Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА СОЦІАЛЬНОЇ МЕРЕЖІ ДЛЯ ОБМІНУ РЕЦЕПТАМИ З ВИКОРИСТАННЯМ FLUTTER, FAST API ТА POSTGRESQL

DEVELOPMENT OF A SOCIAL NETWORK FOR RECIPE SHARING USING FLUTTER, FAST API AND POSTGRESQL

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗс-21
Сивило Роман Андрійович

Керівник:
Паливода Данило Ігорович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ

Завідувач кафедри

«20» 12 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Сивилу Роману Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка соціальної мережі для обміну рецептами з використанням Flutter, FastAPI та PostgreSQL»*
Керівник роботи: *асистент Паливода Д. І.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р. № 474/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *сайти для публікації рецептів, порівняльний аналіз кросплатформних фреймворків для розробки додатків, порівняльний аналіз баз даних, Flutter, FastAPI, PostgreSQL, розробка кросплатформних додатків*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз сучасного стану розробки застосунків для публікації рецептів та існуючих рішень, формулювання вимог і базового функціоналу додатку, проектування взаємодії користувача з додатком, порівняльний аналіз і вибір необхідних інструментів для розробки, розробка серверної частини застосунку, завантаження серверної частини на хостинг, розробка клієнтської частини застосунку, опрацювання результатів розробки.*
5. Перелік графічного матеріалу: *16 рисунків, 6 таблиць, 2 додатки.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Паливода. Д. І.*		
Специфікація вимог до розробленої системи	*Паливода. Д. І.*		
Розробка об'єкта проектування	*Паливода. Д. І.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	6,86 %		
Академічна доброчесність	*Паливода. Д. І.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел з теми розробки соціальної мережі	до 04.02.2025 р.	вик
2	Аналіз проблематики створення соціальної мережі для обміну рецептами	до 01.03.2025 р.	вик.
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	вик.
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 29.03.2025 р.	вик.
5	Практична реалізація об'єкта проектування та розробка бази даних	до 26.04.2025 р.	вик.
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	вик.
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	вик

Здобувач вищої освіти _____ Роман СИВИЛО

Керівник кваліфікаційної роботи _____ Данило ПАЛИВОДА

АНОТАЦІЯ

Сивило Р. А. Розробка соціальної мережі для обміну рецептами з використанням Flutter, Fast API та PostgreSQL. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків і пропозицій, списку використаних джерел та додатків.

У першому розділі здійснено аналіз предметної області, пов’язаної зі створенням спеціалізованої соціальної мережі для обміну кулінарними рецептами. Досліджено сучасні аналоги, їх переваги й недоліки, актуальні потреби користувачів та потенціал розвитку таких платформ. Визначено основні функціональні вимоги до системи, сформульовано мету й завдання кваліфікаційної роботи, а також обґрунтовано актуальність проєкту.

У другому розділі виконано специфікацію вимог до розроблюваної системи. Детально проаналізовано як функціональні, так і нефункціональні вимоги до мобільного застосунку, обґрунтовано вибір архітектури системи. Проведено порівняльний аналіз сучасних технологій для клієнтської частини, серверної частини і бази даних.

У третьому розділі описано практичну реалізацію програмного забезпечення. Розроблено серверну частину API з реалізацією автентифікації, авторизації, обробки рецептів, коментарів та збережених добірок. Здійснено деплой сервісу на платформі Heroku з підключенням хмарної бази даних. Реалізовано клієнтську частину мобільного застосунку з підтримкою основних функцій: реєстрація, публікація, перегляд, оцінювання та коментування рецептів. Проведено тестування застосунку, оцінено його відповідність вимогам та описано напрями подальшого розвитку.

Ключові слова: соціальна мережа, Flutter, FastAPI, PostgreSQL, обмін рецептами, мобільний застосунок, REST API, база даних, архітектура системи.

ABSTRACT

Syvylo R. Development of a social network for recipe sharing using Flutter, Fast API and PostgreSQL. Manuscript. Bachelor's Qualification Thesis in the Educational Program "Software Engineering", Specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions and recommendations, a list of references, and appendices.

In the first chapter, a comprehensive analysis of the subject area related to the development of a specialized social network for recipe sharing was conducted. Existing solutions, their strengths and weaknesses, current user needs, and the potential for such platforms were examined. The main functional requirements for the system were identified, the purpose and objectives of the qualification project were formulated, and the project's relevance was substantiated.

In the second chapter, a detailed specification of the system requirements was presented. Both functional and non-functional requirements for the mobile application were analyzed, and the architecture of the system was justified. A comparative analysis of modern technologies for the client-side, server-side, and database was carried out.

In the third chapter, the practical implementation of the software was described. The server-side API was developed, including user authentication, authorization, recipe processing, commenting, and saved collections. The service was deployed on the Heroku platform with a connected cloud database. The client-side of the mobile application was implemented with support for key functions: user registration, recipe posting, viewing, rating, and commenting. The application was tested for compliance with the specified requirements, and directions for further development were outlined.

Keywords: social network, Flutter, FastAPI, PostgreSQL, recipe sharing, mobile application, REST API, database, cross-platform development, system architecture, software testing.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1 АНАЛІЗ ПЛАТФОРМ ДЛЯ ОБМІНУ РЕЦЕПТАМИ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ 9
1.1 Аналіз сучасного стану проблеми 9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра 11
Висновки до розділу 1 13
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ..... 14
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення 14
2.2 Вибір інструментів вирішення поставленого завдання..... 15
Висновки до розділу 2 22
РОЗДІЛ 3 РОЗРОБКА СОЦІАЛЬНОЇ МЕРЕЖІ ДЛЯ ОБМІНУ РЕЦЕПТАМИ. 23
3.1 Практична реалізація соціальної мережі для обміну рецептами..... 23
3.2 Тестування та налагодження інформаційно-комп’ютерної системи 45
3.3 Публікація API на хостинг 48
Висновки до розділу 3 52
ВИСНОВКИ..... 54
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 56
ДОДАТКИ 58

ВСТУП

Актуальність теми. У наш час способи взаємодії людей у цифровому середовищі зазнають суттєвих змін. Хоча традиційні соціальні мережі залишаються популярним інструментом комунікації, дедалі більшою стає потреба у спеціалізованих платформах, що відповідають конкретним інтересам і вподобанням користувачів. Особливої популярності набувають мережі, орієнтовані на обмін кулінарними рецептами, які дозволяють не лише публікувати рецепти, а й активно взаємодіяти через коментарі, оцінки, особисті рекомендації та тематичні добірки.

Соціальна мережа для обміну рецептами здатна вирішити практичні завдання сучасних користувачів – від організації та зберігання особистих рецептів до зручного доступу до кулінарних ідей у будь-який час. Використання таких платформ сприяє створенню тематичних спільнот, розвитку кулінарної культури та полегшує повсякденне життя завдяки швидкому пошуку рецептів з урахуванням особистих уподобань.

У цій роботі поставлено за мету розробити мобільний застосунок у форматі соціальної мережі для обміну рецептами з використанням технологій Flutter, FastAPI та PostgreSQL.

Об’єкт кваліфікаційної роботи – процес розробки соціальної мережі для обміну рецептами.

Предмет кваліфікаційної роботи – технології кросплатформної розробки, архітектура REST API та особливості роботи з базою даних у процесі створення соціальної мережі для обміну рецептами.

В ході роботи над кваліфікаційною роботою бакалавра, були поставлені наступні завдання:

– здійснити аналіз стану розробки соціальних мереж для обміну рецептами, визначити тенденції та недоліки існуючих платформ;

– сформулювати функціональні й нефункціональні вимоги до застосунку;

– виконати порівняльний аналіз технологій та обґрунтувати вибір Flutter, FastAPI і PostgreSQL;

– розробити архітектуру додатку, включаючи проектування бази даних і структуру REST API;

– реалізувати серверну частину із авторизацією, автентифікацією, управлінням рецептами, коментарями та збереженими добірками;

– розмістити розроблене API на хмарній платформі Heroku;

– створити клієнтську частину з можливістю реєстрації, публікації, коментування рецептів і керування персональними добірками.

РОЗДІЛ 1
АНАЛІЗ ПЛАТФОРМ ДЛЯ ОБМІНУ РЕЦЕПТАМИ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Упродовж останніх років значне зростання популярності соціальних мереж докорінно змінило спосіб взаємодії користувачів у цифровому середовищі. Соціальні мережі поступово трансформувалися від простих платформ для загального спілкування до спеціалізованих ресурсів, які відповідають конкретним інтересам та потребам користувачів. Однією з найпопулярніших тем серед таких платформ є кулінарія, оскільки інтерес до обміну рецептами, кулінарними порадами та власним досвідом приготування страв постійно зростає.

Cookpad – це міжнародна кулінарна платформа, яка спеціалізується на обміні рецептами та об'єднує мільйони активних користувачів по всьому світу. Платформа пропонує великий обсяг кулінарного контенту, охоплюючи різноманітні кухні та кулінарні традиції (рис. 1.1). Однак аналіз функціональних можливостей та інтерфейсу Cookpad дозволяє виявити низку недоліків, які можна покращити для забезпечення ще більшого комфорту та зручності користувачів.

Рисунок 1.1 – Головна сторінка Cookpad.com [1]

Хоча Cookpad [2] має досить зрозумілий дизайн, багато користувачів відзначають перевантаженість інтерфейсу, що ускладнює швидкий пошук рецептів. Платформа не завжди забезпечує ефективні фільтри та структуру, через що користувачам доводиться витрачати зайвий час на пошук релевантного контенту.

Персоналізація контенту на Cookpad реалізована на досить базовому рівні. Користувачі не отримують рекомендацій, які повністю відповідають їхнім смакам, дієтичним потребам або індивідуальним уподобанням. Покращення алгоритмів рекомендацій могло б значно підвищити задоволеність користувачів.

Також платформа має обмежені інтерактивні можливості, зокрема комунікація користувачів здебільшого обмежується коментарями під рецептами. Відсутність глибокої взаємодії та тематичних спільнот не дозволяє повноцінно розкрити соціальний потенціал платформи.

Cookpad також потребує вдосконалення системи оцінювання та перевірки рецептів, адже наявна система є досить простою і не завжди гарантує якість контенту. Розширення функцій модерації могло б значно підвищити надійність платформи.

Додатково, актуальною є проблема відсутності зручної вкладки для збережених рецептів, яка дозволила б користувачам легко формувати та керувати персональними добірками улюблених страв. Інтеграція такої функції стала б суттєвим покращенням користувацького досвіду.

Серед додаткових покращень, необхідних для підвищення комфорту використання платформи, є введення темної теми інтерфейсу, яка стає все більш затребуваною серед сучасних користувачів, знижуючи навантаження на очі під час тривалого перегляду контенту.

Крім того, для забезпечення стабільного розвитку та монетизації платформи можна розглянути ідею впровадження преміум-підписки, яка б відкривала користувачам доступ до додаткових ексклюзивних функцій, таких як поглиблені персоналізовані рекомендації, доступ до спеціальних рецептів та відсутність реклами.

Отже, хоча Cookpad є однією з провідних кулінарних платформ, існує значний потенціал для покращення, зокрема в плані зручності інтерфейсу, персоналізації, інтерактивності, а також додаткових функцій, таких як темна тема, вкладка збережених рецептів та преміум-підписка.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Сформовано мету роботи, яка полягає в розробці програмного забезпечення у вигляді соціальної мережі для обміну кулінарними рецептами. Функціональність цієї мережі передбачає можливість створення, перегляду, коментування, оцінювання рецептів та формування персональних колекцій, задовольняючи потребу користувачів у зручній та інтерактивній платформі для кулінарної взаємодії.

Відповідно до мети визначено завдання, які виконуватимуться під час роботи над кваліфікаційним проєктом:

– здійснити аналіз стану розробки соціальних мереж для обміну рецептами, визначити тенденції та недоліки існуючих платформ;

– сформулювати функціональні й нефункціональні вимоги до застосунку;

– виконати порівняльний аналіз технологій та обґрунтувати вибір Flutter, FastAPI і PostgreSQL;

– розробити архітектуру додатку, включаючи проектування бази даних і структуру REST API;

– реалізувати серверну частину із авторизацією, автентифікацією, управлінням рецептами, коментарями та збереженими добірками;

– розмістити розроблене API на хмарній платформі Heroku;

– створити клієнтську частину з можливістю реєстрації, публікації, коментування рецептів і керування персональними добірками.

