, що інкапсулює роботу з хмарною базою даних та зовнішніми інтерфейсами.

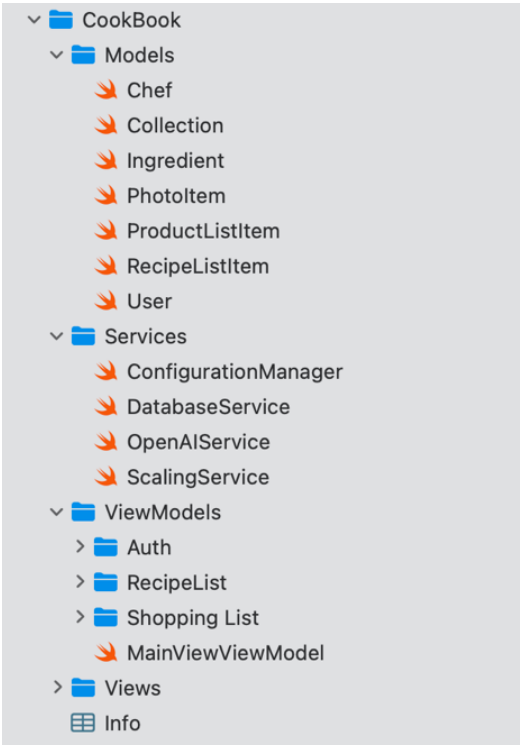


Рисунок 3.1 – Структура проєкту

3.2 Реалізація розмітки мобільного застосунку

Розроблення графічного інтерфейсу мобільного застосунку виконано за допомогою фреймворку SwiftUI. Цей інструмент використовує декларативний підхід до побудови користувацького інтерфейсу. Замість ручного перетягування елементів у візуальному редакторі розробник описує вигляд екрана за допомогою програмного коду мовою Swift. Основою кожного екрана програми є структура, яка підпорядковується системному протоколу View. Така структура обов'язково містить обчислювальну властивість body. Саме всередині цієї властивості формується ієрархія візуальних компонентів. Для зв'язування графічних елементів із даними використовуються спеціальні обгортки властивостей. Вони забезпечують автоматичне оновлення інтерфейсу при зміні внутрішнього стану програми.

Головним контейнером застосунку є структура MainView. Вона виконує функцію базового маршрутизатора екранів. Залежно від статусу профілю система приймає рішення про відображення певного набору вкладок. Якщо користувач успішно пройшов авторизацію, програма генерує повноцінну нижню панель навігації за допомогою компонента TabView. Для гостьового доступу завантажується інший, обмежений інтерфейс. Якщо користувач не авторизований і не обрав гостьовий режим, на екран виводиться форма входу.

Для реалізації режиму обмеженого доступу створено окрему структуру GuestTabView. Гостьовий користувач має право переглядати лише публічну стрічку рецептів. При спробі переходу на вкладку збережених рецептів або власного профілю спрацьовує програмне блокування. Замість відкриття нового екрана програма фіксує зміну стану змінної activeTab і повертає користувача на початкову вкладку. Одночасно з цим інтерфейс викликає системний модифікатор alert. На екран виводиться діалогове вікно з вимогою пройти реєстрацію для продовження роботи.

Головний OverviewView екран реалізовано у вигляді вертикального прокручуваного контейнера ScrollView. У верхній частині розміщено повноекранну карткову область з фоновим зображенням страви, поверх якого виводиться мітка категорії та назва рецепта. Нижче зображення розташовано рядок із кнопкою переходу до деталей рецепта та кнопкою додавання до обраних. Секція «Just for You» реалізована у вигляді горизонтального ScrollView з картками рецептів у форматі LazyHStack. Навігація між розділами застосунку здійснюється через нижню панель TabView з п'ятьма вкладками: Home, Search, My Recipes, Shopping та Account (Рисунок 3.2).

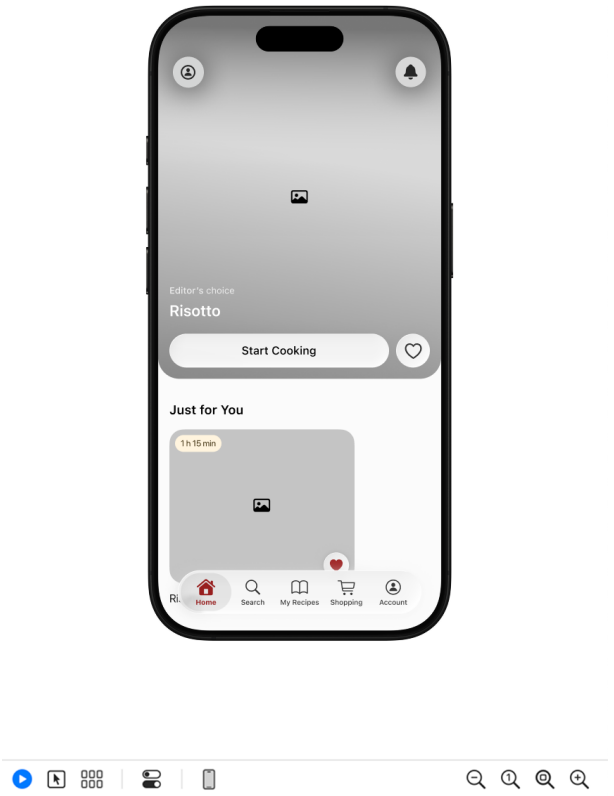


Рисунок 3.2 – Макет головного екрана

Екран «Мої рецепти» RecipeMainView організовано як двоколонкову сітку LazyVGrid з рівномірним розподілом елементів. Кожна клітинка сітки є навігаційним посиланням NavigationLink і відображає тематичну іконку, назву розділу та кількість елементів у вигляді мітки-бейджа. Представлено чотири розділи: «Bookmarks», «My Recipes», «My Collections» та «Photo Recipes». У

верхній частині екрана розміщено пошукове поле TextField з іконкою мікрофона та кнопкою виклику панелі фільтрів (Рисунок 3.3).

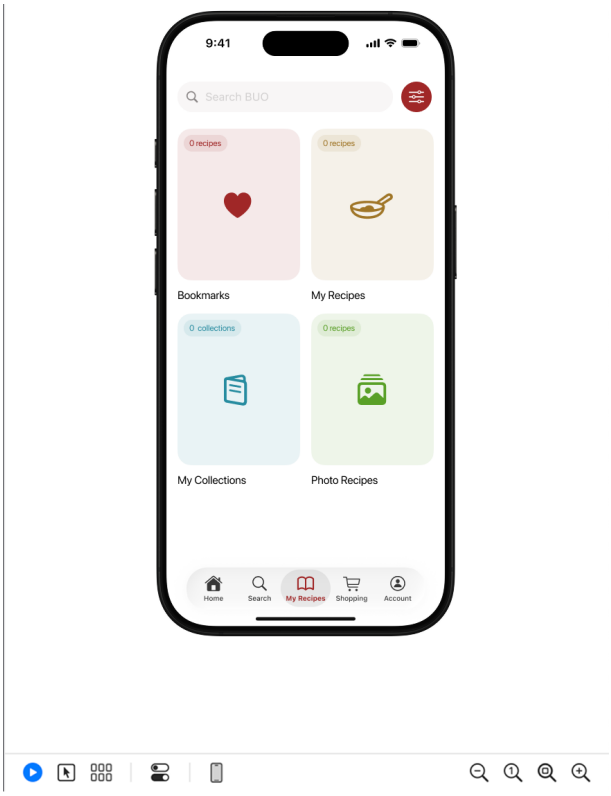


Рисунок 3.3 – Макет екрана «Мої рецепти»

```
var body: some View {
    VStack(spacing: 0) {
        ZStack {
            Image(recipeImageName)
                .resizable()
                .scaledToFill()
                .frame(height: 500)
                .cornerRadius(39)
                .clipped()
                .ignoresSafeArea(edges: .top)

            VStack {
                CustomHeaderView()
                    .padding(.horizontal, 16)
                    .padding(.top, 60)
                Spacer()
                VStack(alignment: .leading, spacing: 10) {
                    Text(recipeCategory)
                        .font(.system(size: 14, weight: .regular))
                        .foregroundColor(.white)
                    Text(recipeTitle)
                        .font(.system(size: 22, weight: .semibold))
                        .foregroundColor(.white)
                    HStack {
                        Button { } label: {
```

```
                            Text("Start Cooking")
                                .frame(width: 312, height: 48)
                                .background(Capsule().fill(Color.white))
                                .foregroundColor(.black)
                        }
                        Spacer()
                        Button { } label: {
                            Image(systemName: "heart.fill")
                                .frame(width: 48, height: 48)
                                .background(Circle().fill(Color.white))
                                .foregroundColor(Color("brown-500"))
                        }
                    }
                }
                .padding(.horizontal, 16).padding(.bottom, 16)
            }
        }
        .frame(height: 500)
    }
    .ignoresSafeArea(edges: .top)
}
```

Екран деталей рецепта RecipeDetailView побудовано на основі ScrollView з фіксованим заголовком. Верхню частину займає повноекранне зображення страви, завантажене асинхронно через AsyncImage. Поверх зображення розміщено кнопки повернення та додавання до обраних. Нижче зображення у білій картці відображається назва рецепта, а під нею горизонтальний рядок з чотирма метриками: рейтинг із зіркою, час приготування, рівень складності та кількість порцій. Секція інгредієнтів містить лічильник порцій, реалізований через кнопки збільшення та зменшення зі збереженням стану у @State-змінній. Список інгредієнтів відображається з автоматичним перерахунком кількості пропорційно до обраної кількості порцій (Рисунок 3.4).

```
struct ProductListView: View {
    @StateObject var viewModel: ProductListViewViewModel
    @FirestoreQuery var items: [ProductListItem]

    init(userId: String) {
        self._items = FirestoreQuery(collectionPath: "users/\(userId)/products")
        self._viewModel = StateObject(wrappedValue:
ProductListViewViewModel(userId: userId))
    }

    private var groupedItems: [(title: String, items: [ProductListItem])] {
        let dict = Dictionary(grouping: items) { $0.recipeTitle ?? "Other" }
        return dict.sorted { $0.key == "Other" ? false : $1.key == "Other" ?
true :
            $0.key.localizedCaseInsensitiveCompare($1.key) == .orderedAscending
}
```

```
            .map { (title: $0.key, items: $0.value) }
    }

    var body: some View {
        NavigationView {
            List {
                ForEach(groupedItems, id: \.title) { group in
                    Section(header: Text(group.title)) {
                        ForEach(group.items) { item in
                            ProductListItemView(item: item)
                                .swipeActions {
                                    Button(role: .destructive) {
                                        viewModel.delete(id: item.id)
                                    } label: { Label("Delete", systemImage:
"trash") }
                                }
                        }
                    }
                }
            }
            .listStyle(.insetGrouped)
            .navigationTitle("Shopping List")
        }
    }
}
```

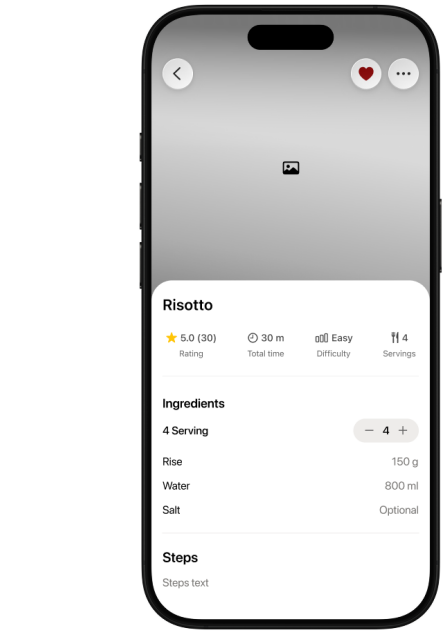


Рисунок 3.4 – Макет екрана «Детальна сторінка рецепту»

Список покупок ProductListView реалізовано на основі компонента List зі свайп-жестами для видалення елементів через модифікатор .swipeActions. Кожен рядок списку є окремим екземпляром ProductListItemView і містить назву

продукту, кількість та чекбокс для позначення виконання. Додавання нового елемента здійснюється через модальний аркуш sheet, що викликається кнопкою з плаваючим стилем FloatingAddButton. Стан списку синхронізується з Firestore у режимі реального часу через ListenerRegistration (Рисунок 3.5).

```
private var groupedItems: [(title: String, items: [ProductListItem])] {
    let dict = Dictionary(grouping: items) { $0.recipeTitle ?? "Other" }
    return dict.sorted { $0.key.localizedCaseInsensitiveCompare($1.key) ==
.orderedAscending }
                .map { (title: $0.key, items: $0.value) }
}
```

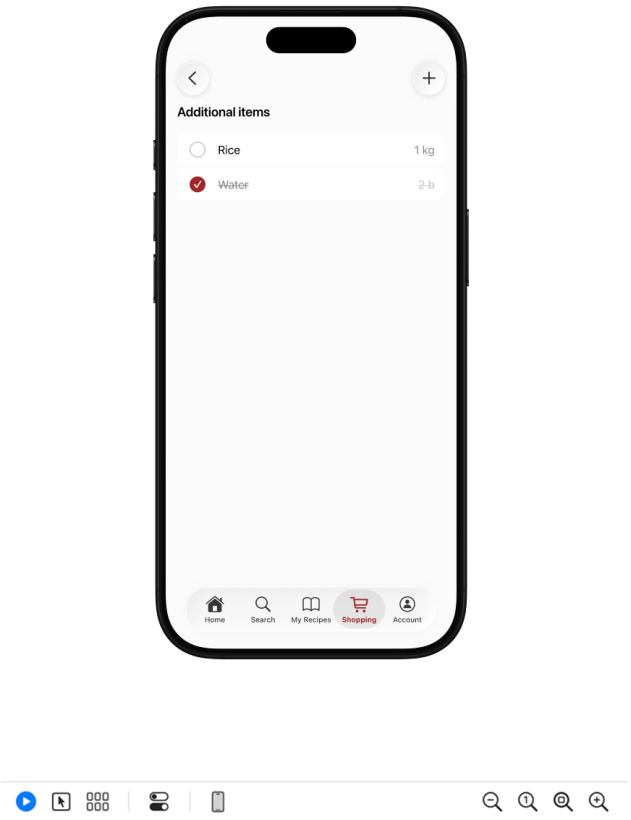


Рисунок 3.5 – Макет екрана «Список покупок»

3.3 Розроблення бази даних

У мобільному застосунку як основне середовище збереження даних застосовується хмарна документно-орієнтована нереляційна база даних Cloud Firestore. Ця система належить до платформи розробки Firebase від компанії

Google. На відміну від класичних реляційних таблиць, сховище організовує інформацію у вигляді гнучкої ієрархії колекцій і окремих документів. Кожен документ зберігає пари ключів та значень у структурі, сумісній із форматами JSON або BSON. Для взаємодії з базою даних у проєкті використовується комплект засобів розробки FirebaseFirestore. Зазначений пакет підключається безпосередньо через вбудований менеджер залежностей Swift Package Manager.

Архітектура сховища побудована на основі кореневих колекцій та вкладених підколекцій. Головна гілка містить профілі зареєстрованих осіб. Кожен такий запис ідентифікується унікальним текстовим ключем і містить базові відомості про клієнта. Усередині документа профілю розміщуються окремі підколекції для збереження створених кулінарних рецептів, списків необхідних продуктів, персональних збірок та завантажених фотографій. Окрім індивідуальних даних, на кореневому рівні існують глобальні колекції. Вони відповідають за збереження загальнодоступних публічних збірок, відстеження оцінок страв і зберігання інформації про професійних шеф-кухарів (Рисунок 3.6).

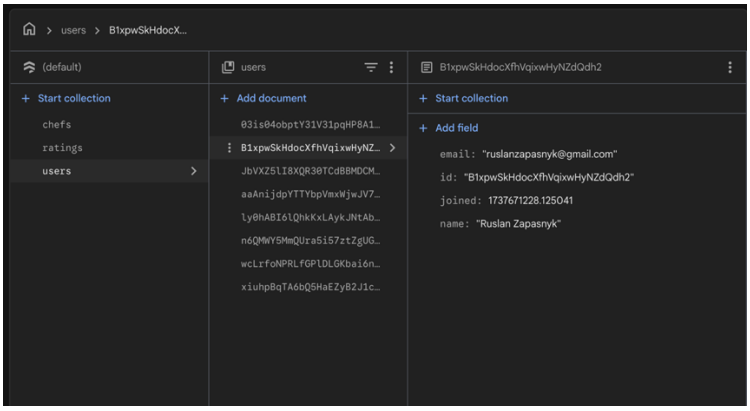


Рисунок 3.6 – Структура колекції документів у Cloud Firestore

Підключення до хмарного середовища ініціалізується під час запуску програми. Структура головного файлу зчитує конфігураційний файл системи та встановлює безпечне з'єднання з серверами. Для централізованого управління доступом до даних створено окремий сервісний клас. Цей компонент реалізовано за допомогою шаблону проєктування Singleton, що гарантує існування лише

одного активного екземпляра підключення в пам'яті пристрою. Сервіс зберігає постійні посилання на базу даних та хмарне сховище медіафайлів.

```
// CookBookApp.swift
import FirebaseCore
import SwiftUI

@main
struct CookBookApp: App {
    init() {
        FirebaseApp.configure() // зчитує GoogleService-Info.plist
    }
    var body: some Scene {
        WindowGroup { MainView() }
    }
}

// DatabaseService.swift – Singleton-сервіс
import FirebaseFirestore
import FirebaseStorage
import FirebaseAuth

class DatabaseService {
    static let shared = DatabaseService()
    private let db = Firestore.firestore()        // екземпляр БД
    private let storage = Storage.storage()       // Firebase Storage
    private init() {}
}
```

Обмін даними між хмарною базою та мобільним клієнтом потребує перетворення документів у внутрішні об'єкти мови Swift. Цей процес забезпечується протоколом Codable [32]. Моделі даних описують усі необхідні поля та це дозволяє системі коректно відображати зміст. Під час розробки враховано необхідність збереження підтримки старих форматів документів. Реалізовано власний алгоритм декодування профілю. Якщо збережений запис не містить поля визначення ролі, система не перериває роботу через помилку. Замість цього алгоритм автоматично присвоює об'єкту стандартний рівень доступу. Статус перевірки рецепта також представлено окремим переліком для зручного збереження текстових значень.

```
// User.swift
enum UserRole: String, Codable {
    case regular
    case admin
}
```

```
struct User: Codable, Identifiable {
    let id: String
    let name: String
    let email: String
    let joined: TimeInterval
    var role: UserRole

    // Кастомний декодер – backward compatibility:
    // старі документи Firestore без поля role не кидають помилку
    init(from decoder: Decoder) throws {
        let c = try decoder.container(keyedBy: CodingKeys.self)
        id     = try c.decode(String.self,       forKey: .id)
        name   = try c.decode(String.self,       forKey: .name)
        email  = try c.decode(String.self,       forKey: .email)
        joined = try c.decode(TimeInterval.self, forKey: .joined)
        role   = (try? c.decodeIfPresent(UserRole.self, forKey: .role)) ??
.regular
    }
}

// RecipeListItem.swift – статус модерації як Codable enum
enum RecipeStatus: String, Codable, CaseIterable {
    case draft    = "draft"
    case pending  = "pending"
    case approved = "approved"
    case rejected = "rejected"
}
```