В ході дослідження проведено аналіз сучасного стану проблематики соціальних платформ для обміну рецептами, визначено актуальність створення таких спеціалізованих мереж та охарактеризовано предметну область їх застосування, окреслено потенційні напрями подальшого розвитку.

На етапі аналізу здійснено огляд існуючих рішень, проаналізовано їхні сильні та слабкі сторони, визначено сучасні тенденції та потреби користувачів. В результаті такого аналізу було обрано найбільш оптимальні рішення й функціональні можливості, які мають бути реалізовані в соціальній мережі для обміну рецептами.

Вибір технологій та інструментів є результатом попереднього етапу аналізу, оскільки саме від цих рішень залежить подальший процес створення додатку, його зручність використання, масштабованість та підтримка великої кількості користувачів.

Формування вимог і визначення основного функціоналу додатку створює фундамент для практичного етапу розробки. На цьому етапі формуються чіткі технічні та функціональні вимоги, які представлені у вигляді текстових описів, UML-діаграм, сценаріїв використання, таблиць та інших способів моделювання.

Наступним важливим завданням є безпосередньо розробка програмного забезпечення соціальної мережі. На цьому етапі реалізуються програмні компоненти, розробляються алгоритми, здійснюється створення користувацького інтерфейсу, а також встановлюється взаємодія між клієнтською та серверною частинами платформи.

Заключним етапом роботи є тестування розробленої соціальної мережі. На даному етапі проводиться перевірка як базового функціоналу, так і загальної продуктивності системи в різних сценаріях використання. Тестування дозволяє виявити та усунути можливі помилки, оцінити зручність інтерфейсу, швидкість роботи додатку, коректність роботи функцій оцінювання та коментування, а також перевірити відповідність програмного продукту поставленим вимогам та очікуванням кінцевих користувачів.

Висновки до розділу 1

Було здійснено аналіз поточного стану розробки спеціалізованих соціальних мереж для обміну кулінарними рецептами, визначено ключові тенденції та проблеми існуючих рішень, а також окреслено перспективні напрями подальшого вдосконалення таких платформ.

Соціальні мережі кулінарної тематики є багатообіцяючим напрямом, оскільки вони створюють природне цифрове середовище для взаємодії користувачів з близькими кулінарними інтересами. В процесі аналізу було виявлено, що наявні рішення часто мають недостатньо зручні інтерфейси, обмежену персоналізацію рекомендацій, низький рівень інтерактивності та відсутність ефективних механізмів перевірки й оцінювання контенту.

Сучасні користувачі потребують платформи, яка не просто пропонувала б каталог рецептів, але й дозволяла б взаємодіяти, обмінюватися досвідом, формувати особисті добірки та отримувати якісні персоналізовані рекомендації. Саме ці фактори визначають актуальність створення нової соціальної мережі, яка поєднає зручний, адаптивний інтерфейс із широкими функціональними можливостями.

На основі проведеного аналізу було сформульовано мету роботи: розробити програмне забезпечення у вигляді соціальної мережі для обміну рецептами, яке забезпечить зручність користування, ефективну взаємодію між користувачами, персоналізацію контенту та інтерактивність платформи.

Для реалізації поставленої мети було визначено конкретні завдання, серед яких: обґрунтування актуальності та аналіз сучасних підходів до створення кулінарних соціальних мереж, вибір відповідних технологій розробки, визначення чітких функціональних і технічних вимог до платформи, безпосередня розробка додатку, а також проведення ретельного тестування для забезпечення відповідності розробленого продукту вимогам і очікуванням користувачів.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Проаналізувавши зростаючий інтерес користувачів до онлайн-спільнот навколо кулінарії, було вирішено створити спеціалізовану соціальну мережу, що забезпечить обмін рецептами, взаємодію та персоналізовані рекомендації. В основі системи лежить підтримка повноцінного життєвого циклу рецепту. Від його створення й ілюстрації до коментування, оцінювання та зберігання в особисті добірки. Для комфортної роботи платформи вона має бути доступною як з мобільних пристроїв, так і з комп'ютерів, з мінімальною затримкою відгуку та інтерактивним інтерфейсом.

Функціональні вимоги що будуть наведені у таблиці 2.1, описують основні можливості, які система повинна мати, щоб відповідати очікуванням користувачів і гарантувати, що платформа працює повноцінно.

Таблиця 2.1 – Функціональні вимоги та їх пріоритет

ID	Вимога	Пріоритет
FR1	Реєстрація, вхід/вихід користувача	Високий
FR2	Створення, редагування та видалення рецептів	Високий
FR3	Коментарі	Високий
FR4	Обмін рецептами	Середній
FR5	Зміна теми застосунку	Середній

Система повинна підтримувати реєстрацію та аутентифікацію користувачів через електронну пошту, забезпечуючи при цьому захист даних за допомогою сучасних алгоритмів шифрування

Необхідно, щоб платформа надала можливість створювати та публікувати нові рецепти за допомогою зручного текстового редактора. Користувач може додати до рецепта список інгредієнтів, та покрокову інструкцію. приготування.

Система повинна підтримувати соціальну взаємодію, дозволяючи користувачам коментувати рецепти, а також додавати їх до списку улюблених.

Система має давати можливість скопіювати текст рецепту чи дати поділитись рецептом з іншими користувачами у різних месенджерах.

У плані продуктивності та масштабованості система повинна забезпечувати швидке реагування, зокрема запити на отримання стрічки новин мають оброблятися не довше ніж за 200 мс. При цьому вона повинна залишатися стабільною та ефективною навіть при великому навантаженні.

Щодо доступності та кросплатформності, застосунок має працювати як на мобільних пристроях (iOS та Android), так і у веб-середовищі, забезпечуючи адаптивний та зручний інтерфейс на будь-якому пристрої.

Надійність і безпека – це один з основних аспектів. Платформа повинна бути захищена від звичайних атак, таких як XSS, CSRF і SQL-ін’єкції. Передача даних відбувається виключно через HTTPS, а паролі користувачів зберігаються в зашифрованому вигляді за допомогою алгоритму bcrypt.

У плані зручності використання інтерфейс має бути інтуїтивно зрозумілим, з чіткою структурою меню та логічною навігацією. Створення нових публікацій повинно проходити швидко і всього за кілька кроків.

Таким чином, аналіз вимог охоплює всі ключові аспекти розробки: від базових функцій соціальної взаємодії до масштабованості, безпеки й підтримки користувацького досвіду, що забезпечить створення сучасної та ефективної платформи для обміну рецептами.

2.2 Вибір інструментів вирішення поставленого завдання

Для клієнтської частини мобільного застосунку існує велика кількість різноманітних фреймворків, на яких можна почати розробку додатку. Вибір відповідного інструменту значно впливає на швидкість розробки, якість і зручність підтримки застосунку. Серед сучасних і популярних рішень виокремлюються такі кросплатформні фреймворки, як Flutter, React Native та Xamarin. Для зручності порівняння їхні ключові характеристики наведено у таблиці (табл. 2.2).

Таблиця 2.2 – Порівняння фрейморків для клієнтської частини

Критерій	Flutter	React Native	Xamarin
Продуктивність	Висока	Середня	Висока
Гнучкість інтерфейсу (UI)	Висока	Середня	Середня
Кросплатформність	Повна	Повна	Повна
Простота розробки	Середня	Висока	Низька
Розмір додатку	Великий	Невеликий	Великий
Документація\спільнота	Висока	Висока	Середня
Підходить для UI-інтенсивних застосунків	Так	Обмежено	Обмежено
Критерій	Flutter	React Native	Xamarin

Flutter – це кросплатформний фреймворк від Google [3], який дозволяє створювати застосунки для Android, iOS, вебу та десктопу, використовуючи єдину кодову базу.

Він написаний на мові Dart, що забезпечує високу продуктивність завдяки компіляції у нативний код. Однією з ключових переваг Flutter є система Hot Reload, яка значно пришвидшує розробку, дозволяючи миттєво бачити зміни в інтерфейсі.

Особливістю фреймворку є власна система віджетів, яка забезпечує гнучкість та візуальну привабливість інтерфейсів, незалежно від платформи. Завдяки підтримці з боку Google та активній спільноті, Flutter стрімко розвивається та має широку екосистему для більшості типових задач.

Разом із тим, фреймворк має і свої недоліки. Оскільки він базується на мові Dart, це може створити певний бар’єр для новачків, які не знайомі з нею. Крім того, застосунки на Flutter зазвичай мають більший фінальний розмір, ніж аналоги, створені на нативних технологіях. Також для деяких специфічних функцій може бракувати готових плагінів, особливо в порівнянні з більш зрілими фреймворками, як-от React Native.

React Native – це фреймворк, створений Facebook [4], який дозволяє розробляти мобільні додатки на JavaScript, використовуючи підходи React. Він особливо цікавий для веб-розробників, оскільки базується на знайомих концепціях компонентного підходу та JSX-синтаксису. Завдяки цьому, вхідний

поріг для тих, хто вже працював з React у вебі, є досить низьким.

Він має велике та активне ком’юніті, що сприяє появі безлічі плагінів, бібліотек і готових рішень. Якщо потрібно, фреймворк дозволяє використовувати нативні модулі, написані на Java чи Swift, що дає можливість розширити функціональність додатку за межі стандартного API.

Також варто враховувати й певні обмеження. У порівнянні з Flutter, React Native часто поступається у продуктивності, особливо в складних інтерфейсах або анімаціях. Крім того, іноді виникають проблеми з сумісністю сторонніх бібліотек або їх оновленням. У деяких випадках для реалізації певного функціоналу все ж доводиться звертатися до написання нативного коду, що може ускладнити підтримку та розгортання додатку.

Xamarin – це фреймворк від Microsoft [5], який дозволяє створювати мобільні застосунки на C# з використанням платформи .NET. Його основна перевага полягає в тісній інтеграції з екосистемою Microsoft, що є особливо важливим для корпоративного середовища. Завдяки можливості використовувати спільний код для різних платформ, Xamarin дозволяє значно зменшити час і зусилля на розробку, особливо коли йдеться про бізнес-додатки з повторюваною логікою.

Крім того, фреймворк добре інтегрується з бізнес-інструментами Microsoft, що робить його привабливим вибором для створення корпоративних мобільних рішень.

Проте Xamarin має й свої недоліки. Спільнота розробників навколо нього менш активна в порівнянні з іншими мобільними фреймворками, такими як Flutter або React Native. Додатки на Xamarin часто мають великий розмір, а процес налаштування середовища розробки може займати більше часу, особливо для новачків.

Можна дійти висновку, що Flutter забезпечує найкращий баланс між продуктивністю, UI-гнучкістю та швидкістю розробки, тому він обраний як основа для клієнтської частини кулінарної соцмережі.

Наступним важливим елементом є серверна частина додатку, а

конкретніше саме фреймворк для розробки API, який буде оброблювати запити і взаємодіяти з нашою базою даних. Розглянемо три популярних фреймворки: FastAPI, Django та Node.js і зведемо всі необхідні дані в таблицю 2.3.

Таблиця 2.3 – Порівняння веб-фреймворків

Критерій	FastAPI	Django	Node.js
Продуктивність	Висока	Середня	Висока
Простота REST API	Висока	Ускладнена	Помірна
Наявність документації	Автоматична	Вручну	Часткова
Синхрон/асинхрон підтримка	Повна	Обмежена	Повна
Мова	Python	Python	JavaScript
Стартовий час	Швидко	Довго	Швидко

FastAPI – це сучасний фреймворк [6] на Python, який призначений для створення високошвидкісних REST API. Його основна перевага – неймовірна продуктивність, що наближається до таких мов, як Node.js і Go. Завдяки підтримці асинхронного програмування через «async/await», FastAPI дозволяє ефективно обробляти велику кількість запитів, що робить його ідеальним вибором для систем з високим навантаженням.

Крім того, FastAPI має простий і зрозумілий синтаксис, а також автоматично генерує документацію API у форматі Swagger [7], що значно спрощує розробку та тестування. Його тісна інтеграція з Pydantic для валідації даних і з SQLAlchemy для роботи з базами даних робить цей фреймворк дуже зручним у повсякденному використанні.

Водночас, варто враховувати, що FastAPI – це відносно молодий проєкт, тому наразі можна зустріти менше прикладів його використання в масштабних проєктах, а також менше готових рішень у порівнянні зі зрілими фреймворками. Ще однією особливістю є відсутність вбудованої адміністративної панелі, подібної до тієї, що пропонує Django.

Django – це потужний веб-фреймворк на Python, який ідеально підходить для створення повноцінних веб-застосунків. Його основна перевага полягає в тому, що він «із коробки» пропонує широкий набір інструментів, включаючи вбудовану ORM, зручну адміністративну панель та систему автентифікації.

Завдяки цьому Django чудово підходить для розробки великих проєктів [8], де важливо швидко створювати функціонал і мати доступ до численних готових рішень і плагінів.

Однак, незважаючи на свої переваги, фреймворк має й певні недоліки. Django працює повільніше в порівнянні з більш легкими рішеннями, такими як FastAPI, що може стати критичним для високонавантажених REST-сервісів. Крім того, його структура може бути надто складною для невеликих застосунків або простих API, а високий рівень абстракції іноді обмежує контроль над низькорівневою логікою, що може бути важливим для деяких специфічних завдань.

Node.js (JavaScript) – це серверне середовище для виконання JavaScript з неблокуючим введенням/виведенням (I/O), яке забезпечує асинхронне виконання коду. Основні його переваги – це висока швидкість роботи, підтримка асинхронності, велика кількість доступних npm-бібліотек, а також можливість використовувати один і той самий стек технологій як для фронтенду, так і для бекенду. Водночас Node.js має й певні недоліки: налагодження може бути ускладненим через асинхронну природу виконання, у базовому вигляді відсутня типізація, а структура проєктів часто є менш чіткою в порівнянні з фреймворками на Python.

FastAPI забезпечує найкращу продуктивність і зручність створення API. Його вибір дозволяє швидко реалізувати бекенд з повноцінною документацією.

Останнім елементом нашого додатку є база даних, яка буде тримати наші дані в безпеці, та реалізовуватиме просту взаємодію між даними та API. Розглянемо декілька з найпопулярніших.

PostgreSQL – це потужна реляційна система управління базами даних, яка славиться своєю надійністю, функціональністю та підтримкою сучасних стандартів. Вона забезпечує повну підтримку транзакцій, контроль цілісності даних і дозволяє працювати як із класичними реляційними структурами, так і з JSON-даними, що робить її гнучким інструментом для різноманітних типів застосунків.

Однією з основних переваг PostgreSQL є можливість виконання складних SQL-запитів, робота з розгалуженими індексами, а також підтримка розширень, таких як PostGIS для геоданих чи pg_trgm для пошуку за подібністю. Це надає широкі можливості для налаштування під специфічні потреби проєкту.

Водночас, порівняно з MySQL, налаштування PostgreSQL може бути дещо складнішим, особливо для новачків. Крім того, при роботі з великими обсягами даних система може бути більш вимогливою до ресурсів, що варто враховувати під час планування інфраструктури.

MySQL – це одна з найвідоміших реляційних систем управління базами даних, яка славиться своєю простотою та легкістю використання.

Завдяки зрозумілому інтерфейсу та швидкому налаштуванню, вона часто стає першим вибором для малих проєктів або в тих випадках, де важлива висока швидкість обробки простих запитів.

Система добре виконує базові операції і забезпечує стабільну роботу в більшості стандартних сценаріїв. Проте, MySQL поступається більш функціональним СУБД, таким як PostgreSQL, коли мова йде про складне моделювання даних, створення складних запитів або використання розширених можливостей.

Серед недоліків варто зазначити менш гнучку підтримку JSON-даних і відсутність деяких сучасних функцій, які можуть бути критично важливими у великих або нестандартних проєктах. Але для звичайних веб-застосунків і простих систем MySQL [9] залишається зручним і ефективним вибором.

MongoDB – це документно-орієнтована NoSQL-база даних, яка вирізняється гнучкою структурою зберігання- даних. Замість традиційних таблиць і рядків, MongoDB використовує документи у форматі BSON (розширення JSON) [10], що дозволяє легко змінювати схему без потреби в складних міграціях. Це робить її особливо зручною для швидкої розробки прототипів і проєктів, де структура даних часто змінюється на ходу.

База даних добре справляється з великими обсягами інформації, забезпечуючи високу швидкість читання й запису, що робить її популярним

вибором для сучасних веб-застосунків і реального часу.

Однак MongoDB [11] має і свої обмеження. Вона не підтримує транзакції на рівні SQL у звичному для реляційних СУБД вигляді (хоча з останніх версій є часткова підтримка багатодокументних транзакцій), що ускладнює побудову складних операцій із гарантованою цілісністю. Крім того, забезпечення узгодженості даних вимагає більше уваги з боку розробника, оскільки відсутня жорстка перевірка цілісності, притаманна реляційним баз.

Результати порівнянь баз даних занесемо в таблицю 2.4.

Таблиця 2.4 – Порівняння баз даних

Критерій	PostgreSQL	MySQL	MongoDB
Реляційність	Повна	Повна	Відсутня
Підтримка транзакцій	Надійна	Є	Часткова
Гнучкість схеми	Жорстка	Жорстка	Висока
Підтримка JSON	Повна	Часткова	Повна
Продуктивність (читання)	Висока	Висока	Висока
Підходить для соцмережі	Так	Обмежено	Обмежено

В результаті порівняння було обрано PostgreSQL, оскільки він найкраще відповідає структурі соціальної мережі з реляційними зв’язками: користувачі → рецепти → коментарі → оцінки.

Фінальні результати вибору занесемо в таблицю 2.5.

Таблиця 2.5 – Остаточний вибір технологій

Рівень	Технологія	Причина вибору
Клієнт	Flutter	Висока продуктивність, гнучкий UI, кросплатформність
Сервер	FastAPI	Простота створення REST API, висока швидкість
База даних	PostgreSQL	Реляційна структура, підтримка транзакцій.

Цей стек забезпечує розробку сучасної, стабільної соціальної мережі з привабливим дизайном, продуктивним API та надійною базою даних. Flutter гарантує зручну клієнтську частину, FastAPI – швидке й масштабоване API, а

PostgreSQL – ефективне зберігання та обробку даних. Разом вони створюють надійну архітектуру для реалізації повноцінного функціоналу.

Висновки до розділу 2

У цьому розділі було проведено детальний аналіз бізнес-вимог і вимог користувачів до нової кулінарної соціальної мережі та обґрунтовано вибір технічного стеку.

Визначено, що система має підтримувати повний життєвий цикл рецепта – від створення й ілюстрування до оцінювання, коментування і зберігання в особисті добірки. Серед ключових функцій – реєстрація/автентифікація, текстовий редактор для публікації рецептів із покроковими інструкціями, соціальна взаємодія. Також окреслено важливі нефункціональні вимоги: час відповіді API ≤ 200 мс, кросплатформність (мобільні й веб-інтерфейси), шифрування паролів (bcrypt), локалізація українською й англійською.

Для фронтенду порівняно Flutter, React Native і Xamarin; найкращим балансом продуктивності, гнучкості UI та швидкості розробки визнано Flutter. На боці бекенду розглянуто FastAPI, Django і Node.js; FastAPI відзначено за високу продуктивність, асинхронність та автоматичну Swagger-документацію. Серед СУБД порівняли PostgreSQL, MySQL і MongoDB; PostgreSQL обрано за надійність транзакцій, підтримку реляційних зв’язків і розширень для повнотекстового пошуку. Остаточний стек: Flutter, FastAPI та PostgreSQL який забезпечує високу швидкість розробки, стабільність, масштабованість та безпеку платформи.

Таким чином, обґрунтовано як бізнес-вимоги, так і технологічний вибір, що гарантує створення сучасної, надійної та зручної для користувачів кулінарної соціальної мережі.

РОЗДІЛ 3
РОЗРОБКА СОЦІАЛЬНОЇ МЕРЕЖІ ДЛЯ ОБМІНУ РЕЦЕПТАМИ

3.1 Практична реалізація соціальної мережі для обміну

Для грамотної розробки API [12] нашого застосунку необхідно мати чітко продуману й організовану архітектуру. Це передбачає створення зрозумілої структури проєкту з логічно названими файлами та папками, що сприяє зручності підтримки й подальшого розширення функціоналу (рис. 3.1).

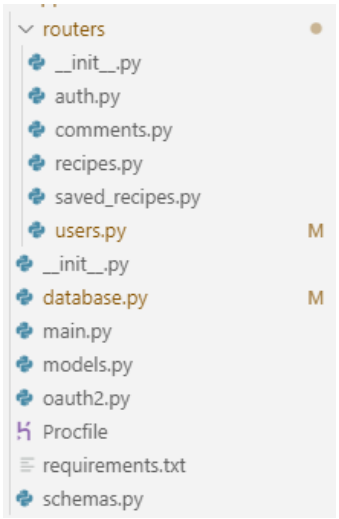

Рисунок 3.1 – Архітектура API

З цією метою проєкт побудовано за модульним підходом, що дозволяє розподілити відповідальності між окремими модулями та полегшує спільну роботу команди.

Одним із ключових компонентів архітектури є каталог Routers, що містить файли-маршрутизатори, які відповідають за обробку HTTP-запитів до різних частин API. Серед них можна виділити такі файли:

– auth.py – реалізує автентифікацію користувачів;
– comments.py – управління коментарями до рецептів;
– recipes.py – операції з рецептами;
– saved_recipes.py – збереження рецептів користувачами;
– users.py – управління користувачами.

Окрім перелічених вище маршрутів, важливими елементами загальної структури архітектури застосунку, які забезпечують стабільність, гнучкість і безпеку роботи системи, виступають такі додаткові модулі:

– database.py – модуль конфігурації та взаємодії з базою даних (SQLAlchemy);

– models.py – оголошення моделей даних, що представляють структуру бази даних;

– schemas.py – Pydantic-схеми для валідації та серіалізації даних;

– oauth2.py – реалізація механізму OAuth2 для автентифікації та авторизації (JWT-токени);

– main.py – основний модуль застосунку, що ініціалізує FastAPI, підключає маршрутизатори та middleware.

Таким чином, така архітектура забезпечує чіткий поділ відповідальностей і зручність для подальшого розвитку системи.

Загалом, фінальна архітектура API поділена за глобальним функціоналом на 2 частини. Перша частина у вигляді файлів: models.py, schemas.py, database.py, oauth2.py, та main.py, декларують загальний вигляд бази даних, та її взаємодію між собою. Друга у вигляді папки routers і її вмісту, відповідає за функціонал окремих компонентів API.

Також є два спеціальних файли які необхідні для правильної роботи на хостингу, а саме Procfile та requirement.txt. В requirements.txt записуються необхідні бібліотеки, яких немає у стандартному наборі Python, які необхідні для роботи програми. Procfile це спеціальний файл, який буде зазначати яким саме чином взаємодіяти файлам на хостингу Heroku. Необхідні налаштування наведено у лістингу 3.1.

Лістинг 3.1 – Налаштування та початкові команди для хостингу Heroku

```
web: uvicorn main:app —host=0.0.0.0 —port=${PORT:–5000}
```

кінець лістингу 3.1

Основною функцією для взаємодії між клієнтською частиною та API є реєстрація та авторизація. Без авторизації у застосунок користувач не зможе створити рецепт та взаємодіяти з ним, а авторизуватись неможливо, попередньо не зареєструвавшись.

Для забезпечення безпечного доступу користувачів до ресурсів програмного забезпечення було реалізовано систему автентифікації та авторизації за допомогою JWT (JSON Web Tokens) [13]. У розробленій системі використовуються бібліотеки python-jose, fastapi.security та ORM-інструменти SQLAlchemy для взаємодії з базою даних.

Основні компоненти реалізованої системи автентифікації у лістингу 3.2 включають такі складові:

– генерацію JWT-токена при автентифікації користувача;

– перевірку валідності JWT-токена та отримання поточного користувача.