Збереження нових записів відбувається асинхронно для запобігання блокуванню графічного інтерфейсу. Логіка запису рецепта передбачає попередню обробку вибраного зображення. Файл стискається у форматі JPEG і передається до хмарного сховища медіафайлів. Після успішного завантаження програма отримує пряме мережеве посилання на збережену фотографію. Це посилання додається до загальної структури об'єкта. Далі сформований словник параметрів записується за відповідним шляхом усередині колекції певного облікового запису. Окремий метод забезпечує швидке збереження або оновлення базових даних профілю.

```
// DatabaseService.swift
func saveRecipe(_ recipe: RecipeListItem, selectedImage: UIImage?,
                userId: String, completion: @escaping () -> Void) {
    func write(imageURL: String?) {
        var toSave = recipe
        toSave.imageURL = imageURL
        // Шлях: users/{userId}/recipes/{recipeId}
        db.collection("users").document(userId)
            .collection("recipes").document(recipe.id)
            .setData(toSave.asDictionary()) { error in
                if error == nil { completion() }
            }
```

```
    }


    // Завантаження зображення у Firebase Storage перед записом документа
    guard let image = selectedImage,
          let data = image.jpegData(compressionQuality: 0.8) else {
        write(imageURL: recipe.imageURL); return
    }

    let ref = storage.reference()
              .child("users/\(userId)/recipes/\(recipe.id).jpg")
    ref.putData(data) { _, error in
        if error == nil {
            ref.downloadURL { url, _ in write(imageURL: url?.absoluteString) }
        } else {
            write(imageURL: recipe.imageURL)
        }
    }
}

func saveUser(_ user: User) {
    db.collection("users").document(user.id).setData(user.asDictionary())
}
```

Для забезпечення миттєвого оновлення інформації на екрані реалізовано механізм прослуховування змін у реальному часі. Програма підключає спеціальний обробник до колекції страв клієнта. База даних автоматично надсилає новий знімок документів під час будь-якого додавання чи редагування записів на сервері. Отриманий масив проходить фільтрацію для вилучення пошкоджених елементів і передається до контролера стану. Об'єкт реєстрації слухача зберігається у приватній змінній. Це дозволяє безпечно розірвати мережеве з'єднання під час деініціалізації класу та звільнити оперативну пам'ять.

```
// DatabaseService.swift
func listenToRecipes(userId: String,
                     onChange: @escaping ([RecipeListItem]) -> Void) ->
ListenerRegistration {
    db.collection("users").document(userId).collection("recipes")
        .addSnapshotListener { snapshot, error in
            guard let documents = snapshot?.documents else { return }
            // compactMap – ігнорує документи з помилкою декодування
            onChange(documents.compactMap { try? $0.data(as:
RecipeListItem.self) })
        }
}

// RecipeListViewViewModel.swift – підписка та відписка
private var firestoreListener: ListenerRegistration?

private func fetchRecipes() {
```

```
    firestoreListener = DatabaseService.shared.listenToRecipes(userId: userId) {
[weak self] recipes in
        self?.recipes = recipes    }
}

deinit {
    firestoreListener?.remove() // обов'язкове знімання підписки
}
```

Відображення загальної стрічки страв вимагає одночасного пошуку по всіх вкладених колекціях рецептів незалежно від їхнього батьківського профілю. Цю задачу вирішують групові запити. Програма виконує вибірку документів за заданим ідентифікатором колекції та застосовує критерії фільтрації. Для формування публічної стрічки відбираються лише ті страви, які пройшли перевірку та отримали відповідний статус. Адміністративний інтерфейс використовує аналогічний підхід для завантаження списку записів, що очікують рішення модератора. Окремий метод дозволяє отримувати рецепти конкретного шеф-кухара за його ідентифікатором. Виконання таких складних вибірок за окремими полями потребує попереднього налаштування складених індексів у панелі керування базою даних.

```
// DatabaseService.swift
// Публічна стрічка – approved рецепти з усіх підколекцій "recipes"
func fetchPublicFeed(completion: @escaping ([RecipeListItem]) -> Void) {
    db.collectionGroup("recipes")
        .whereField("status", isEqualTo: RecipeStatus.approved.rawValue)
        .getDocuments { snapshot, error in
            guard let documents = snapshot?.documents, error == nil else {
return }
            let recipes: [RecipeListItem] = documents.compactMap { doc in
                guard let jsonData = try? JSONSerialization.data(withJSONObject:
doc.data())
                else { return nil }
                return try? JSONDecoder().decode(RecipeListItem.self, from:
jsonData)
            }            DispatchQueue.main.async { completion(recipes) }
        }
}


// Черга модерації – pending рецепти по всіх користувачах (Admin)
func fetchPendingRecipes(completion: @escaping (Result<[RecipeListItem], Error>)
-> Void) {
    db.collectionGroup("recipes")
        .whereField("status", isEqualTo: RecipeStatus.pending.rawValue)
        .getDocuments { snapshot, error in
            if let error = error {
                DispatchQueue.main.async { completion(.failure(error)) }
```

```
                return
            }
            let recipes: [RecipeListItem] = (snapshot?.documents ??
[]).compactMap { doc in
                guard let jsonData = try? JSONSerialization.data(withJSONObject:
doc.data())
                else { return nil }
                return try? JSONDecoder().decode(RecipeListItem.self, from:
jsonData)
            }            DispatchQueue.main.async {
completion(.success(recipes)) }
        }
}


// Рецепти конкретного шефа – collectionGroup + whereField по chefId
func fetchRecipesByChef(_ chefId: String) async throws -> [RecipeListItem] {
    let snapshot = try await db.collectionGroup("recipes")
        .whereField("chefId", isEqualTo: chefId)
        .getDocuments()
    return snapshot.documents.compactMap { doc in
        try? decode(RecipeListItem.self, from: doc.data(), docId:
doc.documentID)
    }
}
```

Оцінювання страв кількома клієнтами одночасно може призвести до конфліктів перезапису інформації. Для запобігання втраті даних застосовується механізм транзакцій [33]. Програма зчитує поточний стан документа оцінок і виконує розрахунки нових середніх показників усередині ізольованого блоку. Якщо інший процес встигає змінити ці дані під час виконання операції, база даних автоматично повторює транзакцію з новими вхідними параметрами. Це гарантує цілісність інформації. Розрахований рейтинг одночасно записується в окремий документ статистики та оновлюється у загальному записі самої страви.

```
func submitRating(recipeId: String, ownerId: String?, stars: Int) async throws {
    guard let userId = Auth.auth().currentUser?.uid else { throw NSError(...) }
    let clampedStars = max(1, min(5, stars))
    let ratingRef  = db.collection("ratings").document(recipeId)
    let recipeRef: DocumentReference? = ownerId.map { oid in

db.collection("users").document(oid).collection("recipes").document(recipeId)
    }

    _ = try await db.runTransaction { transaction, errorPointer in
        let snapshot: DocumentSnapshot
        do { snapshot = try transaction.getDocument(ratingRef) }
        catch let e as NSError { errorPointer?.pointee = e; return nil }

        var userRatings = snapshot.data()?["userRatings"] as? [String: Int] ??
[:]
```

```
        userRatings[userId] = clampedStars
        let total = userRatings.values.reduce(0, +)
        let count = userRatings.count
        let avg   = count > 0 ? Double(total) / Double(count) : 0.0

        // Атомарний запис у ratings/{recipeId
        // Синхронізація середнього у документ рецепта

    }
}
```

Масове додавання елементів вимагає оптимізації кількості мережевих звернень. Перенесення списку інгредієнтів до переліку покупок реалізовано через механізм пакетного запису. Програма створює локальний пакет команд і послідовно додає до нього вказівки на створення нових документів продуктів. Усі сформовані операції відправляються на сервер єдиним запитом. Пакетний запис виконується за принципом повної атомарності. Зміни зберігаються лише у випадку успішного виконання всіх команд, що виключає появу частково збережених списків.

```
func addProducts(_ items: [ProductListItem], userId: String) {
    let batch = db.batch()
    // ... цикл формування записів для колекції products ...
    batch.commit { error in
        if let error = error { print("Batch error: \(error.localizedDescription)")
}
    }
}
```

Редагування масивів усередині існуючих документів реалізовано за допомогою спеціальних команд на рівні полів бази даних. Такий підхід дозволяє додавати або видаляти ідентифікатори рецептів у колекціях без необхідності попереднього зчитування всього масиву на клієнтський пристрій. Використання атомарних операцій об'єднання та вилучення елементів запобігає появі дублікатів і забезпечує коректне оновлення переліку збережених збірок.

```
// Збереження публічної колекції у вибраних користувача
func addFavouriteCollection(collectionId: String, userId: String) async throws {
    try await db.collection("users").document(userId)
        .setData(["favouriteCollectionIds":
FieldValue.arrayUnion([collectionId])],
                 merge: true)
}
```

Отримання списку продуктів для відображення на екрані побудовано на основі декларативної реактивної прив'язки. Інтерфейс використовує спеціальну обгортку властивості для автоматичної підписки на вказаний шлях колекції. Компонент самостійно керує життєвим циклом з'єднання та ініціює оновлення розмітки при зміні статусу окремих покупок на сервері.

```
// ProductListView.swift
struct ProductListView: View {
    @FirestoreQuery var items: [ProductListItem]  // автоматична підписка на
колекцію

    init(userId: String) {
        self._items = FirestoreQuery(
            collectionPath: "users/\(userId)/products"        )
    }
    // items оновлюється автоматично при будь-якій зміні у Firestore
}
```

Для спрощення обробки даних розроблено універсальний допоміжний метод декодування. Оскільки серверні запити повертають інформацію у вигляді нетипізованих словників, система виконує проміжне перетворення. Словник серіалізується у бінарний формат і передається стандартному декодеру мови програмування для створення типізованої структури. У разі виникнення невідповідності типів метод генерує зрозумілу помилку з вказівкою на ідентифікатор проблемного документа.

```
    let jsonData = try JSONSerialization.data(withJSONObject: data)    do {
return try JSONDecoder().decode(type, from: jsonData)
    } catch {
        throw DecodingError.dataCorrupted(
            .init(codingPath: [],debugDescription: "doc '\(docId)': \(error)")
```

3.4 Керівництво користувача

Керівництво користувача містить опис функціональних можливостей та правил взаємодії з інтерфейсом мобільного застосунку. Програмний продукт розрахований на роботу в межах трьох основних рольових моделей, кожна з яких

має визначений рівень доступу до даних. Робочий процес побудовано навколо центральної панелі навігації, яка забезпечує перехід між ключовими розділами системи. Доступ до функцій застосунку розподіляється за такими рівнями:

– гість (неавторизований користувач);

– авторизований користувач;

– адміністратор системи.

Для неавторизованих користувачів передбачено перегляд загальної стрічки публічних рецептів та використання інструментів пошуку. Гість може відкривати сторінки деталей страв, переглядати інгредієнти та покрокові інструкції. Після успішної авторизації користувач отримує можливість керувати власним контентом. Основні сценарії роботи авторизованого профілю включають:

– створення нових рецептів через форму з додаванням медіафайлів;

– використання модуля штучного інтелекту для генерації рецептів на основі списку наявних інгредієнтів;

– формування персональних тематичних колекцій для групування страв;

– додавання продуктів до інтегрованого списку покупок безпосередньо з картки рецепта;

– маркування страв як обраних для швидкого доступу.

Процес створення рецепта передбачає заповнення текстових полів, вибір категорії та рівня складності. Система автоматично зберігає чернетки в базі даних. Роль адміністратора передбачає наявність спеціалізованого інтерфейсу для управління контентом бази даних. Адміністратор має доступ до розділу модерації, де відображаються всі нові публічні записи. Також адміністратор може редагувати інформацію про професійних шеф-кухарів та керувати глобальними підбірками рецептів. Взаємодія з інтерфейсом супроводжується візуальним зворотним зв'язком. Під час виконання мережевих запитів або завантаження фотографій на екрані відображаються індикатори активності. У разі виникнення помилок авторизації або збоїв зв'язку з сервером система виводить відповідні текстові сповіщення.

3.5 Технічні характеристики мобільного застосунку

Для гарантованого успішного виконання розробленого мобільного застосунку пристрій користувача повинен відповідати чітко визначеним апаратним та програмним вимогам. Програмний продукт розроблено виключно для екосистеми мобільних пристроїв компанії Apple. Цільовою платформою програми є операційна система iOS.

Для гарантування плавної роботи складних анімацій та швидкого рендерингу багаторівневих списків інтерфейсу, пристрій повинен базуватися на мікроархітектурі процесора не нижче Apple A12 Bionic. Цей процесор містить вбудовану систему Neural Engine, що позитивно впливає на загальну швидкодію при обробці масивів даних. Мінімальний рекомендований обсяг оперативної пам'яті становить 3 ГБ. Це обмеження зумовлене необхідністю утримувати в активній пам'яті зображення страв та забезпечувати роботу реактивних підписок контролерів стану під час взаємодії користувача із застосунком.

Мінімальна підтримувана версія операційної системи iOS 17.0. Ця жорстка вимога зумовлена специфікою написання програмного коду та орієнтацією на найсучасніші стандарти розробки. У застосунку використано нові системні модифікатори життєвого циклу, сучасні макроси спостереження за станом (наприклад, @Observable), а також вдосконалений навігаційний компонент NavigationStack, який дозволяє будувати передбачувану та стабільну ієрархію переходів між екранами. Ці технологічні рішення не підтримуються попередніми версіями системи (iOS 15 чи 16). Компроміс на користь зниження версії ОС призвів би до необхідності написання великого обсягу додаткового коду та загального зниження відмовостійкості програми.

Для встановлення та коректної початкової роботи застосунку пристрій повинен мати щонайменше 150 мегабайтів вільної пам'яті. Цей обсяг є базовим: він необхідний безпосередньо для збереження локальних виконуваних файлів програми, графічних ресурсів інтерфейсу та системних бібліотек. Проте під час

активної експлуатації система інтенсивно використовує постійну пам'ять пристрою для тимчасового кешування завантажених фотографій страв. З часом обсяг кешу може динамічно зростати, тому для оптимального функціонування рекомендується мати до 500 мегабайтів вільного простору. У разі критичної нестачі пам'яті на пристрої, операційна система iOS самостійно очищає директорію тимчасових файлів застосунку без ризику втрати кулінарних записів.

Повноцінна робота застосунку вимагає обов'язкового та стабільного підключення до мережі Інтернет. Програма не підтримує повноцінний автономний режим роботи. Постійне мережеве з'єднання потрібне для двосторонньої синхронізації даних із хмарною базою Cloud Firestore у режимі реального часу. Хоча ядро Firestore містить вбудовані механізми локальної персистенції, будь-які дії щодо модифікації даних потребують підтвердження. Створення нових страв, публікація записів, генерація глибоких посилань або оновлення статусу продуктів у списку покупок ініціюють транзакції до сервера. Інтернет також критично необхідний для завантаження медіафайлів із Firebase Storage та обміну інформацією із зовнішніми сервісами, зокрема для звернень до OpenAI API при генерації страв. Важливо зазначити, що програма коректно реагує на раптову втрату з'єднання: замість аварійного завершення, система зберігає введені користувачем дані у локальний буфер та сповіщає про відсутність зв'язку через інтерфейсні індикатори. Управління потоками даних та реактивне оновлення інтерфейсу реалізовано засобами системного фреймворку Combine у поєднанні з сучасними механізмами паралелізму

Для розроблення застосунку використано сучасну об'єктно-орієнтовану мову програмування Swift версії 5.9. Компіляція та збирання проєкту виконувалися в офіційному інтегрованому середовищі Xcode версії 15.0. Графічна розмітка повністю побудована за допомогою декларативного фреймворку SwiftUI

Окремою технічною вимогою до застосунку є забезпечення високого рівня швидкодії інтерфейсу. Завдяки суворому дотриманню архітектурного шаблону MVVM, усі ресурсомісткі обчислення ізольовані у фонових потоках.

3.6 Тестування мобільного застосунку

3.6.1 Аналіз методів тестування мобільного застосунку

Для забезпечення належного рівня надійності системи обрано комбінований підхід до тестування. Верифікація базується на поєднанні методів білої та чорної скриньки. Перевірка внутрішньої логіки на рівні окремих компонентів виконується за методом білої скриньки за допомогою автоматизованих модульних тестів. Такий підхід дозволяє перевірити точність математичних обчислень, алгоритми фільтрації та правила обробки структур даних без запуску графічного інтерфейсу.

Функціональне тестування готового продукту виконується за методом чорної скриньки. Система розглядається як цілісний об'єкт, внутрішня будова якого залишається невідомою під час перевірки. Тестування спрямоване на підтвердження коректної реакції програми на вхідні сигнали..

3.6.2 Тестування мобільного застосунку за допомогою емулятора

Перевірка роботи мобільного застосунку виконується в інтегрованому середовищі розроблення Xcode за допомогою вбудованих програмних симуляторів. Використання емулятора дозволяє швидко та безпечно відтворити поведінку системи на різних поколіннях пристроїв без необхідності залучення фізичного обладнання. Для автоматизації процесу перевірки розроблено набір тестів на базі нативного фреймворку XCTest. Логіку роботи розділено на два окремі модулі: перевірку бізнес-логіки та перевірку стабільності запуску графічного інтерфейсу.

Перший набір автоматизованих перевірок стосується сервісу масштабування інгредієнтів ScalingService. Модульні тести підтверджують правильність перерахунку цілих, десяткових та дробових числових значень пропорцій залежно від зміни кількості порцій. Також перевіряється стійкість

алгоритму до некоректних вхідних параметрів. Лістинг реалізованих тестів наведено нижче.

```
    func test_scale_decimalQuantity() {
        let ingredients = [ProductListItem(id: "3", title: "Oil", quantity: "2.5
tbsp", isDone: false)]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "5 tbsp")
    }

    func test_scale_fractionQuantity_toWhole() {
        let ingredients = [ProductListItem(id: "4", title: "Milk", quantity:
"1/2 cup", isDone: false)]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "1 cup")
    }
```

Другий блок тестів спрямований на перевірку коректності декодування інформації з формату обміну даними у внутрішні структури мови програмування. Тести підтверджують безпечне зчитування документів профілів, статусів модерації та списків покупок. Реалізовано перевірку збереження зворотної сумісності для застарілих записів у базі даних. Вона повністю виключає виникнення критичних помилок інтерфейсу при парсингу опціональних полів.

```
    func test_user_missingRole_defaultsToRegular() throws {
        let json = """
        {
            "id": "u2",
            "name": "Guest",
            "email": "g@example.com",
            "joined": 1700000000.0
        }
        """.data(using: .utf8)!
        let user = try JSONDecoder().decode(User.self, from: json)
        XCTAssertEqual(user.role, .regular)
    }
```

Третій набір модульних перевірок ізольовано тестує механізми пошуку, впорядкування та застосування фільтрів до масиву страв. Перевіряється незалежність регістру символів під час введення пошукових запитів, робота дієтичних позначок відсутності м'яса чи глютену, а також поєднання кількох критеріїв одночасно.