Для створення JWT-токена використовується функція create_access_token. Вона приймає словник data, що зазвичай містить ідентифікатор користувача. До цих даних додається час закінчення дії токена, що становить 30 хвилин. Після цього токен шифрується за допомогою секретного ключа (SECRET_KEY) та алгоритму HS256.

Лістинг 3.2 – Реалізація автентифікації та авторизації користувачів із застосуванням JWT-токенів у FastAPI

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/token")
SECRET_KEY = "a3d9d4e5f7c0e9f3d8c2a1e7f4b0c8e9a3f1d2e5c6b7a8d9e0c1f3b4a5c6d7e8"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 30
def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(database.get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED, detail="Could not validate credentials",
        headers={"WWW–Authenticate": "Bearer"},
    )
    try:
```

```
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        user_id = payload.get("user_id")
        if user_id is None:
            raise credentials_exception
    except JWTError:
        raise credentials_exception
    user = db.query(models.User).filter(models.User.id == user_id).first()
    if user is None:
        raise credentials_exception
    return user
```

кінець лістингу 3.2

Алгоритм дій у функції create_access_token здійснює наступні дії:

– отримує JWT-токен із запиту;

– декодує токен, використовуючи секретний ключ та заданий алгоритм;

– перевіряє наявність поля user_id у декодованому токені;

– виконує пошук користувача у базі даних за ідентифікатором, отриманим із токена;

– у разі невдачі будь-якого з етапів перевірки повертається помилка http зі статусом 401.

Реалізований підхід забезпечує:

– короткий термін дії токена (30 хвилин), що зменшує ризики його компрометації;

– захист від несанкціонованого доступу завдяки строгій перевірці ідентифікатора користувача, що міститься в JWT-токені;

– відсутність потреби зберігати стан на сервері, оскільки токени є самодостатніми.

Наступним функціоналом є опис роботи конкретних елементів у папці routers. Файли у цій папці відповідають за реалізацію таких елементів як авторизація і реєстрація на стороні користувача, перегляд профілю користувача, CRUD-функції для рецептів та коментарів, а також функціонал зберігання рецептів в улюблене.

Реалізація процесів авторизації та реєстрації користувача на клієнтській стороні застосунку представлена у лістингу 3.3. Вона включає перевірку

введених даних, відправлення відповідних HTTP-запитів до серверної частини та обробку відповіді для забезпечення доступу до особистого кабінету користувача.

Лістинг 3.3 – Авторизації та реєстрації на стороні користувача

```
router = APIRouter(tags=["auth"])
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
@router.post("/register", response_model=UserProfile)
def register(user: UserCreate, db: Session = Depends(get_db)):
    hashed_pwd = pwd_context.hash(user.password)
    db_user = User(name=user.name, email=user.email, password=hashed_pwd)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
@router.post("/login")
def login(user: UserLogin, db: Session = Depends(get_db)):
    db_user = db.query(User).filter(User.email == user.email).first()
    if not db_user or not pwd_context.verify(user.password, db_user.password):
        raise HTTPException(status.HTTP_403_FORBIDDEN, detail="Invalid credentials")
    token = create_access_token({"user_id": db_user.id})
    return {"access_token": token, "token_type": "bearer"}
@router.post("/token")
def login_for_access_token(form_data: OAuth2PasswordRequestForm = Depends(), db: Session =
Depends(get_db)):
    user = db.query(User).filter(User.email == form_data.username).first()

    if not user or not pwd_context.verify(form_data.password, user.password):
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Invalid credentials")

    token = create_access_token({"user_id": user.id})
    return {"access_token": token, "token_type": "bearer"}
```

кінець лістингу 3.3

Цей код представляє реалізацію аутентифікації користувачів у FastAPI за допомогою JWT-токенів. Для початку створюється роутер («APIRouter»), який відповідає за групування і управління ендпоінтами, що стосуються автентифікації. Для забезпечення безпеки паролів використовується «CryptContext» із алгоритмом хешування «bcrypt».

Перший ендпоінт «/register» відповідає за реєстрацію нового користувача. Користувач передає свої дані (ім'я, email та пароль), після чого пароль безпечно хешується і зберігається в базі даних. Після успішної реєстрації користувач

отримує свій профіль у відповіді, який містить усю необхідну інформацію, окрім пароля, який зберігається лише в хешованому вигляді для захисту даних.

Другий ендпоінт «/login» дозволяє користувачам увійти до свого акаунту, використовуючи email та пароль. Код перевіряє, чи існує користувач із зазначеним email-ом, та чи введений пароль збігається з хешованим паролем у базі. Якщо перевірка проходить успішно, створюється JWT-токен («access_token»), який повертається користувачеві для подальшого використання під час взаємодії із захищеними ресурсами.

Останній ендпоінт «/token» схожий за функціональністю на «/login», проте він використовує стандартну форму OAuth2 («OAuth2PasswordRequestForm») для отримання даних входу. Це дозволяє легко інтегрувати цей сервіс із зовнішніми клієнтами, що підтримують OAuth2, для аутентифікації користувачів та отримання JWT-токенів. JWT-токени є стандартом, який забезпечує надійну ідентифікацію користувачів під час взаємодії з API, а також захист даних завдяки своїй структурі та підпису.

Наступним важливим елементом застосунку є взаємодія з коментарями, код до якого можна побачити у лістингу 3.4.

Лістинг 3.4 – Код для взаємодії коментарів

```
router = APIRouter(prefix="/comments", tags=["comments"])
@router.post("/", response_model=schemas.CommentDisplay)
def create_comment(comment: schemas.CommentCreate,
                   db: Session = Depends(database.get_db),
                   current_user: models.User = Depends(oauth2.get_current_user)):
    recipe = db.query(models.Recipe).filter(models.Recipe.id == comment.recipe_id).first()
    if not recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Recipe not found")
    new_comment = models.Comment(content=comment.content, recipe_id=comment.recipe_id,
user_id=current_user.id)
    db.add(new_comment)
    db.commit()
    db.refresh(new_comment)
    return new_comment
@router.get("/recipe/{recipe_id}", response_model=List[schemas.CommentDisplay])
def get_comments(recipe_id: int, db: Session = Depends(database.get_db)):
    comments = db.query(models.Comment).filter(models.Comment.recipe_id == recipe_id).all()
    return comments
```

кінець лістингу 3.4

Даний код відповідає за реалізацію функціоналу коментарів у додатку, що використовує FastAPI та SQLAlchemy. Вся логіка пов'язана з коментарями згрупована в роутері «comments», який має префікс «/comments» та відповідний тег для зручності документування і структурування коду.

Перший ендпоінт («POST /comments/») дозволяє авторизованим користувачам створювати нові коментарі до конкретних рецептів. Для цього користувач надсилає дані коментаря, що містять текст коментаря та ідентифікатор рецепта. Сервер перевіряє, чи існує рецепт з відповідним ID в базі даних, і, якщо такого рецепта немає, повертає помилку «404 Not Found». У випадку, якщо рецепт знайдено, коментар додається до бази даних, зберігаючи інформацію про зміст коментаря, ідентифікатор рецепта та ідентифікатор користувача, який його створив.

Другий ендпоінт («GET /comments/recipe/{recipe_id}») забезпечує отримання всіх коментарів для конкретного рецепта. Клієнт передає ідентифікатор рецепта, і сервер здійснює запит до бази даних, щоб знайти всі коментарі, пов'язані з цим рецептом. Результатом виконання запиту є список коментарів, які повертаються користувачеві.

Таким чином, цей модуль забезпечує базові функції додавання та отримання коментарів, які є важливою складовою інтерактивної взаємодії користувачів у соціальній мережі для обміну рецептами.

Код, що наведено у лістингу 3.5, описує повноцінний набір операцій з рецептами у соціальній мережі, реалізований за допомогою FastAPI. Усі методи об'єднані в роутері з префіксом «/recipes» для кращої організації API та мають відповідні теги для легкості документації та структурування.

Лістинг 3.5 – Взаємодія з рецептами

```
router = APIRouter(prefix="/recipes", tags=["recipes"])
@router.post("/", response_model=schemas.RecipeDisplay)
def create_recipe(recipe: schemas.RecipeCreate,
                  db: Session = Depends(database.get_db),
                  current_user: models.User = Depends(oauth2.get_current_user)):
    new_recipe = models.Recipe(**recipe.dict(), owner_id=current_user.id)
    db.add(new_recipe)
```

```
    db.commit()
    db.refresh(new_recipe)
    return new_recipe
@router.get("/", response_model=List[schemas.RecipeDisplay])
def get_recipes(db: Session = Depends(database.get_db)):
    recipes = db.query(models.Recipe).all()
    return recipes
@router.get("/{recipe_id}", response_model=schemas.RecipeDisplay)
def get_recipe(recipe_id: int, db: Session = Depends(database.get_db)):
    recipe = db.query(models.Recipe).filter(models.Recipe.id == recipe_id).first()
    if not recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Recipe not found")
    return recipe
@router.delete("/{recipe_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_recipe(recipe_id: int,
                  db: Session = Depends(database.get_db),
                  current_user: models.User = Depends(oauth2.get_current_user)):
    recipe = db.query(models.Recipe).filter(models.Recipe.id == recipe_id).first()
    if not recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Recipe not found")
    if recipe.owner_id != current_user.id:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
                            detail="Not authorized to delete this recipe")
    db.delete(recipe)
    db.commit()
    return None
@router.put("/{recipe_id}", response_model=schemas.RecipeDisplay)
def update_recipe(recipe_id: int,
                  recipe_update: schemas.RecipeCreate,
                  db: Session = Depends(database.get_db),
                  current_user: models.User = Depends(oauth2.get_current_user)):
    recipe_query = db.query(models.Recipe).filter(models.Recipe.id == recipe_id)
    recipe = recipe_query.first()
    if not recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Recipe not found")
    if recipe.owner_id != current_user.id:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
                            detail="Not authorized to update this recipe")
    recipe_query.update(recipe_update.dict(), synchronize_session=False)
    db.commit()
    db.refresh(recipe)
    return recipe
```

кінець лістингу 3.5

Перший ендпоінт («POST /recipes/») забезпечує можливість створення нового рецепту. Користувач надсилає інформацію про рецепт, яка автоматично поєднується з ID поточного авторизованого користувача як автора рецепту. Рецепт зберігається у базі даних, і створений рецепт повертається у відповідь.

Другий ендпоінт («GET /recipes/») дозволяє отримати список усіх рецептів у системі. Цей метод здійснює запит до бази даних для отримання повного списку рецептів і повертає їх клієнту.

Третій ендпоінт («GET /recipes/{recipe_id}») повертає інформацію про конкретний рецепт, визначений його ID. Якщо рецепт із вказаним ідентифікатором не знайдено, користувач отримує повідомлення про помилку з кодом «404».

Четвертий ендпоінт («DELETE /recipes/{recipe_id}») дозволяє автору рецепта видалити власний рецепт. Код перевіряє існування рецепту і право користувача на його видалення (тільки автор може видалити). У разі відсутності рецепту повертається помилка «404», а при спробі видалення рецепту іншого користувача видається помилка «403». Успішне видалення не повертає контенту («204 No Content»).

Останній ендпоінт («PUT /recipes/{recipe_id}») відповідає за оновлення рецепту. Як і у випадку з видаленням, здійснюється перевірка існування рецепту та авторизації користувача. Якщо всі умови виконані, рецепт оновлюється новими даними, що надійшли від користувача, та повертається оновлена інформація.

Цей модуль забезпечує комплексне управління рецептами в додатку, підтримує безпеку та зручність взаємодії з даними, а також є основою для інтерактивності та персоналізації кулінарної соціальної мережі.

Код, що наведений у лістингу 3.6, реалізує функціонал збережених рецептів у соціальній мережі для обміну рецептами, забезпечуючи зручне управління особистими добірками користувачів. Усі дії об'єднані в окремий роутер із префіксом «/saved-recipes» для логічної структуризації API.

Лістинг 3.6 – Взаємодія з збереженням розділу