Окрім перевірки логічних класів, створено модуль автоматизованого тестування графічного інтерфейсу CookBookUITests. Завданням цього модуля є підтвердження успішного запуску програми без виникнення критичних збоїв та вимірювання часових затримок під час ініціалізації стартового екрана.

```
@MainActor
func test_launch_doesNotCrash() {
    XCTAssertTrue(app.state == .runningForeground)
}

@MainActor
func test_launch_performance() throws {
    measure(metrics: [XCTApplicationLaunchMetric()]) {
        XCUIApplication().launch()
    }
```

Усі наведені автоматизовані перевірки успішно пройдено в середовищі розроблення. Звіт системи підтверджує відсутність помилок компіляції та виконання. Додатково проведено повний цикл ручного тестування програми в середовищі симулятора. Перевірка здійснювалася шляхом послідовного виконання запланованих дій з фіксацією отриманої реакції інтерфейсу. Сценарії охоплюють введення коректних даних та перевірку захисних механізмів під час спроб виконання недопустимих операцій. Результати виконання тестових сценаріїв зведено в таблиці 3.1, 3.2 та 3.3. Комплексний підхід до верифікації дозволив змоделювати поведінку кінцевого користувача (Рисунок 3.7).

Таблиця 3.1 – Сценарії перевірки процесів ідентифікації та доступу

Ідентифікатор	Екран/ Контекст	Вхідні дані / Дія	Реакція програми	Результат
1	2	3	4	5
TC-AUTH-01	Екран входу	Введення валідної пошти та коректного пароля, натискання кнопки підтвердження	Система надсилає запит, закриває форму та відкриває головну панель навігації	Успішно

Продовження таблиці 3.1

1	2	3	4	5
TC-AUTH-03	Екран входу	Введення пошти без символу «@» та пароля, натискання кнопки підтвердження	Програма залишається на екрані входу, виводиться повідомлення про неправильний формат пошти	Успішно
TC-AUTH-04	Екран входу	Натискання кнопки переходу в гостьовий режим перегляду	Форма авторизації зникає, завантажується стрічка страв із прихованими кнопками додавання	Успішно
TC-AUTH-05	Профіль	Натискання кнопки виходу з облікового запису в налаштуваннях	Сесія завершується, очищаються локальні дані, відкривається стартовий екран ідентифікації	Успішно

Рисунок 3.7 – Результати перевірки процесів ідентифікації та доступу

Таблиця 3.2 – Сценарії перевірки управління кулінарним контентом

Ідентифікатор	Екран / Контекст	Вхідні дані / Дія	Реакція програми	Результат
1	2	3	4	5
TC-REC-01	Створення страви	Заповнення назви, вибір категорії, додавання кроку приготування, збереження	Запис успішно додається до приватної колекції, форма закривається, оновлюється загальний список	Успішно

Продовження таблиці 3.2

1	2	3	4	5
TC-REC-02	Створення страви	Спроба збереження форми без введення назви рецепта	Кнопка збереження ігнорує запит або система підсвічує пусте поле червоним контуром	Успішно
TC-REC-03	Картка страви	Натискання на іконку збереження в обране у верхній панелі	Іконка змінює заливку, ідентифікатор страви записується до масиву закладок профілю	Успішно
TC-REC-04	Стрічка страв	Введення існуючого слова в текстовий рядок пошуку	Список миттєво скорочується, залишаються лише картки з відповідним текстом у назві	Успішно
TC-REC-05	Стрічка страв	Активація перемикача відображення виключно веганських позицій	Інтерфейс перемальовується, зникають страви, що містять продукти тваринного походження	Успішно

Представлені у таблиці 3.2 сценарії дозволили комплексно оцінити стабільність модулів, що відповідають за маніпулювання даними та відображення динамічних елементів. Усі заплановані перевірки кулінарного контенту були виконані без збоїв у графічній оболонці. Валідація пошукових та фільтраційних механізмів підтвердила високу швидкість обробки локальних масивів. При зміні стану текстового поля або активації специфічних перемикачі інтерфейс реагує без затримок.. Блокування запису неповних даних запобігає засміченню хмарного сховища порожніми або некоректними документами NoSQL. Цілісність бази даних залишається непорушною. Візуальна індикація незаповнених полів введення зникає автоматично відразу після початку введення символів користувачем. Результати також представлено на рисунку 3.8.

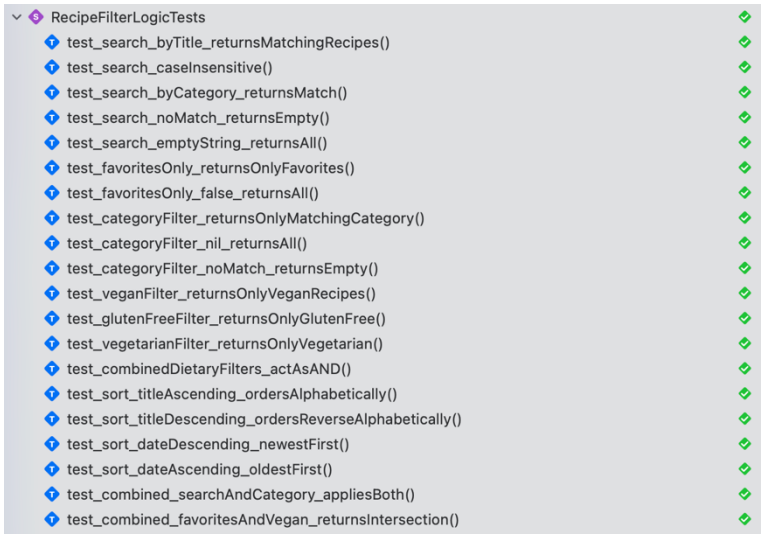


Рисунок 3.8 – Результати перевірки управління кулінарним контентом

Усі зафіксовані на рисунку 3.8 успішні операції з кулінарним каталогом підтверджують повну працездатність інтерфейсних компонентів. Після цього тестування перейшло до фази перевірки модулів автоматизації допоміжних процесів. Особлива увага приділялася логіці взаємодії зі списками покупок та синхронізації станів. Для оцінки коректності виконання операцій пакетного додавання масивів даних до хмари.

Таблиця 3.3 – Сценарії перевірки списків покупок та додаткових засобів

Ідентифікатор	Екран / Контекст	Вхідні дані / Дія	Реакція програми	Результат
1	2	3	4	5
TC-SHOP-01	Картка страви	Натискання кнопки перенесення інгредієнтів до списку покупок	Запускається пакетний запис, з'являється спливаюче повідомлення про успішне додавання	Успішно
TC-SHOP-02	Список покупок	Натискання на круглий маркер біля назви окремого продукту	Маркер заповнюється галочкою, текст перекреслюється, статус запису оновлюється на сервері	Успішно

Продовження таблиці 3.3

1	2	3	4	5
ТС-SHOP-03	Список покупок	Жест гортання вліво на рядку продукту та вибір опції вилучення	Елемент зникає з розмітки з анімацією, документ видаляється з відповідної колекції бази	Успішно
ТС-SHOP-04	Перерахунок	Зміна числового значення кількості порцій у картці страви	Вага кожного інгредієнта автоматично множиться на розрахований коефіцієнт без затримок	Успішно
ТС-SHOP-05	Генерація	Введення переліку наявних інгредієнтів та запуск запиту до ШІ	Відображається індикатор очікування, після відповіді сервера відкривається готова інструкція	Успішно

Проведене тестування підтверджує повну працездатність мобільного застосунку (Рисунок 3.9). Усі перевірені функції працюють стабільно, аварійних завершень роботи чи зависань інтерфейсу під час виконання розрахунків не виявлено. Отримані практичні результати повністю відповідають очікуваній поведінці системи, що свідчить про успішне завершення етапу конструювання та налагодження.

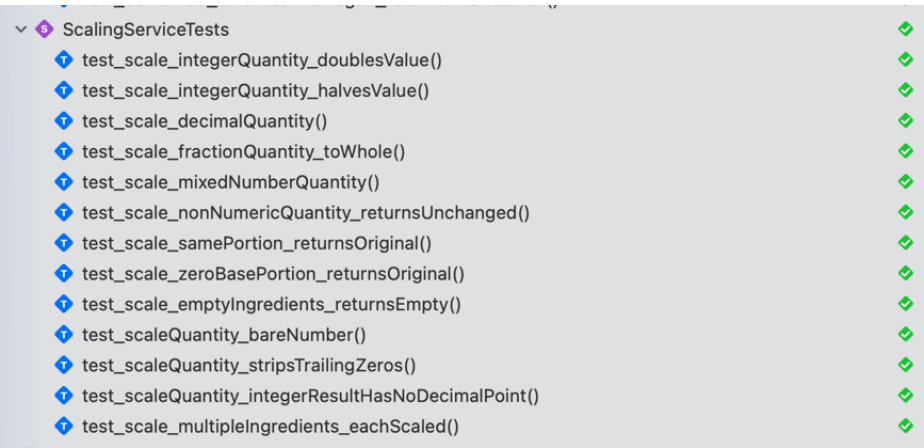


Рисунок 3.9 – Результати перевірки списків покупок та додаткових засобів

3.6.3 Аналіз результатів тестування мобільного застосунку

Верифікація охопила всі базові сценарії роботи системи та підтвердила правильність закладених архітектурних рішень. У процесі автоматизованого тестування виконано тридцять три незалежні перевірки. Модульні тести підтвердили абсолютну точність алгоритмів перерахунку інгредієнтів і надійність зчитування збережених документів із хмарної бази даних. Окремо перевірено роботу обчислювальних властивостей відбору страв. Логіка сортування та пошуку працює без збоїв.

Тести графічного інтерфейсу в середовищі симулятора успішно пройшли етапи ініціалізації та підтвердили відсутність блокувань головного потоку під час старту програми. Ручне проходження п'ятнадцяти ключових сценаріїв підтвердило правильну реакцію розмітки на дії людини. Перевірено механізми захисту від введення некоректних даних. Програма вчасно виводить зрозумілі текстові попередження та блокує надсилання порожніх форм. Переходи між екранами відбуваються плавно і не викликають візуальних помилок. Загальну аналітичну інформацію щодо результатів здійсненого тестування зведено в таблиці 3.4.

Таблиця 3.4 – Зведені результати верифікації мобільного застосунку

Категорія перевірки	Заплановано	Виконано	Успішно	Виявлено дефектів
Модульні тести бізнес-логіки	31	31	31	0
Автоматизовані тести інтерфейсу	2	2	2	0
Ручні сценарії верифікації	15	15	15	0
Разом	48	48	48	0

3.7 Висновки

У третьому розділі кваліфікаційної роботи здійснено опис практичної реалізації та перевірки мобільного застосунку. Програмний засіб створено для операційної системи iOS з використанням мови програмування Swift та декларативного фреймворку SwiftUI. Вибраний технологічний стек забезпечив написання чистого, безпечного коду та побудову сучасного графічного інтерфейсу. Реалізація внутрішньої логіки базується на архітектурному шаблоні декомпозиції. Такий підхід дозволив повністю відокремити правила обробки даних від візуального представлення. Управління станом екранів, мережеві запити та підготовка масивів інформації виконуються в ізольованих класах. Для забезпечення переходів за зовнішніми посиланнями розроблено окремий контролер маршрутизації. Розподіл прав доступу налаштовано відповідно до поточної ролі профілю.

Збереження кулінарних рецептів, списків покупок та ілюстрацій реалізовано за допомогою хмарної інфраструктури. Налаштовано безпечний обмін даними між мобільним клієнтом і документно-орієнтованим сховищем. Перетворення мережевих відповідей у внутрішні об'єкти відбувається автоматично за допомогою вбудованих протоколів кодування. Застосування транзакцій та пакетного запису гарантувало цілісність інформації під час одночасного оцінювання страв або масового додавання продуктів. Для підтвердження надійності створеної програми проведено комплексне тестування в середовищі симулятора. Автоматизовані перевірки підтвердили безпомилковість математичних розрахунків і стабільність фільтрації контенту. Ручне проходження сценаріїв довело зручність розмітки та правильну роботу захисних механізмів під час введення некоректних даних. Усі заплановані можливості реалізовано повною мірою. Програмний продукт працює стабільно і готовий до подальшого поширення.

ВИСНОВКИ

У кваліфікаційній роботі вирішено завдання створення мобільного застосунку для управління кулінарними рецептами та інгредієнтами. Мета роботи досягнута повною мірою. Розроблений програмний продукт дозволяє зручно організовувати інформацію про страви. Він генерує нові кулінарні інструкції за допомогою інструментів штучного інтелекту та автоматизує розрахунок потрібних продуктів.

Для досягнення мети було послідовно виконано низку етапів. На початковій стадії проведено аналіз предметної області та існуючих програмних рішень. Дослідження аналогів на ринку виявило недоліки наявних систем. Це допомогло сформувати точні вимоги до нового продукту. За допомогою методології моделювання IDEF0 побудовано модель системи. Для опису взаємодії аудиторії з програмою застосовано мову UML. Побудовано діаграми варіантів використання. Це дозволило чітко розмежувати права доступу для гостей, авторизованих осіб та адміністраторів.

На етапі проєктування обрано архітектуру з поділом на клієнтську та серверну частини. Графічну розмітку та бізнес-логіку розділено за шаблоном MVVM. Таке рішення спростило подальшу підтримку коду. Спроєктовано логічну структуру нереляційної бази даних. Вона складається з п'яти основних колекцій для збереження профілів, страв, списків покупок та збірок. Проєктування інтерфейсу виконано з дотриманням настанов Apple Human Interface Guidelines. Розроблено макети всіх екранів та інтерактивний прототип програми. Побудовано блок-схеми алгоритмів обробки даних та перерахунку ваги інгредієнтів.

Програмну реалізацію мобільного застосунку виконано мовою Swift. Графічний інтерфейс побудовано за допомогою декларативного фреймворку SwiftUI. Серверна частина базується на хмарних сервісах Firebase. Вони забезпечують авторизацію, збереження медіафайлів та синхронізацію даних у

реальному часі. Для автоматичного створення рецептів за введеним списком продуктів інтегровано зовнішній програмний інтерфейс OpenAI API. Розроблено алгоритм динамічного масштабування кількості інгредієнтів. Він автоматично змінює вагу продуктів при зміні кількості порцій на екрані. Для безпеки даних налаштовано систему ідентифікації та правила захисту бази даних.

Після написання коду проведено тестування програми. Перевірку внутрішньої логіки та точності математичних обчислень виконано за допомогою автоматизованих тестів XCTest. Функціональне тестування підтвердило правильну реакцію інтерфейсу на дії людини та безпомилкову роботу механізмів захисту бази даних. Результати випробувань довели стабільність роботи застосунку та його повну відповідність технічному завданню.

Впровадження розробленого програмного забезпечення надає низку переваг людям, які готують їжу вдома. Програма економить час на пошук потрібних записів та формування списків продуктів перед походом до магазину. Автоматичний перерахунок порцій виключає математичні помилки та допомагає уникнути зайвих витрат на продукти. Приватні рецепти надійно зберігаються у хмарному сховищі і залишаються доступними лише їх авторам. Застосування штучного інтелекту допомагає людям швидко знаходити ідеї для нових страв із тих продуктів, які вже є в наявності.

Отримані результати можна застосовувати і в інших галузях, зокрема для створення електронних меню ресторанів, програм дієтичного супроводу та сервісів доставки продуктів. Подальший розвиток проєкту передбачає додавання підрахунку калорій, календаря планування меню та розширення інструментів для обміну кулінарним досвідом.

Усі заплановані функціональні можливості реалізовано в повному обсязі. Програмний продукт успішно пройшов комплексне тестування на симуляторі Xcode та готовий до практичного застосування. Подальший розвиток застосунку може включати адаптацію інтерфейсу для iPad, локалізацію на інші мови та розширення функцій ШІ-модуля.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Recipe apps market growth analysis - size and forecast 2025–2029 | Technavio. URL: https://www.technavio.com/report/recipe-apps-market-industry-analysis (дата звернення: 12.02.2026).
2. Titu A. M. et al. Contributions on the application of IDEF0 modelling for collaborative design of information systems integration. 2024 16th International Conference on Electronics, Computers and Artificial Intelligence. URL: https://doi.org/10.1109/ecai61503.2024.10607500 (дата звернення: 27.04.2026).
3. FIPS PUB 183. Integration Definition for Function Modeling (IDEF0). Washington : National Institute of Standards and Technology, 1993. 116 p. URL: https://nvlpubs.nist.gov/nistpubs/legacy/fips/fipspub183.pdf (дата звернення: 14.02.2026).
4. What is NoSQL? Amazon Web Services. URL: https://aws.amazon.com/nosql/ (дата звернення: 15.02.2026).
5. ECMA-404. The JSON Data Interchange Syntax. 2nd ed. Geneva : Ecma International, 2017. 14 p. URL: https://ecma-international.org/publications-and-standards/standards/ecma-404/ (дата звернення: 17.02.2026).
6. Firebase Documentation. Firebase. URL: https://firebase.google.com/docs (дата звернення: 16.02.2026).
7. Karlsson J. Software requirements prioritizing. Proceedings of the Second International Conference on Requirements Engineering. 1996. URL: https://doi.org/10.1109/icre.1996.491435 (дата звернення: 22.04.2026).
8. Koç H. et al. UML diagrams in software engineering research: a systematic literature review. Proceedings. 2021. Vol. 74, № 1. P. 13. URL: https://doi.org/10.3390/proceedings2021074013 (дата звернення: 26.04.2026).
9. Tavares J. F., Costa Y. M. G., Colanzi T. E. Classification of UML diagrams to support software engineering education. 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops. 2021. URL:

https://doi.org/10.1109/ASEW52652.2021.00052 (дата звернення: 27.04.2026).

10. About the Unified Modeling Language Specification Version 2.5.1. Object Management Group. URL: https://www.omg.org/spec/UML/ (дата звернення: 16.02.2026).

11. Use-case diagrams. IBM Documentation. URL: https://www.ibm.com/docs/en/rhapsody/9.0.1?topic=diagrams-use-case (дата звернення: 16.02.2026).

12. OpenAI Developer Platform Documentation. OpenAI. URL: https://platform.openai.com/docs/ (дата звернення: 17.02.2026).

13. Designing for iOS. Apple Developer Documentation. URL: https://developer.apple.com/design/human-interface-guidelines/designing-for-ios (дата звернення: 02.04.2026).

14. Human Interface Guidelines. Apple Developer Documentation. URL: https://developer.apple.com/design/human-interface-guidelines/ (дата звернення: 16.02.2026).

15. SwiftUI. Apple Developer Documentation. URL: https://developer.apple.com/documentation/swiftui/ (дата звернення: 23.02.2026).

16. Nyabuto M. G. M. Architectural review of client-server models. International Journal of Scientific Research and Engineering Trends. 2024. Vol. 10, № 1. P. 139–143. URL: https://doi.org/10.61137/ijsret.vol.10.issue1.126 (дата звернення: 26.04.2026).

17. Wu L., Lu W., Chen C. Strengths and weaknesses of client-server and peer-to-peer network models in construction projects. International Journal of Construction Management. 2023. P. 1–15. URL: https://doi.org/10.1080/15623599.2023.2185950 (дата звернення: 01.05.2026).

18. Гамма Е., Гелм Р., Джонсон Р., Вліссідес Дж. Патерни проєктування. Харків : Фабула, 2021. 384 с. URL: https://fabulabook.com/product/paterny-proektuvannya/ (дата звернення: 05.03.2026).

19. Основні відомості про бази даних. Microsoft Support. URL: https://support.microsoft.com/uk-ua/topic/основні-відомості-про-бази-даних-

a849ac16-07c7-4a31-9948-3c8c94a7c204 (дата звернення: 09.03.2026).

20. Що таке ER-моделювання? Amazon Web Services. URL: https://aws.amazon.com/what-is/data-modeling/ (дата звернення: 12.03.2026).

21. Cloud Firestore documentation. Firebase. URL: https://firebase.google.com/docs/firestore (дата звернення: 12.03.2026).

22. Initialization. Swift Documentation. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/initialization/ (дата звернення: 14.03.2026).

23. Blinowski G., Ojdowska A., Przybylek A. Monolithic vs. microservice architecture: a performance and scalability evaluation. IEEE Access. 2022. Vol. 10. P. 20357–20374. URL: https://doi.org/10.1109/access.2022.3152803 (дата звернення: 14.03.2026).

24. What is Figma? Figma Help Center. URL: https://help.figma.com/hc/en-us/articles/14563969806359-What-is-Figma (дата звернення: 01.05.2026).

25. Bourque P. et al. The guide to the software engineering body of knowledge. IEEE Software. 1999. Vol. 16, № 6. P. 35–44. URL: https://doi.org/10.1109/52.805471 (дата звернення: 15.03.2026).

26. Codable. Apple Developer Documentation. URL: https://developer.apple.com/documentation/swift/codable (дата звернення: 15.03.2026).

27. Icon Composer Guide. Apple Developer Documentation. URL: https://developer.apple.com/icon-composer/ (дата звернення: 11.04.2026).

28. Competitive Usability Evaluations. Nielsen Norman Group. URL: https://www.nngroup.com/articles/competitive-usability-evaluations/ (дата звернення: 16.03.2026).

29. Concurrency. Swift Documentation. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/ (дата звернення: 16.03.2026).

30. Rana M. E., Saleh O. S. High assurance software architecture and design. System Assurances. 2022. P. 271–285. URL: https://doi.org/10.1016/b978-0-323-

90240-3.00015-1 (дата звернення: 26.04.2026).

31. Fowler M. Service Layer. Catalog of Patterns of Enterprise Application Architecture. URL: https://martinfowler.com/eaaCatalog/serviceLayer.html (дата звернення: 02.04.2026).

32. Firebase Authentication. Firebase Documentation. URL: https://firebase.google.com/docs/auth (дата звернення: 16.03.2026).

33. Transactions and batched writes. Firebase Documentation. URL: https://firebase.google.com/docs/firestore/manage-data/transactions (дата звернення: 30.04.2026).

34. 1. Бедратюк Л. П., Радельчук Г. І. Кваліфікаційна робота: Методичні настанови для здобувачів першого (бакалаврського) рівня вищої освіти спеціальності 121 «Інженерія програмного забезпечення». Хмельницький: ХНУ, 2023. 60 с. (дата звернення: 05.02.2026).

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

А.1 Програмний код точки входу та маршрутизація

CookBookApp.swift:

```
@main
struct CookBookApp: App {
    @StateObject var router = Router()

    init() {
        FirebaseApp.configure()
    }

    var body: some Scene {
        WindowGroup {
            MainView()
                .environmentObject(router)
                .onOpenURL { url in
                    router.handle(url: url)
                }
        }
    }
}
```

Router.swift

```
class Router: ObservableObject {
    @Published var deepLinkedRecipeID: String?

    func handle(url: URL) {
        guard url.scheme == "cookbook",
              url.host == "recipe",
              let components = URLComponents(url: url, resolvingAgainstBaseURL:
false),
              let id = components.queryItems?.first(where: { $0.name == "id"
})?.value
        else { return }

        deepLinkedRecipeID = id
    }
```

А.2 Програмний код сервісного шару

DatabaseService.swift:

```
class DatabaseService {
    static let shared = DatabaseService()
    private let db = Firestore.firestore()
    private let storage = Storage.storage()

    private init() {}

    // MARK: - Recipes

    func listenToRecipes(userId: String, onChange: @escaping ([RecipeListItem])
-> Void) -> ListenerRegistration {
        db.collection("users").document(userId).collection("recipes")
            .addSnapshotListener { snapshot, error in
                guard let documents = snapshot?.documents else {
                    print("Error fetching recipes: \(error?.localizedDescription
?? "unknown")")
                    return
                }
                onChange(documents.compactMap { try? $0.data(as:
RecipeListItem.self) })
            }
    }

    func fetchPublicRecipes(completion: @escaping ([RecipeListItem]) -> Void) {
        db.collectionGroup("recipes")
            .whereField("isPublic", isEqualTo: true)
            .getDocuments { snapshot, error in
                guard let documents = snapshot?.documents, error == nil else {
                    print("Error fetching public recipes:
\(error?.localizedDescription ?? "unknown")")
                    return
                }
                DispatchQueue.main.async {
                    completion(documents.compactMap { try? $0.data(as:
RecipeListItem.self) })
                }
            }
    }

    func saveRecipe(_ recipe: RecipeListItem, selectedImage: UIImage?, userId:
String, completion: @escaping () -> Void) {
        func write(imageURL: String?) {
            var toSave = recipe
            toSave.imageURL = imageURL
            db.collection("users").document(userId)
                .collection("recipes").document(recipe.id)
                .setData(toSave.asDictionary()) { error in
                    if let error = error {
                        print("Error saving recipe:
\(error.localizedDescription)")
                    } else {
                        completion()
                    }
                }
```

```
        }

        guard let image = selectedImage,
              let data = image.jpegData(compressionQuality: 0.8) else {
            write(imageURL: recipe.imageURL)
            return
        }

        let ref =
storage.reference().child("users/\(userId)/recipes/\(recipe.id).jpg")
        ref.putData(data) { _, error in
            if error == nil {
                ref.downloadURL { url, _ in write(imageURL: url?.absoluteString
?? recipe.imageURL) }
            } else {
                write(imageURL: recipe.imageURL)
            }
        }
    }

    func deleteRecipe(id: String, userId: String) {

db.collection("users").document(userId).collection("recipes").document(id).delet
e()
    }

    func setFavorite(_ recipe: RecipeListItem, userId: String) {
        db.collection("users").document(userId)
            .collection("recipes").document(recipe.id)
            .setData(recipe.asDictionary())
    }

    // MARK: - User

    func fetchUser(userId: String, completion: @escaping (User?) -> Void) {
        db.collection("users").document(userId).getDocument { snapshot, error in
            guard error == nil else {
                print("Error fetching user: \(error!.localizedDescription)")
                completion(nil)
                return
            }
            guard let data = snapshot?.data(),
                  let jsonData = try? JSONSerialization.data(withJSONObject:
data) else {
                DispatchQueue.main.async { completion(nil) }
                return
            }
            do {
                let user = try JSONDecoder().decode(User.self, from: jsonData)
                DispatchQueue.main.async { completion(user) }
            } catch {
                print("fetchUser decode error: \(error)")
                DispatchQueue.main.async { completion(nil) }
            }
        }
    }

    func saveUser(_ user: User) {
        db.collection("users").document(user.id).setData(user.asDictionary())
    }

    // MARK: - Products
```

```
    func saveProduct(_ item: ProductListItem, userId: String) {
        db.collection("users").document(userId)
            .collection("products").document(item.id)
            .setData(item.asDictionary())
    }

    func deleteProduct(id: String, userId: String) {

db.collection("users").document(userId).collection("products").document(id).dele
te()
    }

    // MARK: - Moderation
    // Note: collectionGroup queries on "status" require a Firestore composite
index.
    // Create it in the Firebase console: Collection: recipes, Field: status
(Ascending).

    /// Owner action: sets status → pending and clears isPublic.
    func submitForModeration(recipeId: String, userId: String) {
        db.collection("users").document(userId)
            .collection("recipes").document(recipeId)
            .updateData([
                "status": RecipeStatus.pending.rawValue,
                "isPublic": false
            ]) { error in
                if let error = error {
                    print("submitForModeration error:
\(error.localizedDescription)")
                }
            }
    }

    /// Admin action: fetch all recipes currently pending review across all
users.
    func fetchPendingRecipes(completion: @escaping (Result<[RecipeListItem],
Error>) -> Void) {
        db.collectionGroup("recipes")
            .whereField("status", isEqualTo: RecipeStatus.pending.rawValue)
            .getDocuments { snapshot, error in
                if let error = error {
                    print("fetchPendingRecipes query error: \(error)")
                    DispatchQueue.main.async { completion(.failure(error)) }
                    return
                }
                let docs = snapshot?.documents ?? []
                let recipes: [RecipeListItem] = docs.compactMap { doc in
                    let data = doc.data()
                    guard let jsonData = try?
JSONSerialization.data(withJSONObject: data) else {
                        print("fetchPendingRecipes: JSON serialization failed
for doc \(doc.documentID)")
                        return nil
                    }
                    do {
                        return try JSONDecoder().decode(RecipeListItem.self,
from: jsonData)
                    } catch {
                        print("fetchPendingRecipes: decode error for doc
\(doc.documentID): \(error)")
                        return nil
                    }
                }
```

```
                DispatchQueue.main.async { completion(.success(recipes)) }
            }
    }

    /// Admin action: set status to approved or rejected, updating isPublic
accordingly.
    func updateRecipeStatus(recipe: RecipeListItem, newStatus: RecipeStatus,
completion: @escaping () -> Void) {
        guard let ownerId = recipe.ownerId else {
            print("updateRecipeStatus: recipe has no ownerId")
            return
        }
        let isPublic = newStatus == .approved
        db.collection("users").document(ownerId)
            .collection("recipes").document(recipe.id)
            .updateData([
                "status": newStatus.rawValue,
                "isPublic": isPublic
            ]) { error in
                if let error = error {
                    print("updateRecipeStatus error:
\(error.localizedDescription)")
                } else {
                    DispatchQueue.main.async { completion() }
                }
            }
    }

    /// Public feed: all recipes with approved status, regardless of owner.
    func fetchPublicFeed(completion: @escaping ([RecipeListItem]) -> Void) {
        db.collectionGroup("recipes")
            .whereField("status", isEqualTo: RecipeStatus.approved.rawValue)
            .getDocuments { snapshot, error in
                if let error = error {
                    print("fetchPublicFeed query error: \(error)")
                    DispatchQueue.main.async { completion([]) }
                    return
                }
                let docs = snapshot?.documents ?? []
                let recipes: [RecipeListItem] = docs.compactMap { doc in
                    let data = doc.data()
                    guard let jsonData = try?
JSONSerialization.data(withJSONObject: data) else {
                        print("fetchPublicFeed: JSON serialization failed for
doc \(doc.documentID)")
                        return nil
                    }
                    do {
                        return try JSONDecoder().decode(RecipeListItem.self,
from: jsonData)
                    } catch {
                        print("fetchPublicFeed: decode error for doc
\(doc.documentID): \(error)")
                        return nil
                    }
                }
                DispatchQueue.main.async { completion(recipes) }
            }
    }

    /// Fetch a small curated set of approved, human-authored recipes for the
home carousel.
```

```
    /// Queries approved recipes, over-fetches slightly, then filters isAI ==
false client-side.
    func fetchEditorsChoice(limit: Int = 5) async -> [RecipeListItem] {
        do {
            let snapshot = try await db.collectionGroup("recipes")
                .whereField("status", isEqualTo: RecipeStatus.approved.rawValue)
                .limit(to: max(limit * 2, 10))
                .getDocuments()
            return snapshot.documents
                .compactMap { doc -> RecipeListItem? in
                    guard let jsonData = try?
JSONSerialization.data(withJSONObject: doc.data()) else { return nil }
                    return try? JSONDecoder().decode(RecipeListItem.self, from:
jsonData)
                }
                .filter { !$0.isAI }
                .prefix(limit)
                .map { $0 }
        } catch {
            print("fetchEditorsChoice error: \(error)")
            return []
        }
    }

    // MARK: - Products

    func addProducts(_ items: [ProductListItem], userId: String) {
        let batch = db.batch()
        for item in items {
            let ref =
db.collection("users").document(userId).collection("products").document()
            let copy = ProductListItem(id: ref.documentID, title: item.title,
quantity: item.quantity, isDone: false)
            batch.setData(copy.asDictionary(), forDocument: ref)
        }
        batch.commit { error in
            if let error = error {
                print("Error adding products: \(error.localizedDescription)")
            }
        }
    }

    // MARK: - Decode helper

    /// Converts a raw Firestore data dictionary to a Codable model via
JSONDecoder.
    /// Produces explicit errors instead of silent compactMap drops.
    private func decode<T: Decodable>(_ type: T.Type,
                                      from data: [String: Any],
                                      docId: String = "") throws -> T {
        let jsonData = try JSONSerialization.data(withJSONObject: data)
        do {
            return try JSONDecoder().decode(type, from: jsonData)
        } catch {
            throw DecodingError.dataCorrupted(
                .init(codingPath: [], debugDescription: "doc '\(docId)':
\(error)")
            )
        }
    }

    // MARK: - Collections
    // Stored under users/{userId}/collections
```

```
    func saveCollection(_ collection: RecipeCollection, userId: String) async
throws {
        try await db.collection("users").document(userId)
            .collection("collections").document(collection.id)
            .setData(collection.asDictionary())
    }

    func deleteCollection(id: String, userId: String) async throws {
        try await db.collection("users").document(userId)
            .collection("collections").document(id)
            .delete()
    }

    func fetchCollections(userId: String) async throws -> [RecipeCollection] {
        let snapshot = try await db.collection("users").document(userId)
            .collection("collections").getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(RecipeCollection.self, from: doc.data(), docId:
doc.documentID)
        }
    }

    /// Atomically appends recipeId to the collection's recipeIds array.
    func addRecipeToCollection(recipeId: String, collectionId: String, userId:
String) async throws {
        try await db.collection("users").document(userId)
            .collection("collections").document(collectionId)
            .updateData(["recipeIds": FieldValue.arrayUnion([recipeId])])
    }

    // MARK: - Public Collections
    // Global path: /collections/{collectionId}
    // Each document carries isPublic: true so Firestore Security Rules can gate
write access to admins.
    //
    // Favourite Collections
    // A user's saved public collections are tracked as an array of IDs at:
    //   users/{userId}/  →  field "favouriteCollectionIds: [String]"
    // This avoids duplicating the full collection document per user.

    /// Admin: create or overwrite a global public collection.
    func savePublicCollection(_ collection: RecipeCollection) async throws {
        var toSave = collection
        toSave.isPublic = true          // enforce flag regardless of caller
        try await db.collection("collections").document(toSave.id)
            .setData(toSave.asDictionary())
    }

    /// Admin: delete a global public collection.
    func deletePublicCollection(id: String) async throws {
        try await db.collection("collections").document(id).delete()
    }

    /// Fetch all global public collections, ordered by title.
    func fetchPublicCollections() async throws -> [RecipeCollection] {
        let snapshot = try await db.collection("collections")
            .whereField("isPublic", isEqualTo: true)
            .order(by: "title")
            .getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(RecipeCollection.self, from: doc.data(), docId:
doc.documentID)
```

```
        }
    }

    /// Fetch a single public collection by ID (used when opening a deep-linked
collection).
    func fetchPublicCollection(id: String) async throws -> RecipeCollection? {
        let doc = try await
db.collection("collections").document(id).getDocument()
        guard doc.exists, let data = doc.data() else { return nil }
        return try decode(RecipeCollection.self, from: data, docId: id)
    }

    /// Admin: atomically append a recipe to a global public collection.
    func addRecipeToPublicCollection(recipeId: String, collectionId: String)
async throws {
        try await db.collection("collections").document(collectionId)
            .updateData(["recipeIds": FieldValue.arrayUnion([recipeId])])
    }

    // MARK: Favourite Collections (per-user bookmarks of public collections)

    /// Returns the IDs of global collections the user has saved as favourites.
    func fetchFavouriteCollectionIds(userId: String) async throws -> [String] {
        let doc = try await
db.collection("users").document(userId).getDocument()
        return doc.data()?["favouriteCollectionIds"] as? [String] ?? []
    }

    /// Add a public collection to the user's favourites (idempotent via
arrayUnion).
    func addFavouriteCollection(collectionId: String, userId: String) async
throws {
        try await db.collection("users").document(userId)
            .setData(["favouriteCollectionIds":
FieldValue.arrayUnion([collectionId])],
                     merge: true)
    }

    /// Remove a public collection from the user's favourites.
    func removeFavouriteCollection(collectionId: String, userId: String) async
throws {
        try await db.collection("users").document(userId)
            .updateData(["favouriteCollectionIds":
FieldValue.arrayRemove([collectionId])])
    }

    /// Convenience: fetch the full RecipeCollection documents for all of a
user's favourites.
    func fetchFavouriteCollections(userId: String) async throws ->
[RecipeCollection] {
        let ids = try await fetchFavouriteCollectionIds(userId: userId)
        guard !ids.isEmpty else { return [] }
        // Firestore `in` limit is 10; batch into chunks.
        let chunks = stride(from: 0, to: ids.count, by: 10).map {
Array(ids[$0..<min($0+10, ids.count)]) }
        var result: [RecipeCollection] = []
        for chunk in chunks {
            let snapshot = try await db.collection("collections")
                .whereField(FieldPath.documentID(), in: chunk)
                .getDocuments()
            let batch = snapshot.documents.compactMap { doc in
                try? decode(RecipeCollection.self, from: doc.data(), docId:
doc.documentID)
```

```
            }
            result.append(contentsOf: batch)
        }
        return result
    }

    // MARK: - Photos
    // Stored under users/{userId}/photos; images in Storage at
users/{userId}/photos/{id}.jpg

    /// Uploads image (if provided), then saves the PhotoItem document.
    /// Returns the saved item with the resolved imageURL.
    @discardableResult
    func savePhotoItem(_ photo: PhotoItem, image: UIImage?, userId: String)
async throws -> PhotoItem {
        var toSave = photo

        if let image = image, let data = image.jpegData(compressionQuality: 0.8)
{
            let ref =
storage.reference().child("users/\(userId)/photos/\(photo.id).jpg")
            _ = try await ref.putDataAsync(data)
            toSave.imageURL = try await ref.downloadURL().absoluteString
        }

        try await db.collection("users").document(userId)
            .collection("photos").document(toSave.id)
            .setData(toSave.asDictionary())

        return toSave
    }

    func deletePhotoItem(id: String, userId: String) async throws {
        try await db.collection("users").document(userId)
            .collection("photos").document(id)
            .delete()
    }

    func fetchPhotos(userId: String) async throws -> [PhotoItem] {
        let snapshot = try await db.collection("users").document(userId)
            .collection("photos")
            .order(by: "createdDate", descending: true)
            .getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(PhotoItem.self, from: doc.data(), docId: doc.documentID)
        }
    }

    // MARK: - Chefs
    // Global top-level "chefs" collection (not per-user).

    func saveChef(_ chef: Chef) async throws {
        try await db.collection("chefs").document(chef.id)
            .setData(chef.asDictionary())
    }

    func fetchChefs() async throws -> [Chef] {
        let snapshot = try await db.collection("chefs")
            .order(by: "name")
            .getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(Chef.self, from: doc.data(), docId: doc.documentID)
        }
```

```
    }

    func fetchChefById(_ id: String) async throws -> Chef? {
        let doc = try await db.collection("chefs").document(id).getDocument()
        guard doc.exists, let data = doc.data() else { return nil }
        return try decode(Chef.self, from: data, docId: id)
    }

    /// Fetches all recipes across all users whose chefId matches.
    /// Requires a Firestore collection-group index on the "chefId" field.
    func fetchRecipesByChef(_ chefId: String) async throws -> [RecipeListItem] {
        let snapshot = try await db.collectionGroup("recipes")
            .whereField("chefId", isEqualTo: chefId)
            .getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(RecipeListItem.self, from: doc.data(), docId:
doc.documentID)
        }
    }

    // MARK: - Admin: Users

    /// Returns all user records. Caller must be an admin; enforce via Firestore
Security Rules.
    func fetchAllUsers() async throws -> [User] {
        let snapshot = try await db.collection("users").getDocuments()
        return snapshot.documents.compactMap { doc -> User? in
            guard let jsonData = try? JSONSerialization.data(withJSONObject:
doc.data()) else { return nil }
            return try? JSONDecoder().decode(User.self, from: jsonData)
        }
    }

    // MARK: - Ratings
    // Stored at ratings/{recipeId} with a "userRatings" map {userId: stars}.
    // A transaction ensures the running average and count stay consistent.

    /// Records or updates the current user's star rating for a recipe.
    /// Atomically recalculates the average stored in ratings/{recipeId} and
    /// writes the new average + count back to the recipe document.
    /// - Parameters:
    ///   - recipeId: The recipe's Firestore document ID.
    ///   - ownerId:  The recipe owner's user ID (needed to locate the recipe
doc). Pass nil to skip recipe-doc writeback.
    ///   - stars:    Rating value 1–5.
    func submitRating(recipeId: String, ownerId: String?, stars: Int) async
throws {
        guard let userId = Auth.auth().currentUser?.uid else {
            throw NSError(domain: "CookBook", code: 401,
                          userInfo: [NSLocalizedDescriptionKey: "User not signed
in"])
        }
        let clampedStars = max(1, min(5, stars))
        let ratingRef = db.collection("ratings").document(recipeId)
        let recipeRef: DocumentReference? = ownerId.map { oid in

db.collection("users").document(oid).collection("recipes").document(recipeId)
        }

        _ = try await db.runTransaction { transaction, errorPointer in
            let snapshot: DocumentSnapshot
            do {
                snapshot = try transaction.getDocument(ratingRef)
```

```
            } catch let fetchError as NSError {
                errorPointer?.pointee = fetchError
                return nil
            }

            var userRatings: [String: Int] = [:]
            if snapshot.exists,
               let existing = snapshot.data()?["userRatings"] as? [String: Int]
{
                userRatings = existing
            }

            userRatings[userId] = clampedStars
            let total  = userRatings.values.reduce(0, +)
            let count  = userRatings.count
            let avg    = count > 0 ? Double(total) / Double(count) : 0.0

            transaction.setData([
                "userRatings": userRatings,
                "average": avg,
                "count": count
            ], forDocument: ratingRef)

            // Write computed average back to the recipe document so
RecipeListItem stays in sync.
            if let recipeRef = recipeRef {
                transaction.setData(["rating": avg, "ratingCount": count],
                                    forDocument: recipeRef, merge: true)
            }

            return nil
        }
    }

    /// Fetches recipes by their IDs using a collectionGroup query.
    /// Requires the `id` field to be stored in each recipe document (it is, via
`asDictionary()`).
    /// Firestore `in` queries are limited to 10 values; only the first 10 IDs
are queried.
    func fetchRecipesForCollection(ids: [String]) async throws ->
[RecipeListItem] {
        guard !ids.isEmpty else { return [] }
        let batch = Array(ids.prefix(10))
        let snapshot = try await db.collectionGroup("recipes")
            .whereField("id", in: batch)
            .getDocuments()
        return snapshot.documents.compactMap { doc in
            try? decode(RecipeListItem.self, from: doc.data(), docId:
doc.documentID)
        }
    }
}
```