```
router = APIRouter(prefix=»/saved–recipes», tags=[«saved–recipes»])
@router.post(«/{recipe_id}», response_model=schemas.RecipeDisplay)
def save_recipe(recipe_id: int,
        db: Session = Depends(database.get_db),
        current_user: models.User = Depends(oauth2.get_current_user)):
```

```
    recipe = db.query(models.Recipe).filter(models.Recipe.id == recipe_id).first()
    if not recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail=»Recipe not found»)
    existing_save = db.query(models.SavedRecipe).filter(
        models.SavedRecipe.user_id == current_user.id,
        models.SavedRecipe.recipe_id == recipe_id
    ).first()
    if existing_save:
        raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                            detail=»Recipe already saved»)
    new_saved_recipe = models.SavedRecipe(
        user_id=current_user.id,
        recipe_id=recipe_id
    )
    db.add(new_saved_recipe)
    db.commit()
    return recipe
@router.get(«/», response_model=List[schemas.RecipeDisplay])
def get_saved_recipes(db: Session = Depends(database.get_db),
                      current_user: models.User = Depends(oauth2.get_current_user)):
    saved_recipes = db.query(models.Recipe).join(
        models.SavedRecipe,
        models.SavedRecipe.recipe_id == models.Recipe.id
    ).filter(
        models.SavedRecipe.user_id == current_user.id
    ).all()
    return saved_recipes
@router.delete(«/{recipe_id}», status_code=status.HTTP_204_NO_CONTENT)
def remove_saved_recipe(recipe_id: int,
                        db: Session = Depends(database.get_db),
                        current_user: models.User = Depends(oauth2.get_current_user)):
    saved_recipe = db.query(models.SavedRecipe).filter(
        models.SavedRecipe.user_id == current_user.id,
        models.SavedRecipe.recipe_id == recipe_id
    ).first()
    if not saved_recipe:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
                            detail=»Saved recipe not found»)
    db.delete(saved_recipe)
    db.commit()
    return None
```

кінець лістингу 3.6

Перший ендпоінт («POST /saved-recipes/{recipe_id}») дозволяє користувачу зберегти вибраний рецепт у свої особисті добірки. Сервер перевіряє, чи рецепт із зазначеним ID існує у базі даних. Якщо рецепт не знайдено, повертається помилка «404». Далі виконується перевірка, чи не було раніше цей рецепт уже збережено поточним користувачем; якщо так, користувач отримує

помилку «400». Якщо ж перевірки пройшли успішно, створюється новий запис про збережений рецепт, який зберігається у базі.

Другий ендпоінт («GET /saved-recipes/») дозволяє авторизованому користувачеві отримати список усіх рецептів, які він раніше зберіг. Використовуючи операцію JOIN між таблицями рецептів та збережених рецептів, сервер повертає список лише тих рецептів, що були додані користувачем у його персональну колекцію.

Третій ендпоінт («DELETE /saved-recipes/{recipe_id}») відповідає за видалення збереженого рецепта із персональної добірки користувача. Сервер перевіряє, чи існує запис у таблиці збережених рецептів для цього користувача та відповідного рецепта. Якщо такого запису не знайдено, користувач отримує помилку «404». В іншому разі, запис видаляється з бази даних, і повертається статус «204 No Content».

Таким чином, цей модуль забезпечує просте і зручне управління персональними добірками рецептів, що підвищує зручність та індивідуалізацію користувацького досвіду.

Цей код відповідає за реалізацію отримання інформації про поточного авторизованого користувача у соціальній мережі, створеній на FastAPI.

У роутері (ліст. 3.7) визначено ендпоінт («GET /users/me»), який дозволяє користувачу отримати власний профіль після успішної автентифікації. Метод використовує залежність «get_current_user», яка забезпечує отримання інформації про авторизованого користувача на основі JWT-токена.

Лістинг 3.7 – Взаємодія з користувачем

```
router = APIRouter(prefix="/users", tags=["users"])
@router.get("/me", response_model=UserProfile)
def get_user_profile(current_user: User = Depends(get_current_user)):
    return current_user
```

кінець лістингу 3.7

Як результат, користувач отримує повну інформацію про свій профіль, представлену відповідно до моделі «UserProfile». Це дозволяє клієнтським

застосункам зручно та безпечно отримувати персональні дані користувача без додаткових запитів або зайвих перевірок.

Як і з серверною частиною, архітектура для застосунку [14] має бути легка, зрозуміла, та проста для розширення у майбутньому (рис. 3.2).

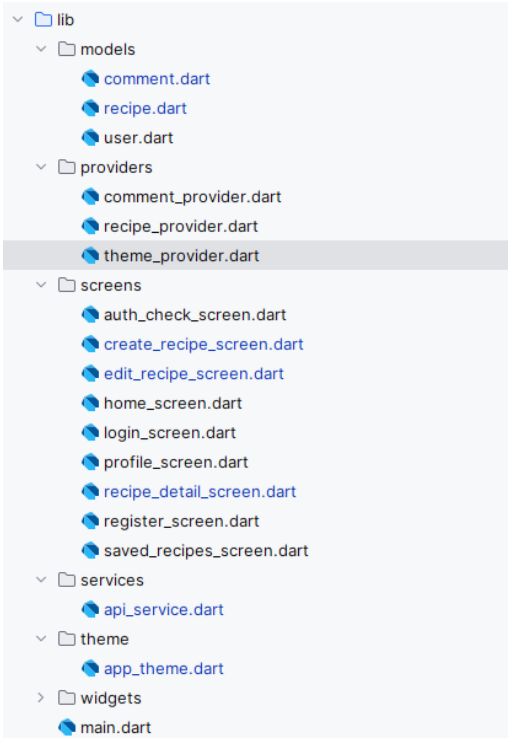

Рисунок 3.2 – Архітектура проєкту

Основний каталог, у якому розміщено весь код клієнтської частини застосунку. Його структура включає такі основні підкаталоги:

Цей модуль містить опис сутностей предметної області у вигляді Dart-класів:

– «comment.dart – описує структуру коментаря;

– «recipe.dart» – містить модель кулінарного рецепта;

– «user.dart» – відповідає за структуру користувача.

У папці «models» використовується код для десеріалізації даних, отриманих з API, а також для формування об'єктів, що передаються на сервер.

Провайдери реалізують керування станом застосунку та забезпечують реактивне оновлення інтерфейсу відповідно до змін у даних:

– «comment_provider.dart» – керує станом коментарів;

– «recipe_provider.dart» – обробляє логіку, пов’язану з рецептами;

– «theme_provider.dart» – відповідає за перемикання теми інтерфейсу (світла/темна).

У цьому модулі зібрано всі екрани (сторінки) застосунку, кожен з яких відповідає за окрему функціональність:

– «auth_check_screen.dart» – екран перевірки автентифікації користувача;

– «create_recipe_screen.dart» – створення нового рецепта;

– «edit_recipe_screen.dart» – редагування існуючого рецепта;

– «home_screen.dart» – головна сторінка застосунку;

– «login_screen.dart» – форма входу в акаунт;

– «profile_screen.dart» – профіль користувача;

– «recipe_detail_screen.dart» – детальний перегляд рецепту;

– «register_screen.dart» – реєстрація нового користувача;

– «saved_recipes_screen.dart» – перегляд збережених рецептів;

– «search_results_screen.dart» – відображення результатів пошуку.

Каталог «services» призначений для зберігання логіки взаємодії клієнтської частини додатку із серверною стороною (API). У файлі «api_service.dart» реалізовано відправлення HTTP-запитів до бекенду, зокрема для таких операцій, як створення, отримання, редагування та видалення рецептів, а також здійснення автентифікації користувачів. Модуль «app_theme.dart», визначає зовнішній вигляд інтерфейсу застосунку. Він містить налаштування стилістичних параметрів, серед яких кольорова палітра, вибір шрифтів, оформлення елементів інтерфейсу, а також можливість перемикання між світлою та темною темами.

Папка «widgets» призначена для збереження багаторазово використовуваних компонентів інтерфейсу, що покращує повторне використання коду та підтримку UI-компонентів.

Файл «main.dart» є точкою входу до застосунку. Він містить ініціалізацію основних компонентів, підключення провайдерів та налаштування маршрутизації між екранами.

Архітектура мобільного застосунку реалізована відповідно до принципів чистого коду, що сприяє чіткому розподілу відповідальностей, покращенню читабельності та спрощенню підтримки й масштабування системи.

Головним файлом є main.dart (ліст. 3.8), де визначено функцію main() для запуску застосунку через runApp(). Кореневим елементом інтерфейсу є віджет MyApp, який відповідає за базову конфігурацію: структуру керування станом, теми оформлення та стартовий екран AuthCheckScreen. Цей екран автоматично перевіряє наявність дійсного токена автентифікації та перенаправляє користувача до відповідного інтерфейсу.

Лістинг 3.8 – Головний файл застосунку

```
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => ThemeProvider()),
        ChangeNotifierProvider(create: (_) => RecipeProvider()),
        ChangeNotifierProvider(create: (_) => CommentProvider()),
      ],
      child: Consumer<ThemeProvider>(
        builder: (context, themeProvider, child) {
          return MaterialApp(
            title: 'Recipe App',
            themeMode: themeProvider.themeMode,
            theme: ThemeData(
              useMaterial3: true,
              colorScheme: ColorScheme.fromSeed(
                seedColor: Colors.orange,
                brightness: Brightness.light,
              ),
            ),
            darkTheme: ThemeData(
              useMaterial3: true,
              colorScheme: ColorScheme.fromSeed(
```

```
            seedColor: Colors.orange,
            brightness: Brightness.dark,
          ),
        ),
        home: const AuthCheckScreen(),
      );
    },
  ),
 );
 }
}
```

кінець лістингу 3.8

Для керування станом застосовується MultiProvider, який дозволяє об'єднати кілька провайдерів у спільну ієрархію. Серед них провайдер теми інтерфейсу, провайдер рецептів і провайдер коментарів. Такий підхід забезпечує централізовану обробку даних і зручне оновлення інтерфейсу при зміні стану.

Інтерфейс конфігурується через MaterialApp, де вказані основні параметри застосунку: назва, теми (світла і темна) та початковий екран. Зміна теми реалізована через Consumer, що перебудовує інтерфейс залежно від активного режиму.

Для стандартизації верхньої панелі застосовується компонент CustomAppBar (ліст. 3.9), який адаптується до обраної теми. Він підтримує як стандартний AppBar, так і SliverAppBar.large, в залежності від контексту екрану. Іконка перемикання теми оновлюється динамічно через ThemeProvider.

Лістинг 3.9 – Адаптована верхня панель

```
class CustomAppBar extends StatelessWidget implements PreferredSizeWidget {
 final String title;
 final List<Widget>? Actions;
 final bool pinned;
 final bool large;
 const CustomAppBar({
  super.key,
  required this.title,
  this.actions,
  this.pinned = true,
  this.large = true,
 });
 @override
```

```
  Widget build(BuildContext context) {
    final themeProvider = context.watch<ThemeProvider>();
    final defaultActions = [
      IconButton(
        icon: Icon(
          themeProvider.isDarkMode ? Icons.light_mode : Icons.dark_mode,
        ),
        onPressed: () {
          themeProvider.setThemeMode(
            themeProvider.isDarkMode ? ThemeMode.light : ThemeMode.dark,
          );
        },
      ),
    ];
    return large
        ? SliverAppBar.large(
            title: Text(title),
            pinned: pinned,
            actions: actions ?? defaultActions,
          )
        : SliverAppBar(
            title: Text(title),
            pinned: pinned,
            actions: actions ?? defaultActions,
          );
  }
  @override
  Size get preferredSize => const Size.fromHeight(kToolbarHeight);
}
```

кінець лістингу 3.9

Провайдер теми (ліст. 3.10) керує режимами відображення (світлий, темний, системний) із використанням SharedPreferences для збереження налаштувань. Зміни автоматично повідомляються інтерфейсу через ChangeNotifier, що дозволяє миттєво оновлювати зовнішній вигляд програми.

Лістинг 3.10 – Провайдер для зміни теми