OpenAIService.swift

```
enum OpenAIError: LocalizedError {
    case httpError(statusCode: Int, body: String)
    case emptyResponse
    case invalidJSON(String)

    var errorDescription: String? {
        switch self {
```

```
        case .httpError(let code, _): return "API error (HTTP \(code)). Check
your API key or quota."
        case .emptyResponse:          return "The AI returned an empty
response."
        case .invalidJSON(let msg):   return "Could not parse AI response:
\(msg)"
        }
    }
}

// MARK: - Service

struct OpenAIService {
    static let shared = OpenAIService()

    // Key is resolved at runtime from Info.plist (see ConfigurationManager).
    private var apiKey: String { ConfigurationManager.openAIApiKey }
    private let model   = "gpt-4o-mini"
    private let endpoint = URL(string:
"https://api.openai.com/v1/chat/completions")!

    private init() {}

    // MARK: - Public API

    func generateRecipe(
        prompt: String,
        ingredients: String,
        isVegan: Bool,
        isGlutenFree: Bool,
        isVegetarian: Bool,
        isDairyFree: Bool,
        isNutFree: Bool,
        isKidsFree: Bool
    ) async throws -> RecipeListItem {

        let systemPrompt = """
        You are a professional chef assistant. \
        Respond ONLY with a single valid JSON object – no markdown fences, no
extra text.

        Required schema (all fields mandatory):
        {
          "title": "string",
          "instructions": "string – numbered steps, each on its own line",
          "categories": "one of: Breakfast, Soup, Salad, Appetizer, Main Course,
Side Dish, Dessert, Snack, Drink, Baking, Pasta, Pizza, Seafood, Quick & Easy",
          "difficulty": "one of: Easy, Medium, Hard",
          "isGlutenFree": boolean,
          "isVegan": boolean,
          "isVegetarian": boolean,
          "isDairyFree": boolean,
          "isNutFree": boolean,
          "isKidsFree": boolean,
          "ingredients": [
            { "title": "string", "quantity": "number + unit, e.g. 200g or 2
cups" }
          ]
        }

        Rules:
        - ingredients must have at least 3 items with non-empty title and
quantity
```

```
        - honour all dietary restrictions exactly
        - instructions must be clear, detailed, and step-by-step
        """

        var userParts: [String] = ["Create a recipe"]
        if !prompt.trimmingCharacters(in: .whitespaces).isEmpty {
            userParts.append("for: \(prompt)")
        }
        if !ingredients.trimmingCharacters(in: .whitespaces).isEmpty {
            userParts.append("using these ingredients if possible:
\(ingredients)")
        }

        var filters: [String] = []
        if isVegan       { filters.append("vegan") }
        if isGlutenFree  { filters.append("gluten-free") }
        if isVegetarian  { filters.append("vegetarian") }
        if isDairyFree   { filters.append("dairy-free") }
        if isNutFree     { filters.append("nut-free") }
        if isKidsFree    { filters.append("kid-friendly") }
        if !filters.isEmpty {
            userParts.append("Dietary requirements: \(filters.joined(separator:
", "))")
        }

        let userMessage = userParts.joined(separator: ". ")

        let requestBody: [String: Any] = [
            "model": model,
            "response_format": ["type": "json_object"],
            "temperature": 0.8,
            "messages": [
                ["role": "system", "content": systemPrompt],
                ["role": "user",   "content": userMessage]
            ]
        ]

        var request = URLRequest(url: endpoint)
        request.httpMethod = "POST"
        request.setValue("Bearer \(apiKey)", forHTTPHeaderField:
"Authorization")
        request.setValue("application/json",  forHTTPHeaderField: "Content-
Type")
        request.httpBody = try JSONSerialization.data(withJSONObject:
requestBody)

        let (data, response) = try await URLSession.shared.data(for: request)

        if let http = response as? HTTPURLResponse, http.statusCode != 200 {
            let body = String(data: data, encoding: .utf8) ?? ""
            throw OpenAIError.httpError(statusCode: http.statusCode, body: body)
        }

        return try parseRecipe(from: data)
    }

    // MARK: - Parsing

    private func parseRecipe(from data: Data) throws -> RecipeListItem {

        // Decode OpenAI wrapper
        struct ChatResponse: Decodable {
            struct Choice: Decodable {
```

```
                struct Message: Decodable { let content: String }
                let message: Message
            }
            let choices: [Choice]
        }

        let chat: ChatResponse
        do {
            chat = try JSONDecoder().decode(ChatResponse.self, from: data)
        } catch {
            throw OpenAIError.invalidJSON("Outer response:
\(error.localizedDescription)")
        }

        guard let raw = chat.choices.first?.message.content,
              let contentData = raw.data(using: .utf8) else {
            throw OpenAIError.emptyResponse
        }

        // Decode inner recipe JSON
        struct AIIngredient: Decodable {
            let title: String
            let quantity: String
        }
        struct AIRecipe: Decodable {
            let title: String
            let instructions: String
            let categories: String
            let difficulty: String
            let isGlutenFree: Bool?
            let isVegan: Bool?
            let isVegetarian: Bool?
            let isDairyFree: Bool?
            let isNutFree: Bool?
            let isKidsFree: Bool?
            let ingredients: [AIIngredient]
        }

        let ai: AIRecipe
        do {
            ai = try JSONDecoder().decode(AIRecipe.self, from: contentData)
        } catch {
            throw OpenAIError.invalidJSON("Recipe payload:
\(error.localizedDescription)")
        }

        let ingredients = ai.ingredients.map {
            ProductListItem(id: UUID().uuidString, title: $0.title, quantity:
$0.quantity, isDone: false)
        }

        return RecipeListItem(
            id: UUID().uuidString,
            ownerId: nil,
            title: ai.title,
            ingredients: ingredients,
            instructions: ai.instructions,
            categories: RecipeCategory(rawValue: ai.categories) ?? .main,
            difficulty: Difficulty(rawValue: ai.difficulty) ?? .easy,
            imageURL: nil,
            createdDate: Date().timeIntervalSince1970,
            isFavorite: false,
            isPublic: false,
```

```
            isGlutenFree: ai.isGlutenFree ?? false,
            isVegan: ai.isVegan ?? false,
            isVegetarian: ai.isVegetarian ?? false,
            isDairyFree: ai.isDairyFree ?? false,
            isNutFree: ai.isNutFree ?? false,
            isKidsFree: ai.isKidsFree ?? false,
            isAI: true
        )
    }
}
```

А.3 Програмний код бізнес-логіки

MainViewViewModel.swift:

```
class MainViewViewModel: ObservableObject {
    @Published var currentUserId: String = ""
    @Published var isGuest: Bool = false
    /// True once the signed-in user's role has been confirmed as `.admin`.
    @Published var isAdmin: Bool = false
    private var handler: AuthStateDidChangeListenerHandle?

    init() {
        self.handler = Auth.auth().addStateDidChangeListener { [weak self] _,
user in
            DispatchQueue.main.async {
                self?.currentUserId = user?.uid ?? ""
                // Sign-in clears guest mode automatically.
                if user != nil { self?.isGuest = false }
            }
            if let uid = user?.uid {
                // Fetch role asynchronously and publish the result.
                DatabaseService.shared.fetchUser(userId: uid) { [weak self]
fetched in
                    DispatchQueue.main.async {
                        self?.isAdmin = fetched?.role == .admin
                    }
                }
            } else {
                DispatchQueue.main.async { self?.isAdmin = false }
            }
        }
    }

    public var isSignedIn: Bool {
        return Auth.auth().currentUser != nil
    }

    func continueAsGuest() { isGuest = true }
    func exitGuest()        { isGuest = false }
}
```

LoginViewViewModel.swift

RecipeListViewViewModel.swift

```
enum SortOption: String, CaseIterable, Identifiable {
    var id: Self { self }
    case titleAscending = "Title (A-Z)"
    case titleDescending = "Title (Z-A)"
    case dateCreatedDescending = "Newest First"
    case dateCreatedAscending = "Oldest First"
}

class RecipeListViewViewModel: ObservableObject {
    @Published var showingNewItemView = false
    @Published var searchText = ""
    @Published var recipes: [RecipeListItem] = []
    @Published var sortOption: SortOption = .dateCreatedDescending
    @Published var shouldShowFavoritesOnly: Bool = false

    // Category chip filter – used by OverviewView
    @Published var categoryFilter: RecipeCategory? = nil

    // Dietary toggles – used by SearchView's sort/filter sheet
    @Published var filterVegan: Bool = false
    @Published var filterGlutenFree: Bool = false
    @Published var filterVegetarian: Bool = false
    @Published var filterDairyFree: Bool = false
    @Published var filterNutFree: Bool = false
    @Published var filterKidsFree: Bool = false

    let userId: String
    private var firestoreListener: ListenerRegistration?
    private var cancellables = Set<AnyCancellable>()
    private let loadPublicRecipes: Bool

    init(userId: String, shouldShowFavoritesOnly: Bool = false,
loadPublicRecipes: Bool = false) {
        self.userId = userId
        self.loadPublicRecipes = loadPublicRecipes
        self.shouldShowFavoritesOnly = shouldShowFavoritesOnly
        $searchText
            .debounce(for: .milliseconds(300), scheduler: RunLoop.main)
            .removeDuplicates()
            .sink { _ in }
            .store(in: &cancellables)

        if loadPublicRecipes {
            fetchPublicRecipes()
        } else {
            fetchRecipes()
        }
    }

    deinit {
        firestoreListener?.remove()
    }

    var displayedRecipes: [RecipeListItem] {
        var list = recipes

        if !loadPublicRecipes {
            if shouldShowFavoritesOnly {
                list = list.filter { $0.isFavorite }
            }
        }
```

```
        if !searchText.isEmpty {
            let q = searchText.lowercased()
            list = list.filter {
                $0.title.lowercased().contains(q) ||
                $0.categories.rawValue.lowercased().contains(q)
            }
        }

        // Category chip filter
        if let category = categoryFilter {
            list = list.filter { $0.categories == category }
        }

        // Dietary filters
        if filterVegan       { list = list.filter { $0.isVegan } }
        if filterGlutenFree  { list = list.filter { $0.isGlutenFree } }
        if filterVegetarian  { list = list.filter { $0.isVegetarian } }
        if filterDairyFree   { list = list.filter { $0.isDairyFree } }
        if filterNutFree     { list = list.filter { $0.isNutFree } }
        if filterKidsFree    { list = list.filter { $0.isKidsFree } }

        switch sortOption {
        case .titleAscending:
            list.sort { $0.title < $1.title }
        case .titleDescending:
            list.sort { $0.title > $1.title }
        case .dateCreatedDescending:
            list.sort { $0.createdDate > $1.createdDate }
        case .dateCreatedAscending:
            list.sort { $0.createdDate < $1.createdDate }
        }

        return list
    }

    func updateFavorite(for id: String, isFavorite: Bool) {
        if let index = recipes.firstIndex(where: { $0.id == id }) {
            recipes[index].isFavorite = isFavorite
        }
    }

    private func fetchRecipes() {
        firestoreListener = DatabaseService.shared.listenToRecipes(userId:
userId) { [weak self] recipes in
            self?.recipes = recipes
        }
    }

    func fetchPublicRecipes() {
        DatabaseService.shared.fetchPublicFeed { [weak self] recipes in
            self?.recipes = recipes
        }
    }

    func delete(id: String) {
        DatabaseService.shared.deleteRecipe(id: id, userId: userId)
    }
}
```

RecipeFormViewModel.swift

```
class RecipeFormViewModel: ObservableObject {
    @Published var title: String = ""
    @Published var instructions: String = ""
    @Published var ingredients: [ProductListItem] = []
    @Published var category: RecipeCategory = .main
    @Published var difficulty: Difficulty = .easy
    @Published var imageURL: String? = nil
    @Published var isPublic: Bool = false
    @Published var isGlutenFree: Bool = false
    @Published var isVegan: Bool = false
    @Published var isVegetarian: Bool = false
    @Published var isDairyFree: Bool = false
    @Published var isNutFree: Bool = false
    @Published var isKidsFree: Bool = false

    // Chef assignment (admin-only)
    @Published var chefId: String? = nil
    @Published var availableChefs: [Chef] = []

    @Published var showAlert: Bool = false

    private let recipeId: String
    let isEditing: Bool
    /// When true the recipe skips the moderation queue on first save:
    /// status → .approved, isPublic → true.
    let isAdmin: Bool
    private let initialCreatedDate: TimeInterval
    private var initialIsFavorite: Bool = false

    init(isAdmin: Bool = false) {
        self.recipeId = UUID().uuidString
        self.isEditing = false
        self.isAdmin = isAdmin
        self.initialCreatedDate = Date().timeIntervalSince1970
    }

    init(recipe: RecipeListItem, isAdmin: Bool = false) {
        self.recipeId = recipe.id
        self.isEditing = true
        self.isAdmin = isAdmin
        self.initialCreatedDate = recipe.createdDate
        self.initialIsFavorite = recipe.isFavorite

        self.title = recipe.title
        self.instructions = recipe.instructions
        self.ingredients = recipe.ingredients
        self.category = recipe.categories
        self.difficulty = recipe.difficulty
        self.imageURL = recipe.imageURL
        self.isPublic = recipe.isPublic
        self.isGlutenFree = recipe.isGlutenFree
        self.isVegan = recipe.isVegan
        self.isVegetarian = recipe.isVegetarian
        self.isDairyFree = recipe.isDairyFree
        self.isNutFree = recipe.isNutFree
        self.isKidsFree = recipe.isKidsFree
        self.chefId = recipe.chefId
    }
```

```
    /// Fetch chefs from Firestore (call this when the form is presented to an
admin).
    @MainActor
    func loadChefs() async {
        do {
            availableChefs = try await DatabaseService.shared.fetchChefs()
        } catch {
            print("RecipeFormViewModel.loadChefs error: \(error)")
        }
    }

    func save(selectedImage: UIImage?, completion: @escaping () -> Void) {
        guard canSave else {
            showAlert = true
            return
        }

        guard let uId = Auth.auth().currentUser?.uid else { return }

        // Admins publishing a new recipe bypass moderation: mark approved +
public immediately.
        // When editing an existing recipe keep whatever status it already has.
        let publishedPublic  = isAdmin && !isEditing ? true  : isPublic
        let publishedStatus: RecipeStatus = isAdmin && !isEditing ? .approved :
.draft

        var recipe = RecipeListItem(
            id: recipeId,
            ownerId: uId,
            title: title,
            ingredients: ingredients,
            instructions: instructions,
            categories: category,
            difficulty: difficulty,
            imageURL: imageURL,
            createdDate: initialCreatedDate,
            isFavorite: initialIsFavorite,
            isPublic: publishedPublic,
            isGlutenFree: isGlutenFree,
            isVegan: isVegan,
            isVegetarian: isVegetarian,
            isDairyFree: isDairyFree,
            isNutFree: isNutFree,
            isKidsFree: isKidsFree
        )
        recipe.chefId  = chefId
        recipe.status  = publishedStatus

        DatabaseService.shared.saveRecipe(recipe, selectedImage: selectedImage,
userId: uId, completion: completion)
    }

    var canSave: Bool {
        !title.trimmingCharacters(in: .whitespaces).isEmpty &&
        !instructions.trimmingCharacters(in: .whitespaces).isEmpty &&
        !ingredients.isEmpty
    }
}
```

А.4 Програмний код графічного інтерфейсу

OverviewView.swift

```
struct OverviewView: View {
    let userId: String

    @StateObject private var viewModel: RecipeListViewViewModel
    @EnvironmentObject private var mainViewModel: MainViewViewModel

    @State private var userName: String = ""
    @State private var editorsChoiceRecipes: [RecipeListItem] = []

    @State private var selectedCategory: RecipeCategory? = nil

    @State private var carouselPage: Int = 0

    init(userId: String, loadPublicRecipes: Bool = false) {
        self.userId = userId
        _viewModel = StateObject(
            wrappedValue: RecipeListViewViewModel(
                userId: userId,
                loadPublicRecipes: loadPublicRecipes
            )
        )
    }

    var body: some View {
        NavigationView {
            ScrollView {
                VStack(spacing: 20) {
                    header
                    searchBar
                    editorsChoiceCarousel
                    topCategories
                    RecipeListG(viewModel: viewModel)
                }
                .padding(.horizontal)
                .padding(.bottom)
            }
        }
        .task {
            if !userId.isEmpty {
                DatabaseService.shared.fetchUser(userId: userId) { user in
                    userName = user?.name ?? ""
                }
            }

            editorsChoiceRecipes = await
DatabaseService.shared.fetchEditorsChoice(limit: 5)
        }
    }

    private var header: some View {
        HStack {
            HStack(spacing: 12) {
                Circle()
                    .fill(Color("grey-100"))
```

```
                    .frame(width: 46, height: 46)
                    .overlay(
                        Image(systemName: "person.fill")
                            .foregroundColor(.gray)
                    )

                VStack(alignment: .leading, spacing: 4) {
                    Text("Hello, \(userName.isEmpty ? "there" : userName)!")
                        .fontWeight(.medium)
                        .font(.system(size: 16))

                    Text("Let's start cooking!")
                        .fontWeight(.regular)
                        .font(.system(size: 12))
                        .foregroundStyle(Color.gray)
                }
            }

            Spacer()

            Button {

            } label: {
                Image(systemName: "bell.fill")
                    .foregroundColor(.black)
                    .font(.system(size: 19, weight: .medium))
                    .frame(width: 46, height: 46)
                    .background(Color("grey-100"))
                    .cornerRadius(100)
            }
        }
    }

    private var searchBar: some View {
        HStack(spacing: 8) {
            HStack {
                Image(systemName: "magnifyingglass")
                    .foregroundColor(Color("brown-500"))
                    .padding(.leading, 10)

                TextField("Search recipes…", text: $viewModel.searchText)
                    .padding(.vertical, 10)
                    .autocorrectionDisabled()
                    .textInputAutocapitalization(.never)
            }
            .background(Color("grey-100"))
            .cornerRadius(10)

            if !viewModel.searchText.isEmpty {
                Button {
                    viewModel.searchText = ""
                } label: {
                    Image(systemName: "xmark.circle.fill")
                        .foregroundColor(.secondary)
                        .font(.system(size: 20))
                }
                .transition(.opacity)
            }
        }
        .animation(.easeInOut(duration: 0.2), value:
viewModel.searchText.isEmpty)
    }
```

```
    private var editorsChoiceCarousel: some View {
        VStack(alignment: .leading, spacing: 8) {
            if !editorsChoiceRecipes.isEmpty {
                Text("Editor's Choice")
                    .fontWeight(.semibold)
                    .font(.system(size: 18))
            }

            TabView(selection: $carouselPage) {
                if editorsChoiceRecipes.isEmpty {
                    ForEach(0..<3, id: \.self) { index in
                        RoundedRectangle(cornerRadius: 12)
                            .fill(Color("grey-100"))
                            .frame(height: 152)
                            .overlay(
                                Image(systemName: "photo")
                                    .font(.largeTitle)
                                    .foregroundColor(.gray)
                            )
                            .tag(index)
                    }
                } else {
                    ForEach(Array(editorsChoiceRecipes.enumerated()), id:
\.element.id) { index, recipe in
                        NavigationLink {
                            RecipeDetailView(
                                item: recipe,
                                listViewModel: viewModel,
                                userId: userId,
                                isAdmin: mainViewModel.isAdmin
                            )
                        } label: {
                            CarouselCard(recipe: recipe)
                        }
                        .tag(index)
                    }
                }
            }
            .tabViewStyle(PageTabViewStyle(indexDisplayMode: .automatic))
            .frame(height: 152)
            .clipShape(RoundedRectangle(cornerRadius: 12))
        }
    }

    private var topCategories: some View {
        VStack(alignment: .leading) {
            Text("Top Categories")
                .fontWeight(.semibold)
                .font(.system(size: 20))

            ScrollView(.horizontal, showsIndicators: false) {
                HStack {
                    categoryChip(
                        title: "All",
                        isSelected: selectedCategory == nil
                    ) {
                        selectedCategory = nil
                        viewModel.categoryFilter = nil
                    }

                    ForEach(RecipeCategory.allCases) { category in
                        categoryChip(
                            title: category.rawValue,
```

```
                            isSelected: selectedCategory == category
                        ) {
                            if selectedCategory == category {
                                selectedCategory = nil
                                viewModel.categoryFilter = nil
                            } else {
                                selectedCategory = category
                                viewModel.categoryFilter = category
                            }
                        }
                    }
                }
            }
        }

        private func categoryChip(
            title: String,
            isSelected: Bool,
            action: @escaping () -> Void
        ) -> some View {
            Text(title)
                .padding(.horizontal, 16)
                .padding(.vertical, 8)
                .foregroundColor(isSelected ? .white : .black)
                .background(isSelected ? Color("brown-500") : Color.clear)
                .cornerRadius(20)
                .overlay(
                    RoundedRectangle(cornerRadius: 20)
                        .stroke(Color("grey-100"), lineWidth: 1.5)
                )
                .onTapGesture {
                    action()
                }
        }
    }

    private struct CarouselCard: View {
        let recipe: RecipeListItem

        var body: some View {
            ZStack(alignment: .bottomLeading) {
                Group {
                    if let urlString = recipe.imageURL,
                       let url = URL(string: urlString) {

                        AsyncImage(url: url) { phase in
                            switch phase {
                            case .success(let img):
                                img
                                    .resizable()
                                    .scaledToFill()

                            case .empty:
                                Color("grey-100")
                                    .overlay(ProgressView())

                            default:
                                Color("grey-100")
                                    .overlay(
                                        Image(systemName: "photo")
                                            .foregroundColor(.gray)
                                    )
```

```
                        }
                    }

                } else {
                    Color("grey-100")
                        .overlay(
                            Image(systemName: "photo")
                                .foregroundColor(.gray)
                        )
                }
            }
            .frame(maxWidth: .infinity, maxHeight: .infinity)
            .clipped()

            LinearGradient(
                colors: [.clear, .black.opacity(0.55)],
                startPoint: .center,
                endPoint: .bottom
            )

            Text(recipe.title)
                .font(.subheadline.bold())
                .foregroundColor(.white)
                .padding(10)
        }
        .frame(height: 152)
        .clipShape(RoundedRectangle(cornerRadius: 12))
    }
}
```

A.5 Програмний код автоматизованих тестів

Код тестів бізнес-логіки:

```
@testable import CookBook

// MARK: - Helpers

private func makeIngredient(id: String = UUID().uuidString,
                            title: String = "Sugar",
                            quantity: String) -> ProductListItem {
    ProductListItem(id: id, title: title, quantity: quantity, isDone: false)
}

private func makeRecipe(
    id: String = UUID().uuidString,
    title: String = "Test Recipe",
    category: RecipeCategory = .main,
    isFavorite: Bool = false,
    isVegan: Bool = false,
    isGlutenFree: Bool = false,
    isVegetarian: Bool = false,
    isDairyFree: Bool = false,
    isNutFree: Bool = false,
    isKidsFree: Bool = false,
    createdDate: TimeInterval = 1_000_000
) -> RecipeListItem {
```

```
    RecipeListItem(
        id: id,
        ownerId: "owner-1",
        title: title,
        ingredients: [],
        instructions: "Mix and cook.",
        categories: category,
        difficulty: .easy,
        imageURL: nil,
        createdDate: createdDate,
        isFavorite: isFavorite,
        isPublic: false,
        isGlutenFree: isGlutenFree,
        isVegan: isVegan,
        isVegetarian: isVegetarian,
        isDairyFree: isDairyFree,
        isNutFree: isNutFree,
        isKidsFree: isKidsFree
    )
}

// MARK: - Filter logic (mirrors RecipeListViewViewModel.displayedRecipes)
// Extracted as a pure function so tests run without touching Firebase.

private func applyFilters(
    to recipes: [RecipeListItem],
    searchText: String = "",
    favoritesOnly: Bool = false,
    categoryFilter: RecipeCategory? = nil,
    sortOption: SortOption = .dateCreatedDescending,
    filterVegan: Bool = false,
    filterGlutenFree: Bool = false,
    filterVegetarian: Bool = false,
    filterDairyFree: Bool = false,
    filterNutFree: Bool = false,
    filterKidsFree: Bool = false
) -> [RecipeListItem] {
    var list = recipes

    if favoritesOnly        { list = list.filter { $0.isFavorite }    }
    if !searchText.isEmpty  {
        let q = searchText.lowercased()
        list = list.filter {
            $0.title.lowercased().contains(q) ||
            $0.categories.rawValue.lowercased().contains(q)
        }
    }
    if let cat = categoryFilter { list = list.filter { $0.categories == cat } }
    if filterVegan       { list = list.filter { $0.isVegan }       }
    if filterGlutenFree  { list = list.filter { $0.isGlutenFree }  }
    if filterVegetarian  { list = list.filter { $0.isVegetarian }  }
    if filterDairyFree   { list = list.filter { $0.isDairyFree }   }
    if filterNutFree     { list = list.filter { $0.isNutFree }     }
    if filterKidsFree    { list = list.filter { $0.isKidsFree }    }

    switch sortOption {
    case .titleAscending:        list.sort { $0.title < $1.title }
    case .titleDescending:       list.sort { $0.title > $1.title }
    case .dateCreatedDescending: list.sort { $0.createdDate > $1.createdDate }
    case .dateCreatedAscending:  list.sort { $0.createdDate < $1.createdDate }
    }
    return list
}
```

```
// ─────────────────────────────────────────────────────────────────────────────
// MARK: - 1. ScalingServiceTests
// ─────────────────────────────────────────────────────────────────────────────

final class ScalingServiceTests: XCTestCase {

    // Scale 1 → 2: doubles integer quantities
    func test_scale_integerQuantity_doublesValue() {
        let ingredients = [makeIngredient(quantity: "200 g")]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "400 g")
    }

    // Scale 2 → 1: halves integer quantities
    func test_scale_integerQuantity_halvesValue() {
        let ingredients = [makeIngredient(quantity: "400 g")]
        let result = ScalingService.scale(ingredients: ingredients, from: 2, to:
1)
        XCTAssertEqual(result[0].quantity, "200 g")
    }

    // Scale with decimal quantity
    func test_scale_decimalQuantity() {
        let ingredients = [makeIngredient(quantity: "2.5 tbsp")]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "5 tbsp")
    }

    // Fraction input: "1/2 cup" × 2 = "1 cup"
    func test_scale_fractionQuantity_toWhole() {
        let ingredients = [makeIngredient(quantity: "1/2 cup")]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "1 cup")
    }

    // Mixed number: "1 1/2 cups" × 2 = "3 cups"
    func test_scale_mixedNumberQuantity() {
        let ingredients = [makeIngredient(quantity: "1 1/2 cups")]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
2)
        XCTAssertEqual(result[0].quantity, "3 cups")
    }

    // Non-numeric quantity (e.g. "to taste") must be returned unchanged
    func test_scale_nonNumericQuantity_returnsUnchanged() {
        let ingredients = [makeIngredient(quantity: "to taste")]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
4)
        XCTAssertEqual(result[0].quantity, "to taste")
    }

    // basePortion == targetPortion → no change
    func test_scale_samePortion_returnsOriginal() {
        let ingredients = [makeIngredient(quantity: "100 ml")]
        let result = ScalingService.scale(ingredients: ingredients, from: 3, to:
3)
        XCTAssertEqual(result[0].quantity, "100 ml")
    }
```

```
    // basePortion = 0 is invalid → returns original
    func test_scale_zeroBasePortion_returnsOriginal() {
        let ingredients = [makeIngredient(quantity: "100 ml")]
        let result = ScalingService.scale(ingredients: ingredients, from: 0, to:
2)
        XCTAssertEqual(result[0].quantity, "100 ml")
    }

    // Empty ingredient list → returns empty
    func test_scale_emptyIngredients_returnsEmpty() {
        let result = ScalingService.scale(ingredients: [], from: 1, to: 2)
        XCTAssertTrue(result.isEmpty)
    }

    // Scale quantity without unit: bare number
    func test_scaleQuantity_bareNumber() {
        let result = ScalingService.scaleQuantity("3", by: 2)
        XCTAssertEqual(result, "6")
    }

    // Trailing zeros stripped: 2.50 → "2.5"
    func test_scaleQuantity_stripsTrailingZeros() {
        let result = ScalingService.scaleQuantity("1 cup", by: 2.5)
        XCTAssertEqual(result, "2.5 cup")
    }

    // Integer result has no decimal point: 400.0 → "400"
    func test_scaleQuantity_integerResultHasNoDecimalPoint() {
        let result = ScalingService.scaleQuantity("200 g", by: 2)
        XCTAssertFalse(result.contains("."), "Integer result should not contain
a decimal point")
    }

    // Multiple ingredients are each scaled independently
    func test_scale_multipleIngredients_eachScaled() {
        let ingredients = [
            makeIngredient(id: "1", title: "Flour",  quantity: "100 g"),
            makeIngredient(id: "2", title: "Butter", quantity: "50 g")
        ]
        let result = ScalingService.scale(ingredients: ingredients, from: 1, to:
4)
        XCTAssertEqual(result[0].quantity, "400 g")
        XCTAssertEqual(result[1].quantity, "200 g")
    }
}


// ——————————————————————————————————————————————————————————————————————
// MARK: - 2. ModelInitializationTests
// ——————————————————————————————————————————————————————————————————————

final class ModelInitializationTests: XCTestCase {

    // MARK: User

    // Full User document decodes correctly
    func test_user_decodesAllFields() throws {
        let json = """
        {
            "id": "u1",
            "name": "Helen",
            "email": "helen@example.com",
```

```
            "joined": 1700000000.0,
            "role": "admin"
        }
        """.data(using: .utf8)!
        let user = try JSONDecoder().decode(User.self, from: json)
        XCTAssertEqual(user.id, "u1")
        XCTAssertEqual(user.name, "Helen")
        XCTAssertEqual(user.role, .admin)
    }

    // Missing "role" field falls back to .regular (backward compatibility)
    func test_user_missingRole_defaultsToRegular() throws {
        let json = """
        {
            "id": "u2",
            "name": "Guest",
            "email": "g@example.com",
            "joined": 1700000000.0
        }
        """.data(using: .utf8)!
        let user = try JSONDecoder().decode(User.self, from: json)
        XCTAssertEqual(user.role, .regular)
    }

    // Unknown role string falls back to .regular
    func test_user_unknownRoleString_defaultsToRegular() throws {
        let json = """
        {
            "id": "u3",
            "name": "Unknown",
            "email": "u@example.com",
            "joined": 1700000000.0,
            "role": "superuser"
        }
        """.data(using: .utf8)!
        let user = try JSONDecoder().decode(User.self, from: json)
        XCTAssertEqual(user.role, .regular)
    }

    // MARK: RecipeListItem


    // All RecipeStatus raw values decode correctly
    func test_recipeStatus_allRawValuesDecodable() {
        let cases: [String: RecipeStatus] = [
            "draft": .draft,
            "pending": .pending,
            "approved": .approved,
            "rejected": .rejected
        ]
        for (raw, expected) in cases {
            let json = "\"\(raw)\"".data(using: .utf8)!
            let decoded = try? JSONDecoder().decode(RecipeStatus.self, from:
json)
            XCTAssertEqual(decoded, expected, "Failed for raw value: \(raw)")
        }
    }

    // publicURL is non-nil only for public recipes
    func test_recipe_publicURL_nonNilWhenPublic() {
        var recipe = makeRecipe()
        recipe.isPublic = true
        XCTAssertNotNil(recipe.publicURL)
```

```
        XCTAssertTrue(recipe.publicURL?.absoluteString.contains(recipe.id) ==
true)
    }

    func test_recipe_publicURL_nilWhenNotPublic() {
        let recipe = makeRecipe()
        XCTAssertNil(recipe.publicURL)
    }

    // MARK: RecipeCollection


    // Explicit isPublic = true decodes correctly
    func test_collection_publicFlagDecodes() throws {
        let json = """
        {
            "id": "c2",
            "title": "Editor Picks",
            "ownerId": "admin",
            "isPublic": true,
            "recipeIds": ["r1","r2"]
        }
        """.data(using: .utf8)!
        let col = try JSONDecoder().decode(RecipeCollection.self, from: json)
        XCTAssertTrue(col.isPublic)
        XCTAssertEqual(col.recipeIds.count, 2)
    }

    // MARK: ProductListItem

    func test_productListItem_decodesAllFields() throws {
        let json = """
        {
            "id": "p1",
            "title": "Flour",
            "quantity": "200 g",
            "isDone": false,
            "recipeTitle": "Pancakes"
        }
        """.data(using: .utf8)!
        let item = try JSONDecoder().decode(ProductListItem.self, from: json)
        XCTAssertEqual(item.title, "Flour")
        XCTAssertEqual(item.quantity, "200 g")
        XCTAssertEqual(item.recipeTitle, "Pancakes")
    }
}


// ─────────────────────────────────────────────────────────────────────────
// MARK: - 3. RecipeFilterLogicTests
// Tests the displayedRecipes filtering & sorting logic in isolation (no
Firebase).
// ─────────────────────────────────────────────────────────────────────────

final class RecipeFilterLogicTests: XCTestCase {

    // Shared fixture
    private var recipes: [RecipeListItem] = []

    override func setUp() {
        super.setUp()
        recipes = [
```

```
            makeRecipe(id: "1", title: "Apple Pie",     category: .dessert,
isFavorite: true,  isVegan: false, createdDate: 3000),
            makeRecipe(id: "2", title: "Caesar Salad",  category: .salad,
isFavorite: false, isVegan: false, createdDate: 2000),
            makeRecipe(id: "3", title: "Vegan Burger",  category: .main,
isFavorite: true,  isVegan: true,  createdDate: 1000),
            makeRecipe(id: "4", title: "Oat Pancakes",  category: .breakfast,
isFavorite: false,
                       isGlutenFree: true, createdDate: 4000),
            makeRecipe(id: "5", title: "Tomato Soup",   category: .soup,
isFavorite: false,
                       isVegetarian: true, createdDate: 500)
        ]
    }

    // MARK: Search

    func test_search_byTitle_returnsMatchingRecipes() {
        let result = applyFilters(to: recipes, searchText: "salad")
        XCTAssertEqual(result.count, 1)
        XCTAssertEqual(result[0].id, "2")
    }

    func test_search_caseInsensitive() {
        let result = applyFilters(to: recipes, searchText: "APPLE")
        XCTAssertEqual(result.count, 1)
        XCTAssertEqual(result[0].title, "Apple Pie")
    }

    func test_search_byCategory_returnsMatch() {
        // "Soup" category raw value matches search text
        let result = applyFilters(to: recipes, searchText: "soup")
        XCTAssertTrue(result.contains { $0.id == "5" })
    }

    func test_search_noMatch_returnsEmpty() {
        let result = applyFilters(to: recipes, searchText: "xyz_not_found")
        XCTAssertTrue(result.isEmpty)
    }

    func test_search_emptyString_returnsAll() {
        let result = applyFilters(to: recipes, searchText: "")
        XCTAssertEqual(result.count, recipes.count)
    }

    // MARK: Favorites

    func test_favoritesOnly_returnsOnlyFavorites() {
        let result = applyFilters(to: recipes, favoritesOnly: true)
        XCTAssertTrue(result.allSatisfy { $0.isFavorite })
        XCTAssertEqual(result.count, 2)
    }

    func test_favoritesOnly_false_returnsAll() {
        let result = applyFilters(to: recipes, favoritesOnly: false)
        XCTAssertEqual(result.count, recipes.count)
    }

    // MARK: Category filter

    func test_categoryFilter_returnsOnlyMatchingCategory() {
        let result = applyFilters(to: recipes, categoryFilter: .dessert)
        XCTAssertEqual(result.count, 1)
```

```
        XCTAssertEqual(result[0].id, "1")
    }

    func test_categoryFilter_nil_returnsAll() {
        let result = applyFilters(to: recipes, categoryFilter: nil)
        XCTAssertEqual(result.count, recipes.count)
    }

    func test_categoryFilter_noMatch_returnsEmpty() {
        let result = applyFilters(to: recipes, categoryFilter: .pizza)
        XCTAssertTrue(result.isEmpty)
    }

    // MARK: Dietary filters

    func test_veganFilter_returnsOnlyVeganRecipes() {
        let result = applyFilters(to: recipes, filterVegan: true)
        XCTAssertTrue(result.allSatisfy { $0.isVegan })
        XCTAssertEqual(result.count, 1)
    }

    func test_glutenFreeFilter_returnsOnlyGlutenFree() {
        let result = applyFilters(to: recipes, filterGlutenFree: true)
        XCTAssertTrue(result.allSatisfy { $0.isGlutenFree })
    }

    func test_vegetarianFilter_returnsOnlyVegetarian() {
        let result = applyFilters(to: recipes, filterVegetarian: true)
        XCTAssertTrue(result.allSatisfy { $0.isVegetarian })
    }

    // Combined dietary filters act as AND
    func test_combinedDietaryFilters_actAsAND() {
        // No recipe is both vegan AND gluten-free in the fixture
        let result = applyFilters(to: recipes, filterVegan: true,
filterGlutenFree: true)
        XCTAssertTrue(result.isEmpty)
    }

    // MARK: Sorting

    func test_sort_titleAscending_ordersAlphabetically() {
        let result = applyFilters(to: recipes, sortOption: .titleAscending)
        let titles = result.map { $0.title }
        XCTAssertEqual(titles, titles.sorted())
    }

    func test_sort_titleDescending_ordersReverseAlphabetically() {
        let result = applyFilters(to: recipes, sortOption: .titleDescending)
        let titles = result.map { $0.title }
        XCTAssertEqual(titles, titles.sorted(by: >))
    }

    func test_sort_dateDescending_newestFirst() {
        let result = applyFilters(to: recipes, sortOption:
.dateCreatedDescending)
        let dates = result.map { $0.createdDate }
        XCTAssertEqual(dates, dates.sorted(by: >))
    }

    func test_sort_dateAscending_oldestFirst() {
        let result = applyFilters(to: recipes, sortOption:
.dateCreatedAscending)
```

```
        let dates = result.map { $0.createdDate }
        XCTAssertEqual(dates, dates.sorted())
    }

    // MARK: Combined

    func test_combined_searchAndCategory_appliesBoth() {
        // "Salad" search + category .salad should find only Caesar Salad
        let result = applyFilters(to: recipes, searchText: "caesar",
categoryFilter: .salad)
        XCTAssertEqual(result.count, 1)
        XCTAssertEqual(result[0].id, "2")
    }

    func test_combined_favoritesAndVegan_returnsIntersection() {
        // Only "Vegan Burger" is both favourite and vegan
        let result = applyFilters(to: recipes, favoritesOnly: true, filterVegan:
true)
        XCTAssertEqual(result.count, 1)
        XCTAssertEqual(result[0].id, "3")
    }
}
```