```
class ThemeProvider with ChangeNotifier {
  static const String _themeKey = 'theme_mode';
  late SharedPreferences _prefs;
  ThemeMode _themeMode = ThemeMode.system;
  ThemeMode get themeMode => _themeMode;
  ThemeProvider() {
    _loadThemeMode();
  }
  Future<void> _loadThemeMode() async {
```

```
    _prefs = await SharedPreferences.getInstance();
    final savedTheme = _prefs.getString(_themeKey);
    if (savedTheme != null) {
      _themeMode = ThemeMode.values.firstWhere(
        (mode) => mode.toString() == savedTheme,
        orElse: () => ThemeMode.system,
      );
      notifyListeners();
    }
  }
  Future<void> setThemeMode(ThemeMode mode) async {
    _themeMode = mode;
    await _prefs.setString(_themeKey, mode.toString());
    notifyListeners();
  }
  bool get isDarkMode => _themeMode == ThemeMode.dark;
}
```

кінець лістингу 3.10

Клас APIService (додаток А) відповідає за взаємодію з сервером через HTTP-запити. Він централізує всю мережеву логіку, що охоплює автентифікацію, роботу з рецептами, коментарями та збереженими даними. Методи класу реалізують операції реєстрації, авторизації, обробки JWT-токенів, а також CRUD-функції для рецептів. Усі запити автоматично доповнюються токеном у заголовку для безпечної авторизації користувача.

RecipeProvider (додаток Б) забезпечує керування даними про рецепти. Він отримує, створює, редагує й видаляє рецепти, а також синхронізує добірки користувача з сервером. Використання прапорця _isLoading дозволяє інтерфейсу відображати індикатори завантаження. Зміни передаються через notifyListeners(), що забезпечує реактивність інтерфейсу.

CommentProvider (ліст. 3.11) відповідає за повноцінне управління даними коментарів у мобільному застосунку. Він реалізує завантаження списку коментарів, пов’язаних із конкретними рецептами, обробку нових записів та їх збереження. Додавання коментаря супроводжується автоматичним оновленням інтерфейсу користувача після успішного HTTP-запиту до API, що забезпечує зручний та динамічний досвід взаємодії. Всі коментарі зберігаються у внутрішньому списку _comments, який є приватною змінною. У разі змін

(додавання, оновлення або очищення списку) провайдер сповіщає всі підписані віджети через механізм notifyListeners(), що забезпечує реактивність і актуальність відображуваних даних на екрані. Такий підхід дозволяє ефективно підтримувати стан застосунку та забезпечує стабільну інтеграцію з інтерфейсом.

Лістинг 3.11 – Провайдер для коментарів

```
class CommentProvider with ChangeNotifier {
  final APIService _apiService = APIService();
  List<Comment> _comments = [];
  bool _isLoading = false;
  List<Comment> get comments => _comments;
  bool get isLoading => _isLoading;

  Future<void> fetchComments(int recipeId) async {
    _isLoading = true;
    notifyListeners();
    try {
      final response = await _apiService.getComments(recipeId);
      _comments = response.map((json) => Comment.fromJson(json)).toList();
    } catch (e) {
      print('Error fetching comments: $e');
    }
    _isLoading = false;
    notifyListeners();
  }

  Future<void> addComment(int recipeId, String content) async {
    try {
      await _apiService.addComment(recipeId, content);
      await fetchComments(recipeId); // Refresh comments
    } catch (e) {
      print('Error adding comment: $e');
      rethrow;
    }
  }
}
```

кінець лістингу 3.11

Усі провайдери використовують шаблон ChangeNotifier, що дозволяє організувати ефективну синхронізацію стану з інтерфейсом без надлишкових перерисовок. Такий підхід є типовим для сучасних Flutter-застосунків і сприяє побудові масштабованої, структурованої та гнучкої архітектури.

Загалом, застосунок демонструє цілісну архітектурну модель, у якій чітко визначено розмежування відповідальностей між компонентами. Завдяки централізації мережевої логіки, зручному керуванню станом через провайдери та адаптивному інтерфейсу з темною і світлою темами, досягається високий рівень гнучкості, зручності для користувача та потенціалу до розширення функціоналу в майбутньому. Також головною складовою будь якого застосунку є візуальна складова і кожна сторінка яка є у застосунку, описана в папці «screens».

Першим екраном який зустрічає користувач при першому вході в застосунок, це екран авторизації (рис. 3.3), якщо ж користувач ще не зареєстрований, він має можливість натиснути кнопку «Ще немає акаунта? Зареєструватись» і перейти на екран з реєстрацією. Екрани виконані у мінімалістичному стилі, та максимально зрозумілі для взаємодії.

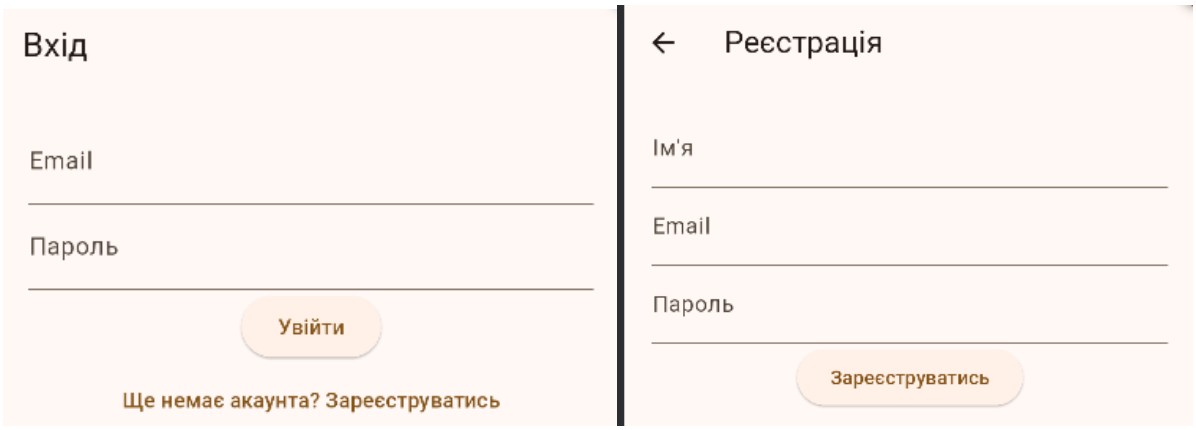

Рисунок 3.3 – Вигляд екрану авторизації та реєстрації

Після входу до системи користувач потрапляє на головний екран застосунку (рис. 3.4), який є основним центром навігації. На ньому розміщено кнопку виходу, доступ до збережених рецептів і кнопку для створення нового рецепту. У центральній частині відображається список усіх наявних рецептів, який можна прокручувати вертикально. Натиснувши на будь-який із них, користувач переходить до екрану з детальною інформацією про обрану страву. Дизайн інтерфейсу залишається лаконічним і інтуїтивно зрозумілим. У разі

відсутності рецептів система також відображає відповідне повідомлення, спонукаючи користувача створити перший запис.

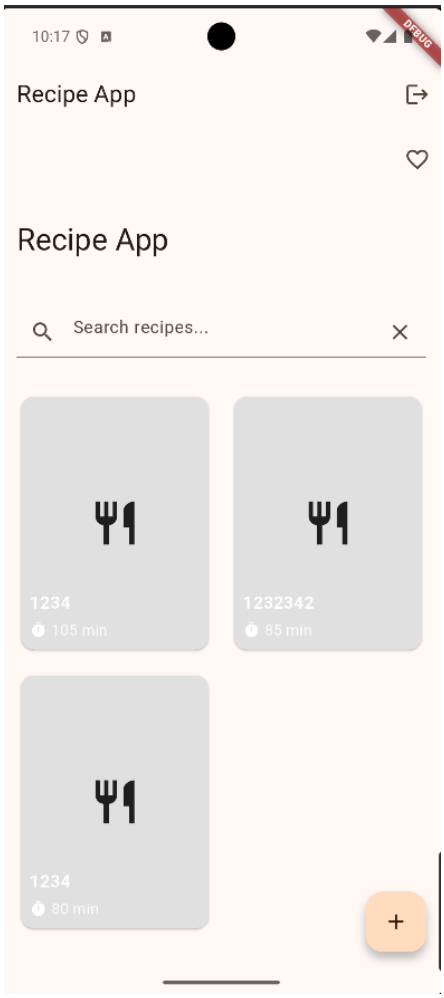

Рисунок 3.4 – Вигляд головного меню

На екрані з деталями рецепту (рис. 3.5) користувач бачить повну інформацію про вибрану страву. Відображаються назва рецепту, час приготування, список необхідних інгредієнтів та ім’я автора. Інтерфейс також містить розділ із коментарями, де можна ознайомитися з відгуками інших користувачів або залишити власний. Усе подано в зручному та зрозумілому форматі для максимальної зручності користувача.

Додатково на цьому екрані розміщено кілька функціональних кнопок. Кнопка збереження дозволяє додати рецепт у власні добірки, а кнопка редагування з’являється лише для автора рецепту. Також доступна функція

«поділитись», яка дозволяє надіслати рецепт через зовнішні додатки або соціальні мережі. Усі ці елементи організовані логічно й гармонійно, зберігаючи мінімалістичний та зручний дизайн інтерфейсу.

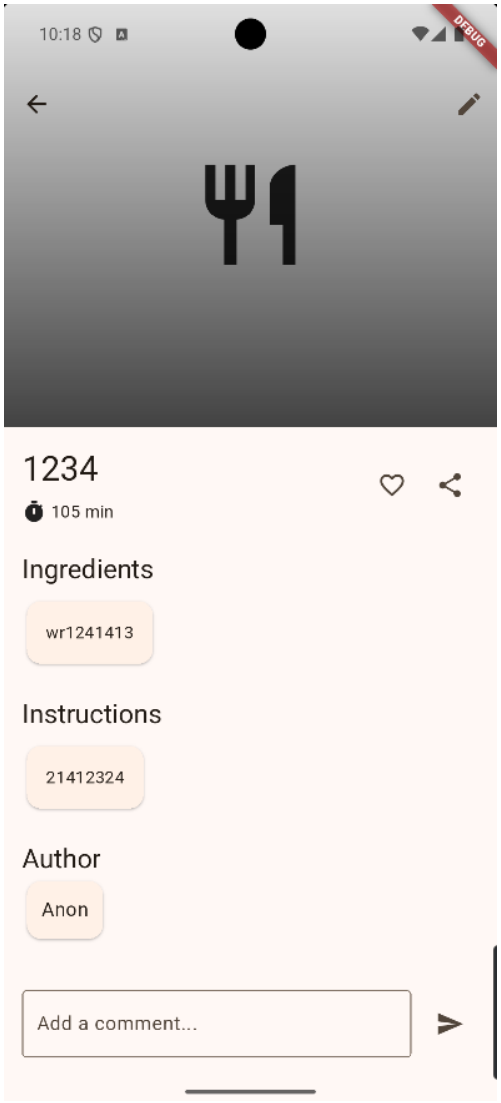

Рисунок 3.5 – Вигляд сторінки з рецептом

Кнопка поділитись, використовує бібліотеки flutter під назвою « share_plus» та «shared_preferences», які дозволяють легко та швидко створити механізм обміну рецепту між нашим застосунком та іншими соціальними мережами. При натисканні кнопки збирається інформація про рецепт, надається можливість зберегти у буфері обміну наш рецепт, або відправити згенерований текст у приватні повідомлення у месенджери (рис. 3.6).

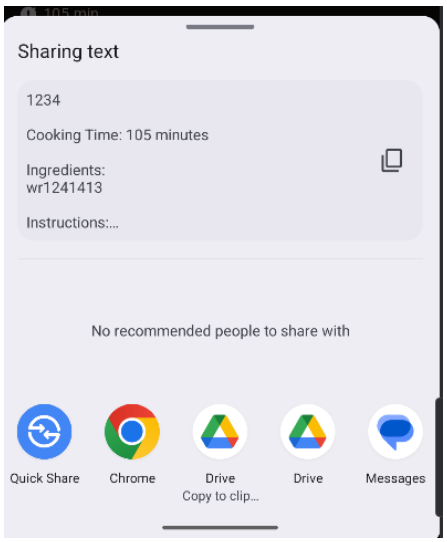

Рисунок 3.6 – Вигляд згенерованого тексту

Якщо користувач є власником рецепту, він може редагувати такі ключові елементи, як назва, список інгредієнтів, детальні інструкції щодо приготування та час, необхідний для завершення страви (рис. 3.7). Це дозволяє підтримувати актуальність інформації та зручно керувати власним контентом. Інтерфейс редагування рецепту також виконаний у мінімалістичному стилі, що робить його зрозумілим, простим у використанні та привабливим для широкого кола користувачів.

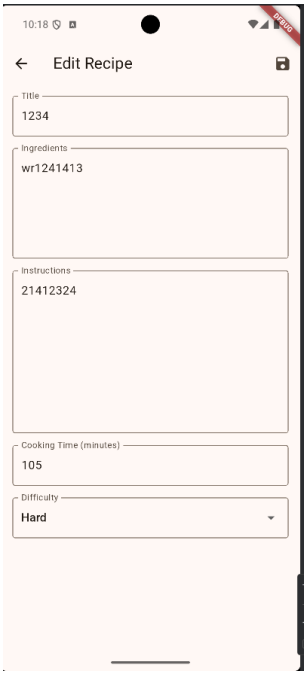

Рисунок 3.7 – Вигляд сторінки з редагуванням

На екрані зі збереженими рецептами (рис. 3.8), можна побачити картки рецептів, які користувач захотів зберегти. Кожен збережений рецепт, може бути видалений, якщо користувач натисне на іконку закладки в картці рецепту.

Також на даному екрані є кнопка для переключення переключені кольорової схеми інтерфейсу, зі світлої на темну. Вибір цього екрану для такого функціоналу, був вибраний, аби не перевантажувати основний екран.

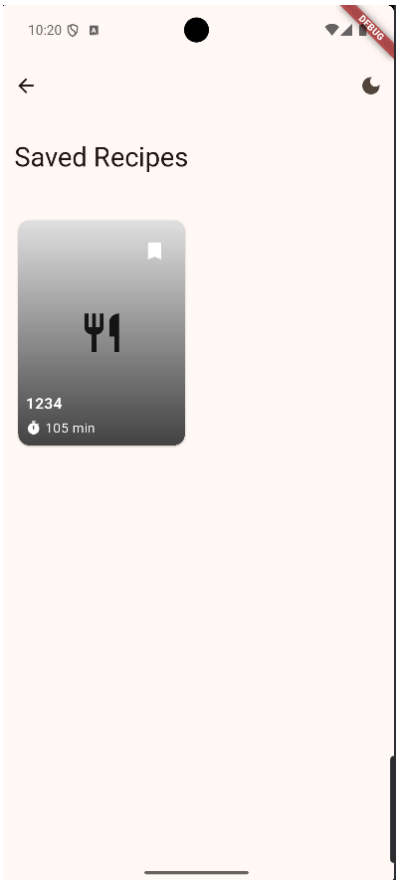

Рисунок 3.8 – Вигляд сторінки зі збереженим рецептом

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

У ході розробки мобільного застосунку було проведено всебічне тестування для перевірки його функціональності, стабільності та зручності використання. Основною метою тестування було виявлення помилок, недоліків у логіці роботи додатку, перевірка відповідності реалізованого функціоналу до

вимог, зазначених на етапі проєктування, а також забезпечення позитивного досвіду взаємодії користувача із системою.

Тестування відбувалось за допомогою інструментів, які надає FastAPI, а саме Swagger UI Documentation (рис. 3.9). Даний інструмент дає змогу зручно та швидко перевіряти розроблений функціонал. Swagger UI також автоматично генерує документовані описи маршрутів та моделей даних, що суттєво прискорює та спрощує процес інтеграційного тестування між клієнтською та серверною частинами.

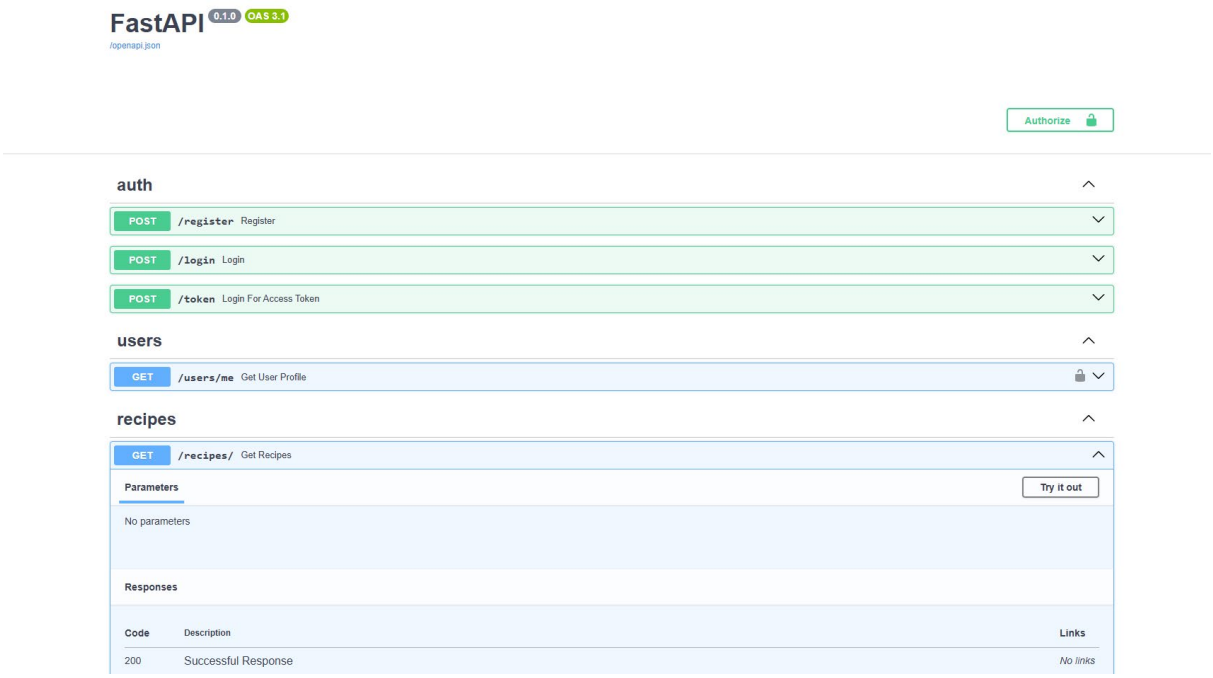

Рисунок 3.9 – Swagger UI Documentation

Першим етапом тестування стало функціональне тестування, яке охоплювало перевірку основних сценаріїв взаємодії користувача з додатком. Зокрема, було протестовано реєстрацію та авторизацію, створення, перегляд, редагування та видалення рецептів, можливість залишати коментарі до рецептів, додавати їх до збережених та переглядати список улюблених. Кожна дія перевірялась у нормальних умовах (при валідних даних) та у випадках, коли користувач вводить некоректні дані, наприклад, неправильний email чи порожні поля. Усі виявлені помилки було виправлено під час розробки.

Другим важливим етапом стало тестування інтерфейсу користувача (UI testing). Воно проводилось вручну на різних пристроях з різними розмірами екранів (smartphone 6", 6.7", tablet), щоб перевірити адаптивність дизайну. Особлива увага приділялася правильному масштабуванню шрифтів, відступам, читабельності тексту, логічності розміщення елементів і відповідності інтерфейсу до очікуваної поведінки. Було виявлено кілька дрібних розбіжностей в компонуванні на малих екранах, які були усунуті шляхом оптимізації верстки за допомогою MediaQuery та LayoutBuilder.

Третім видом перевірки стало тестування продуктивності. Було оцінено швидкодію додатку, час завантаження основних екранів, час відповіді на запити до серверної частини через API. Для цього застосовувались вбудовані інструменти Flutter (DevTools, Performance overlay). Затримка від відкриття додатку до відображення головного екрана не перевищувала 1.2 секунди, а час відповіді сервера на запити REST API був у межах 100-150 мс при стабільному інтернет-з’єднанні. Це відповідає заявленим нефункціональним вимогам і забезпечує плавну взаємодію без помітних затримок.

Також було проведено тестування збереження даних. Перевірено, що інформація, яку вводить користувач, зберігається у базі даних на сервері і коректно відображається при повторному вході в систему. Зокрема, додані рецепти та коментарі залишались у системі навіть після виходу з акаунту або перезапуску додатку.

Особливу увагу приділено тестуванню безпеки. Було перевірено захист реєстрації й авторизації – пароль користувача шифрується на сервері, а доступ до персональних функцій (створення рецептів, збереження, коментування) можливий лише при наявності дійсного JWT-токена. При спробі доступу до захищених маршрутів без авторизації сервер повертає відповідну помилку (401 або 403), що свідчить про правильну реалізацію контролю доступу.

На завершальному етапі виконано кінцеве тестування (end-to-end), яке передбачало проходження всього сценарію від реєстрації нового користувача до публікації рецепта, взаємодії з ним та виходу з акаунту. Цей тест підтвердив

узгодженість роботи всіх компонентів системи, включаючи клієнтську та серверну частини.

У результаті проведеного тестування було підтверджено відповідність реалізованого програмного забезпечення вимогам, визначеним на етапі проєктування, також результати тестування були занесені у таблицю 3. Застосунок продемонстрував стабільну роботу, високу продуктивність та зручний інтерфейс на більшості сучасних пристроїв. Проведене тестування засвідчило готовність мобільного додатку до практичного використання кінцевими користувачами.

Таблиця 3.1 – Результати тестування

№	Тип тестування	Опис перевірки	Результат
1	Функціональне	Реєстрація, авторизація, CRUD-операції з рецептами	Пройдено
2	Інтерфейс (UI)	Адаптивність, правильність відображення на різних екранах	Пройдено
3	Продуктивність	Швидкість запуску додатку, час відповіді API	Пройдено
4	Надійність збереження	Перевірка збереження даних після перезапуску	Пройдено
5	Безпека	Захист авторизації, перевірка доступу до захищених функцій	Пройдено
6	End-to-End	Повний сценарій: від реєстрації до взаємодії з контентом	Пройдено

3.3 Публікація API на хостинг

Проаналізувавши різні хост-системи, дійшли висновку, що найкращим по співвідношенню ціна/якість, буде сайт Heroku. Швидкість, недорога ціна, велика кількість інтеграцій з іншими веб сайтами, безліч можливостей для модифікації проєкту, а також простота налаштування проєкту, робить Heroku одним з найкращих рішень.

Оскільки платформа Heroku має глибоку інтеграцію з GitHub, процес публікації нашого API-сервісу значно спрощується. Достатньо створити новий Git-репозиторій (рис. 3.10), підключити його до Heroku через налаштування автоматичного розгортання (CI/CD), а всі наступні коміти в головну гілку будуть

негайно публікуватися на сервері. Це дозволяє мінімізувати час простою, швидко вносити правки й одразу тестувати нові версії без додаткових ручних операцій.

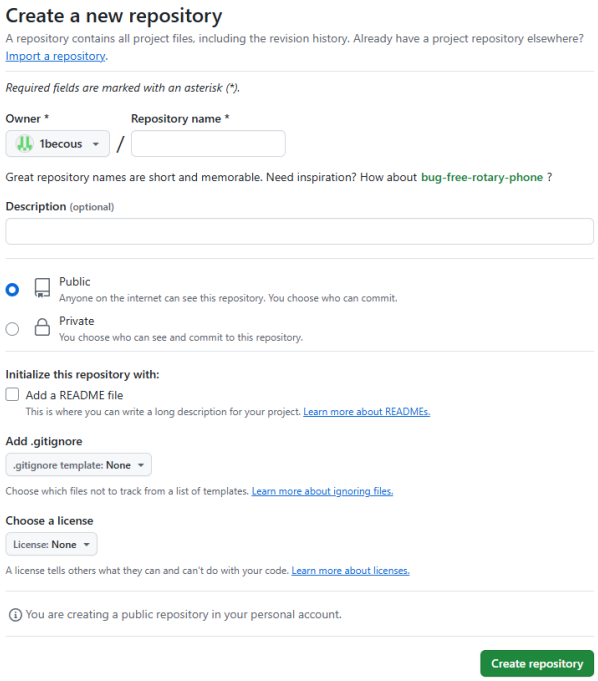

Рисунок 3.10 – Створення репозиторію

Після створення репозиторію, GitHub дає коротку інструкцію, яка за допомогою bash команд (ліст. 3.12) дасть швидко перекинути наш проєкт на цю платформу.

Лістинг 3.12 – Команди для завантаження проєкту на GitHub