ДОДАТОК Б

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

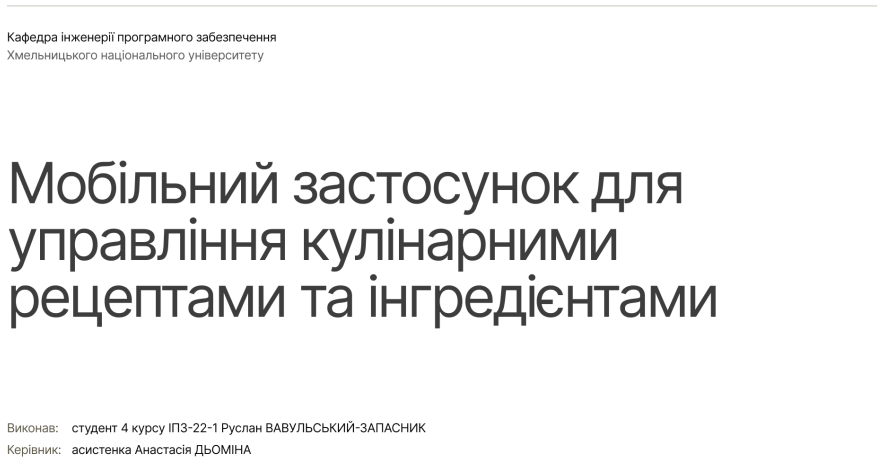


Рисунок Б.1 – Слайд 1

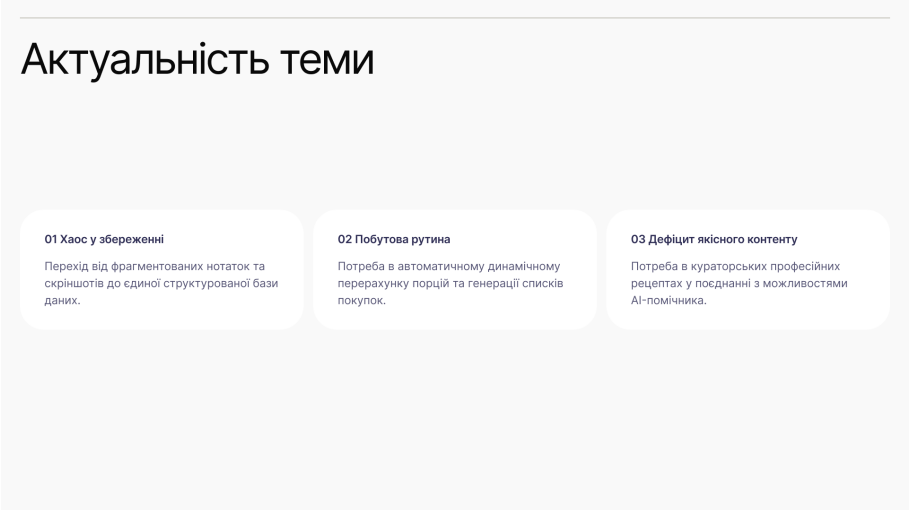


Рисунок Б.2 – Слайд 2

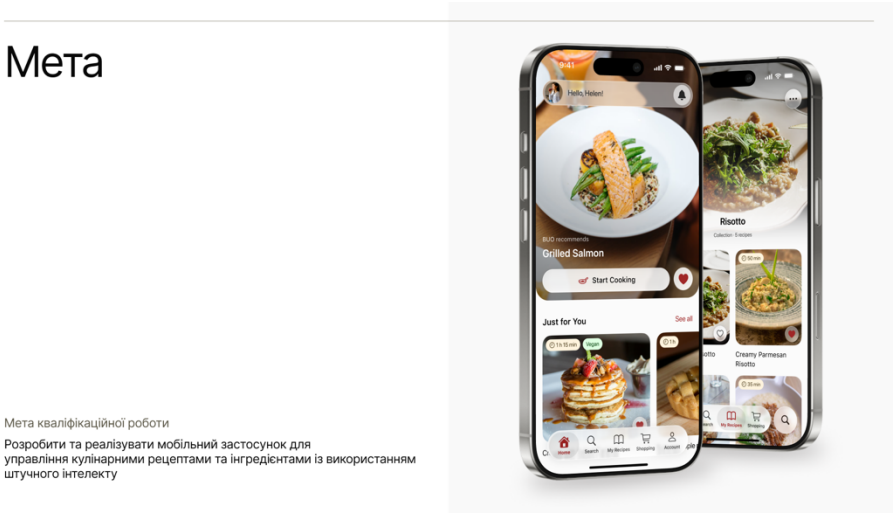


Рисунок Б.3 – Слайд 3

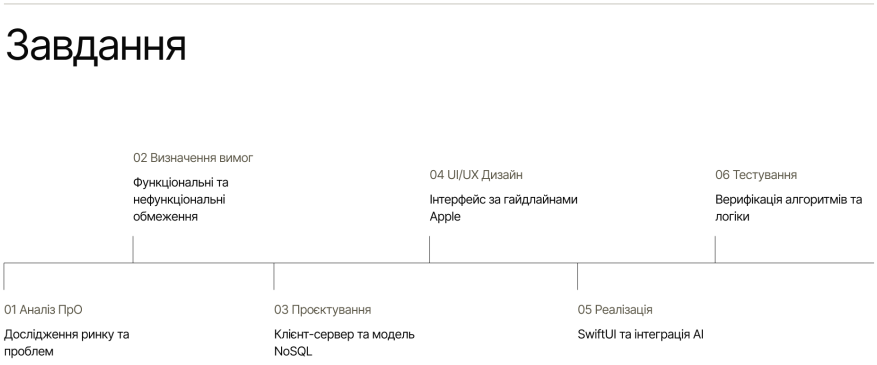


Рисунок Б.4 – Слайд 4

Рисунок Б.5 – Слайд 5

Змістовий аналіз предметної області

Рисунок Б.6 – Слайд 6

Аналіз наявного ПЗ

	ПРОФЕСІЙНИЙ КОНТЕНТ	ПРИВАТНІ ЗАПИСИ	АВТОПЕРЕРАХУНОК ПОРЦІЙ	ШІ-ГЕНЕРАЦІЯ РЕЦЕПТІВ
KITCHEN STORIES	✅	❌	✅	❌
COOKPAD	❌	✅	❌	❌
CULINARA	✅	❌	🟠	❌
РОЗРОБКА	✅	✅	✅	✅

Рисунок Б.7 – Слайд 7

Функціональні та нефункціональні вимоги

ФУНКЦІОНАЛЬНІ ВИМОГИ	НЕФУНКЦІОНАЛЬНІ ВИМОГИ
Динамічний перерахунок і списки покупок	Цільова платформа та дизайн
Пошук та фільтрація	Хмарна синхронізація в реальному часі
Модерація та адміністрування	Відмовостійкість
Генерація контенту штучним інтелектом	Оптимізація пам'яті та швидкодія

Рисунок Б.8 – Слайд 8

Архітектура та шаблони

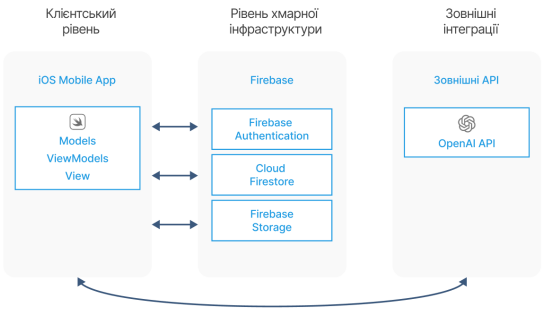


Рисунок Б.9 – Слайд 9

Декомпозиція, залежності

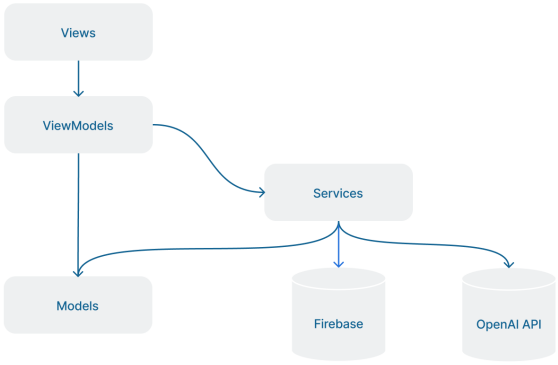


Рисунок Б.10 – Слайд 10

Проєктування модулів і даних

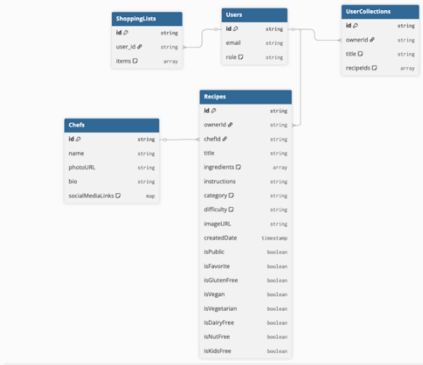


Рисунок Б.11 – Слайд 11

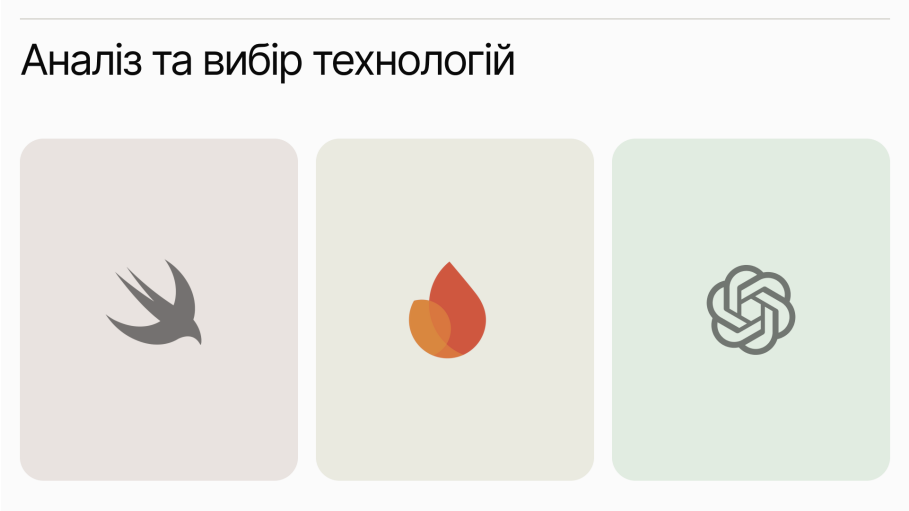


Рисунок Б.12 – Слайд 12

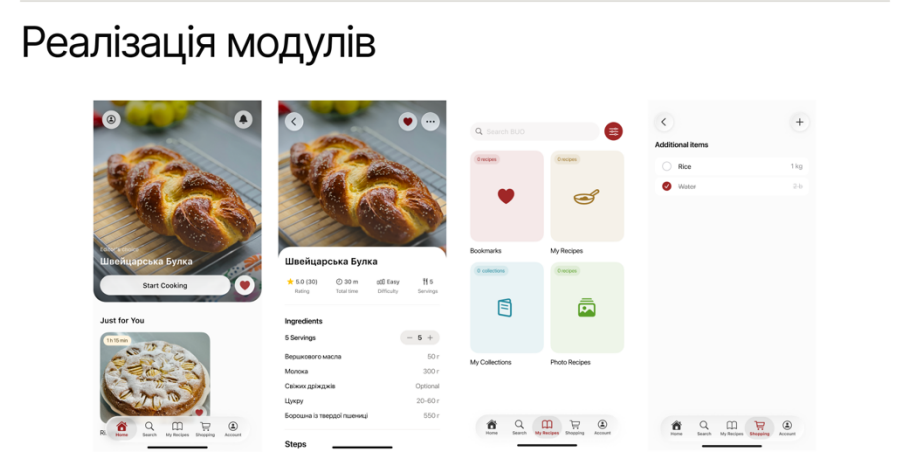


Рисунок Б.13 – Слайд 13

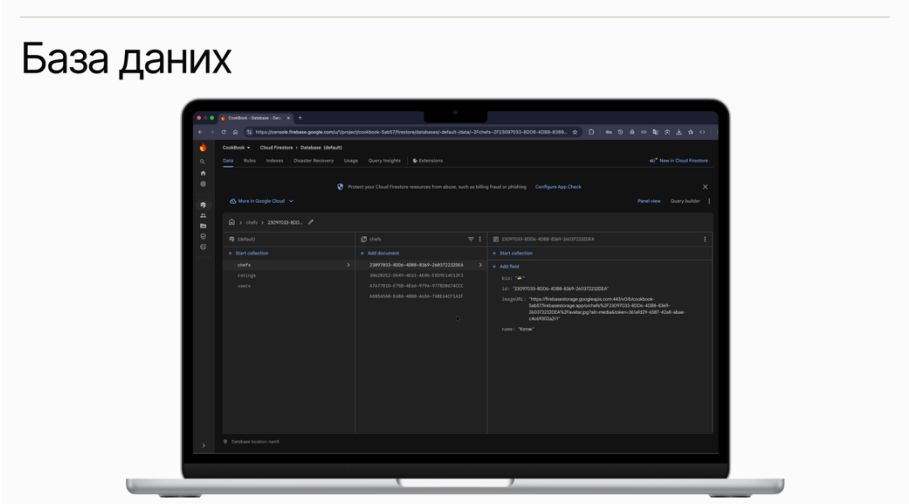


Рисунок Б.14 – Слайд 14

Тестування ПЗ

Сценарії перевірки процесів ідентифікації та доступу

Ідентифікатор	Екран / Контекст	Вхідні дані / Дія	Реакція програми	Результат
TC-AUTH-01	Екран входу	Введення валідної пошти та коректного пароля, натискання кнопки підтвердження	Система надсилає запит, закриває форму та відкриває головну панель навігації	Успішно
TC-AUTH-02	Екран входу	Залишення текстового поля пароля порожнім, натискання кнопки входу	Перехід блокується, під полем введення з'являється попередження про незаповнені дані	Успішно
TC-AUTH-03	Екран входу	Введення пошти без символу «@» та пароля, натискання кнопки підтвердження	Програма залишається на екрані входу, виводиться повідомлення про неправильний формат пошти	Успішно
TC-AUTH-04	Екран входу	Натискання кнопки переходу в гостьовий режим перегляду	Форма авторизації зникає, завантажується стрічка страв із прихованими кнопками додавання	Успішно
TC-AUTH-05	Профіль	Натискання кнопки виходу з облікового запису в налаштуваннях	Сесія завершується, очищаються локальні дані, відкривається стартовий екран ідентифікації	Успішно

Сценарії перевірки управління кулінарним контентом

Ідентифікатор	Екран / Контекст	Вхідні дані / Дія	Реакція програми	Результат
TC-REC-01	Створення страви	Заповнення назви, вибір категорії, додавання кроку приготування, збереження	Запис успішно додається до приватної колекції, форма закривається, оновлюється загальний список	Успішно
TC-REC-02	Створення страви	Спроба збереження форми без введення назви рецепта	Кнопка збереження ігнорує запит або система підсвічує пусте поле червоним контуром	Успішно
TC-REC-03	Картка страви	Натискання на іконку збереження в обране у верхній панелі	Іконка змінює заливку, ідентифікатор страви записується до масиву закладок профілю	Успішно
TC-REC-04	Стрічка страв	Введення існуючого слова в текстовий рядок пошуку	Список миттєво скорочується, залишаються лише картки з відповідним текстом у назві	Успішно
TC-REC-05	Стрічка страв	Активація перемикача відображення виключно веганських позицій	Інтерфейс перемальовується, зникають страви, що містять продукти тваринного походження	Успішно

Рисунок Б.15 – Слайд 15

Висновки

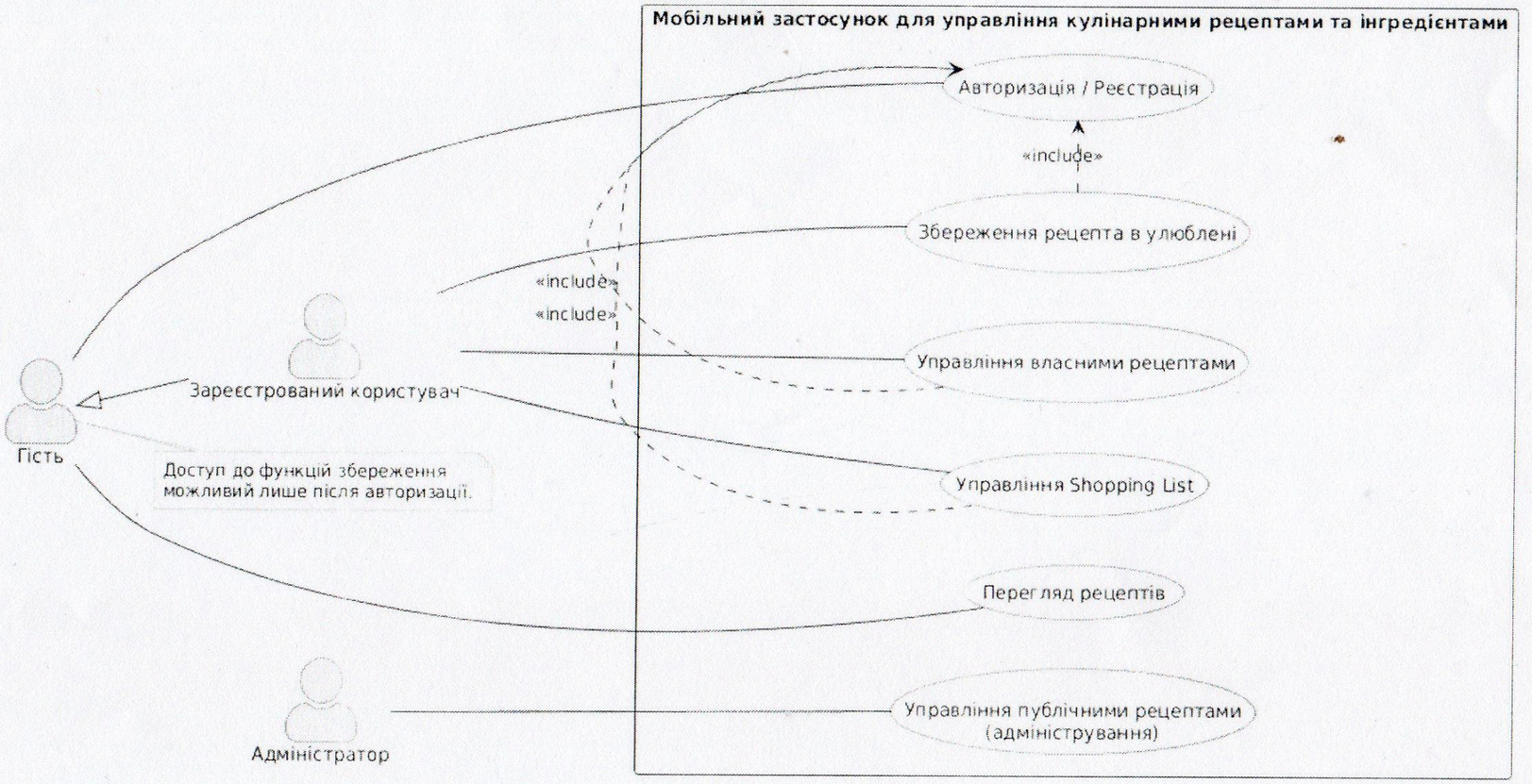
Пункти завдань	Що виконано по кожному завданню
1. Аналіз наявних рішень та предметної області	Проаналізовано 3 аналоги (Culinara, Kitchen Stories, Cookpad), створено контекстну діаграму IDEF0
2. Визначення вимог та рівнів доступу	Сформовано специфікації вимог, розмежовано систему на 3 ролі (Гість, Користувач, Адміністратор)
3. Проєктування архітектури та БД	Обрано патерн MVVM та клієнт-серверну архітектуру, спроєктовано NoSQL БД (Firestore) на 5 колекцій
4. Проєктування інтерфейсу користувача	Створено прототипи у Figma за стандартами Apple HIG, побудовано карту навігаційних переходів
5. Програмна реалізація та алгоритми	Написано код на Swift/SwiftUI, інтегровано Firebase та OpenAI, реалізовано автоматичний перерахунок інгредієнтів
6. Тестування програмного забезпечення	Проведено 48 тестів (Unit, UI, ручні сценарії). Дефектів не виявлено

Рисунок Б.16 – Слайд 16

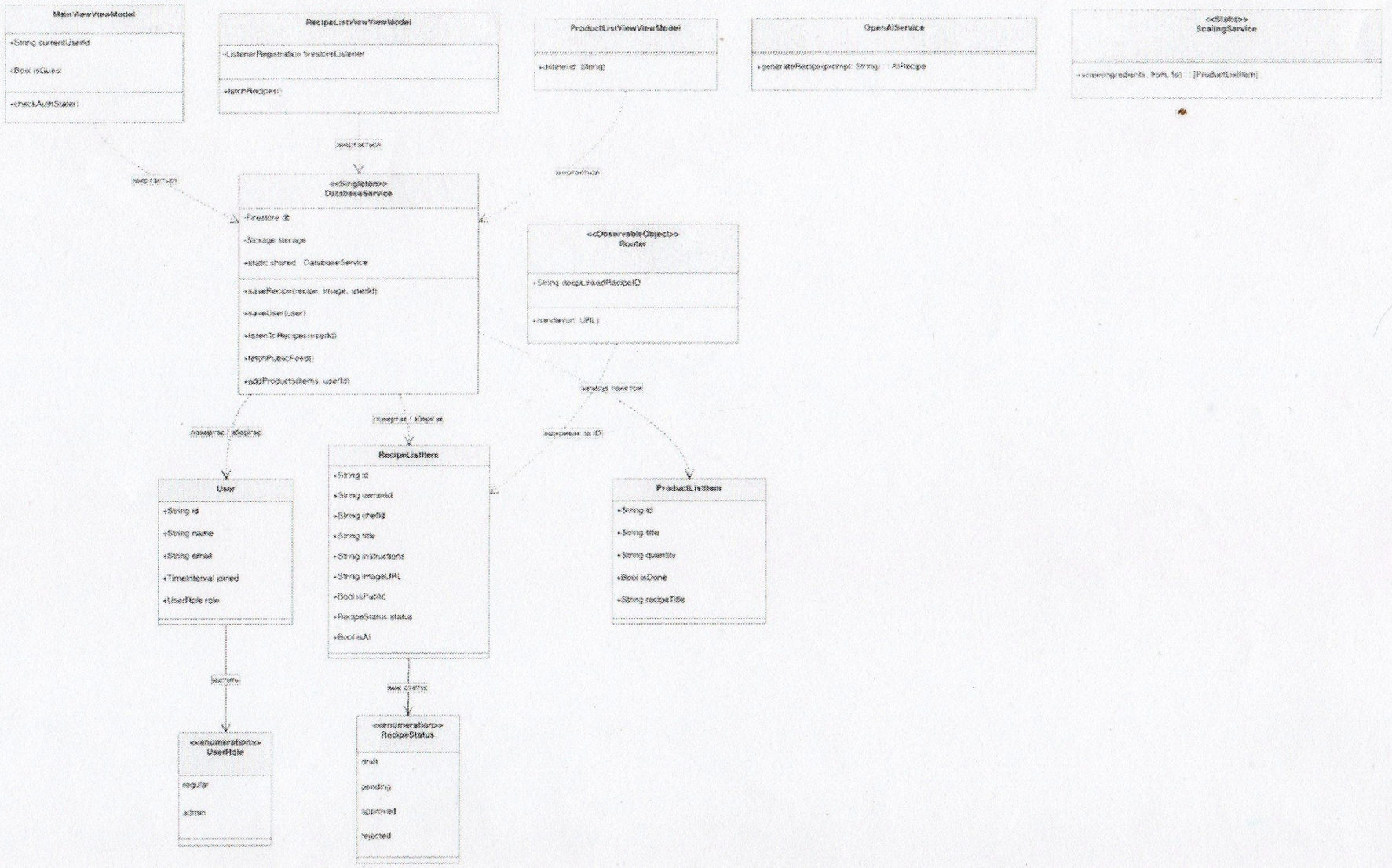
Дякую за увагу!

Рисунок Б.17 – Слайд 17

ГРАФІЧНІ МАТЕРІАЛИ



					КвРІПЗ.220197.01.03.Е8		
Змн.	Арк	№ докум.	Підпис	Дата			
Розробив		Вавульський-Запасник Р. А.		1.6	Діаграма варіантів використання	Літ.	Маса / Масштаб
Керівник		Дьоміна А. І.		1.6.2025			
						Аркуш 1	Аркушів 2
Н. Контр.		Форкун Ю. В.		1.6.2025		ХНУ,ІПЗ-22-1	
Затверд.		Бедратюк Л. П.					



					КвРІПЗ.220197.01.03.Е8			
Змн.	Арк	№ докум.	Підпис	Дата				
Розробив		Вавульський-Запасник Р. А.			Діаграма «сутність-зв'язок»	Літ.	Маса	Масштаб
Керівник		Дьоміна А. І.						
						Аркуш 2	Аркушів 2	
Н. Контр.		Форкун Ю. В.				ХНУ,ІПЗ-22-1		
Затверд.		Бедратюк Л. П.						

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Вавульський-Запасник Руслан Артемович
факультет ІТ, IVкурс, група ІПЗ-22-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

29.05.26
дата

підпис

Anti-Plagiarism (http://ap.km.ua) v-16.718

Максимальне співпадіння з одним документом 11.0%

Словники перевірки: UA, US, RU. **Помилок в документах: 27%**

ID: 272701 Назва: Мобільний застосунок для управління кулінарними рецептами та інгредієнтами Додано в БД: 2026-05-28 Автора: Руслан ВАВУЛЬСЬКИЙ-ЗАПАСНИК Керівники: Анастасія ДЬОМІНА Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	170441	1095	23093 (14%)	241 (22%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
269647	Назва: Переддипломна практика Додано в БД: 2026-03-02 Автора: Руслан ВАВУЛЬСЬКИЙ-ЗАПАСНИК Керівники: Дьоміна А. І., асистентка Консультанти: Опоненти:	18518 (11.0%)	182 (17.0%)

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Руслан ВАВУЛЬСЬКИЙ-ЗАПАСНИК

Співавтор:

Назва: Мобільний застосунок для управління кулінарними рецептами та інгредієнтами

Науковий керівник: Анастасія ДЬОМІНА

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:3.32%

Коефіцієнт подібності 2:1.26%

Мікропробіли: 0

Заміна букв: 0

Інтервали: 0

Білі знаки: 98

Дата створення звіту: 2026-05-28 22:34:33.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 25.05.26

експерт

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «Бакалавр»

Дипломник Вавульський-Запасник Руслан Артемович

Тема Програмне забезпечення для пошуку роботи та підбору персоналу з реалізацією інтерфейсу у вигляді Telegram-бота

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг кваліфікаційної роботи:

Кількість листів креслень 2; кількість сторінок записки 84

1.Короткий зміст пояснювальної записки та прийнятих рішень У кваліфікаційній роботі досліджено і проаналізовано предметну область, визначено усі функціональні та нефункціональні вимоги. Був проведений аналіз існуючих програм на ринку, розглянуто їх переваги і недоліки, та доведено актуальність розробки нового програмного забезпечення. Розглянуто інструменти для реалізації спроектованих рішень, в результаті чого створено програмне забезпечення. Також було проведено тестування програми, за результатами якого доведено, що розроблене програмне забезпечення працює коректно та готове до експуатації.

2. Висновок про відповідність роботи поставленому завданню Кваліфікаційна робота виконана відповідно до поставленого завдання та з дотриманням всіх вимог.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі доведено актуальність теми, визначено мету та завдання дипломного проектування. У першому розділі проведено аналіз предметної області, розглянуто існуючі рішення та визначені функціональні і нефункціональні вимоги до розроблюваного програмного забезпечення. У другому розділі проведено аналіз сучасних архітектур, розглянуто їх переваги і недоліки та визначено, що система буде відповідати монолітній архітектурі та моделі клієнт-сервер. У третьому розділі підготовлено всі залежності для написання коду та виконано практичну розробку програмних модулів і описано їх особливості, в результаті чого створено програмний продукт. Також у цьому розділі виконано модульне тестування системи та проведено його у відповідності до функціональних вимог, в результаті було підтверджено коректну роботу програми.

4. Позитивні сторони роботи Тематика кваліфікаційної роботи є актуальною, оскільки на сьогодні існуючі кулінарні мобільні застосунки не поєднують у собі доступ до професійного редакторського контенту, ведення повністю приватних записів, автоматичний перерахунок порцій та генерацію рецептів за допомогою штучного інтелекту. Також було застосовано новітні технології для побудови програмного продукту та актуальні архітектурні рішення.

5. Негативні сторони роботи У роботі реалізовано динамічний перерахунок інгредієнтів, проте було б доцільно додати функціонал підрахунку калорійності та балансу нутрієнтів страв. Також у майбутньому варто розширити модуль штучного інтелекту можливістю розпізнавання інгредієнтів безпосередньо з фотографій.

6. Оцінка графічного оформлення та пояснювальної записки Графічне оформлення виконано відповідно до теми кваліфікаційної роботи та подано у вигляді діаграм і рисунків. Пояснювальна записка оформлена згідно вимог чинних стандартів.

7. Відгук про кваліфікаційну роботу в цілому Кваліфікаційна робота заслуговує позитивної оцінки. Матеріал пояснювальної записки структурований, послідовний, чіткий та простий, що дозволяє чітко зрозуміти викладений матеріал у рамках тематики проектування. Графічний матеріал дає можливість наочно побачити деталі проектування системи.

8. Інші зауваження

9. Оцінка кваліфікаційної роботи Кваліфікаційна робота виконана у повному обсязі, відповідає поставленій задачі та заслуговує на оцінку «відмінно».

РЕЦЕНЗЕНТ Нічепорук Андрій Олександрович, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та інформаційних систем

"01" червня 2026 р. _____
(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів.
Назва кваліфікаційної роботи: «Мобільний застосунок для управління кулінарними рецептами та інгредієнтами»
Автор: Вавульський-Запасник Руслан Артемович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Дьоміна Анастасія Іванівна, асистенка
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Виявлені запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Виявлені запозичення частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:
Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:
1) у тексті кваліфікаційної роботи системою перевірки на плагіат Anti-Plagiarism виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках, у структурі змісту, назвах розділів/підрозділів, рамках форм, у назвах та URL-адресах публікацій переліку джерел посилання;
2) запозичення, виявлені у тексті роботи, є фрагментарними.
Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 11.0% з одного джерела (звіт з переддипломної практики). Загальна сумарна подібність у базі даних складає 14% за символами та 22% за лексемами. Крім того, за результатами додаткового аналізу коефіцієнт подібності 1 становить 3.32%, коефіцієнт подібності 2 – 1.26%. Не виявлено мікропробілів, замін букв або маніпуляцій з інтервалами. Виявлені білі знаки (98) є технічними символами форматування тексту і не є маніпуляцією. З урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 1.06.2026

Завідувач кафедри _____ Леонід БЕДРАТЮК

Гарант освітньої програми _____ Леонід БЕДРАТЮК

Керівник кваліфікаційної роботи _____ Анастасія ДЬОМІНА