```
echo "Recipe API app" >> README.md
git init
git add README.md
git commit –m "first commit"
git branch –M main
git remote add origin https://github.com/1becous/recipeapp.git
git push –u origin main
```

кінець лістингу 3.12

Далі переходимо на платформу Heroku, та створюємо аккаунт. Платформа надає безкоштовний 7-денний пробний період, чого нам для розробки та

тестування застосунку буде вдосталь, а при необхідності, можна буде оформити платну підписку, або перейти на інший хостинг.

Після реєстрації необхідно створити програму на хостингу, вписавши ім’я та вибрати локацію сервера Europe (рис. 3.11).

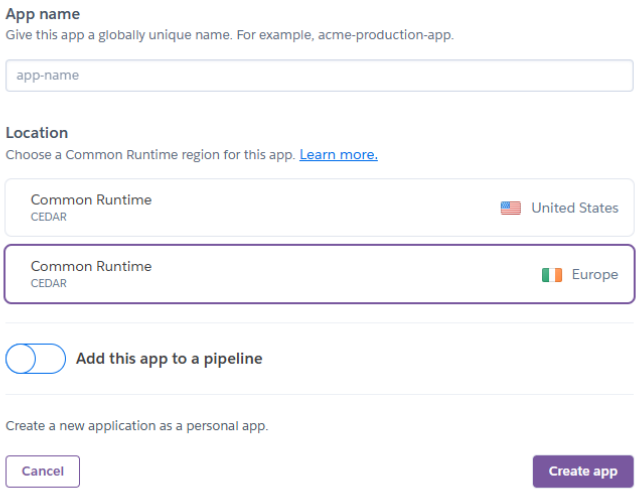

Рисунок 3.11 – Створення програми

Далі, відкривши створену програму, треба перейти на вкладку «Deploy», та вибрати в параметрі «Deployment method» під’єднання до GitHub, авторизуватись в нього та вибрати наш репозиторій як метод для деплою (рис. 3.12).

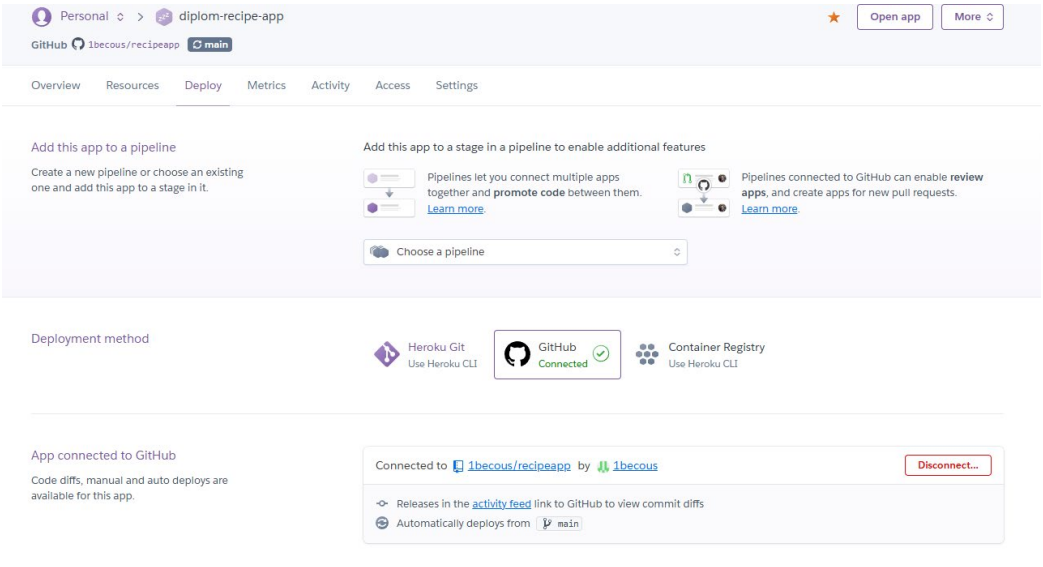

Рисунок 3.12 – Налаштування деплою

Хостинг Heroku дає можливість добавляти різні сервіси до програм, наприклад підтримку різних мов програмування, сервісів, баз даних та багато іншого. Нас цікавить сервіс під назвою «Heroku Postgres», який дозволить нам використовувати базу даних PostgreSQL (рис. 3.13).

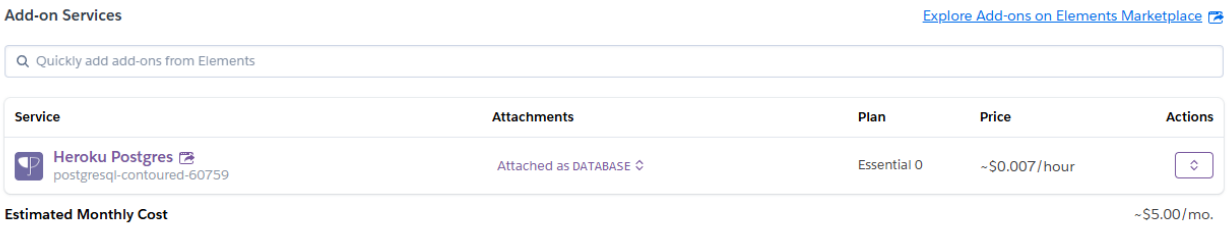

Рисунок 3.13 – Вибір сервісу «Heroku Postgres»

Для того, щоб дізнатись адресу бази даних, треба натиснути на назву сервісу, відкрити налаштування, та в параметрі «Database Credentials» вибрати «View Credentials…». Біля рядка «URL» буде знаходитись наше посилання на базу даних, яке необхідно вставити у наш код (рис. 3.14).

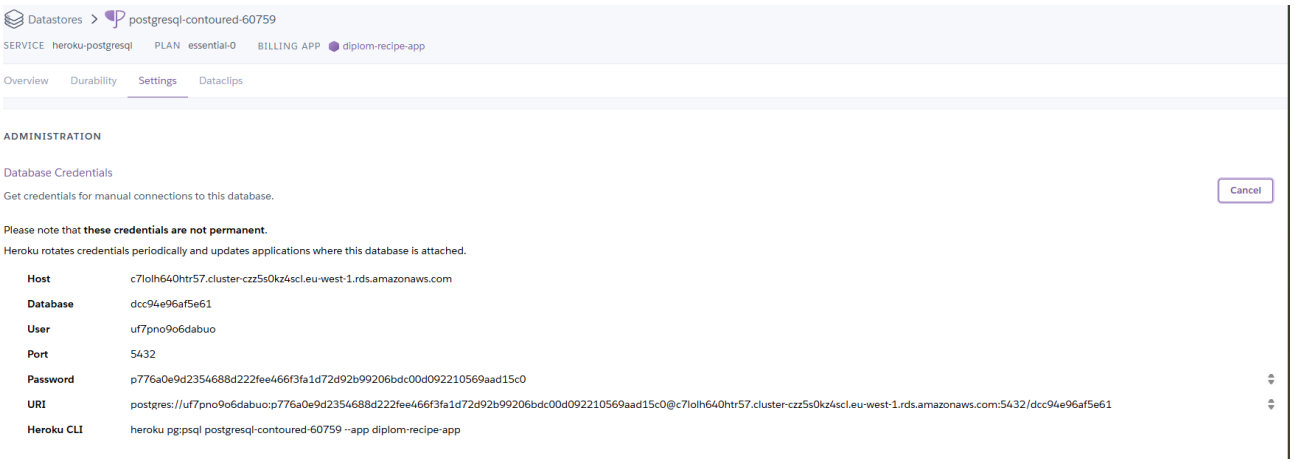

Рисунок 3.14 – Налаштування бази даних

В кінці, перевіряємо працездатність API на самому сайті, та в логах Heroku. При успішному старті, без помилок, на самому сайті нам має показати повідомлення: «{"msg":"API працює!"}» (рис. 3.15).

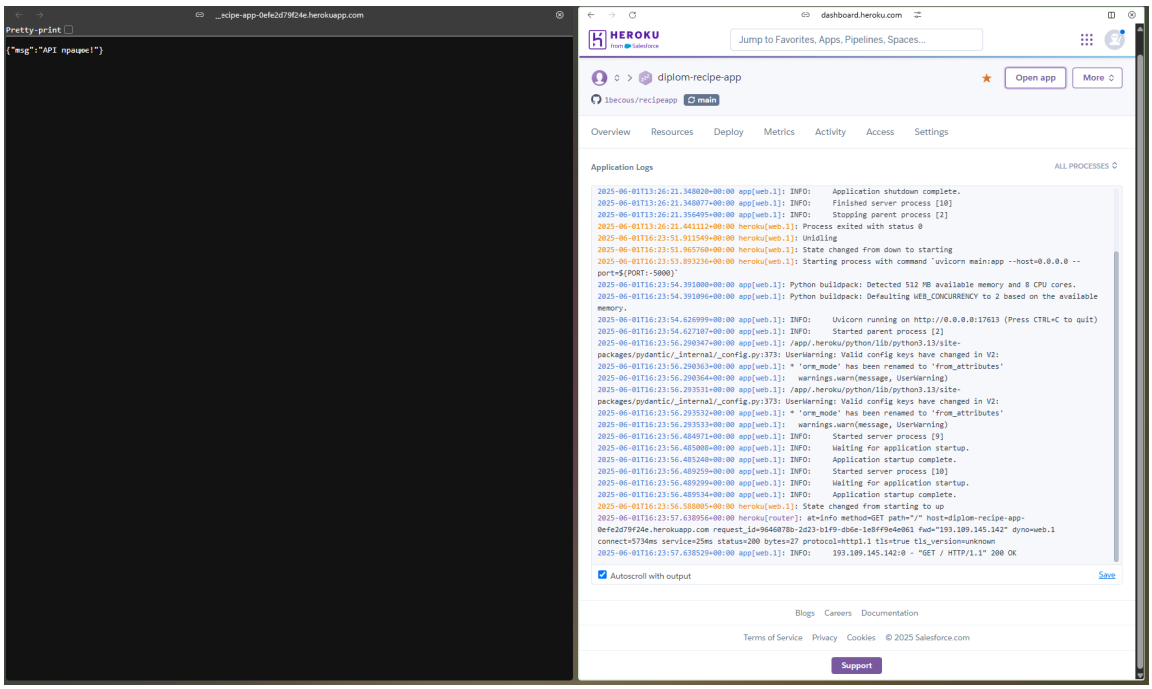

Рисунок 3.15 – Перевірка працездатності API

Основним недоліком платформи Heroku є автоматичний перехід у режим сну застосунку після 15 хв бездіяльності, але час переходу до робочого стану займає менше 20 секунд, тому проблем з обробкою запитів для нашого застосунку не виникне.

Висновки до розділу 3

У даному розділі було детально розглянуто реалізацію серверної частини застосунку кулінарної соціальної мережі на базі фреймворку FastAPI. Основними результатами роботи стали:

– раціональна модульна архітектура: серверна частина реалізована за модульним підходом, що дозволяє чітко поділити відповідальність між файлами. Така структура спрощує розробку, супровід і розширення функціоналу в майбутньому;

– безпечна система автентифікації: впроваджено механізм авторизації та автентифікації користувачів за допомогою JWT-токенів, що забезпечує надійний контроль доступу до ресурсів API без потреби зберігання сесій на сервері;

– підтримка ключового функціоналу: реалізовано основні CRUD-операції для роботи з рецептами, коментарями, збереженими рецептами, а також функціонал користувацьких профілів. Усі маршрути надійно захищені перевіркою прав доступу, що відповідає стандартам розробки Rest API;

– готовність до розгортання: для розміщення проєкту використано хостинг Heroku. Налаштування проєкту включає створення «procfile» та «requirements.txt», що спрощує процес публікації. Забезпечено інтеграцію з GitHub і підключення до PostgreSQL через сервіс Heroku Postgres;

– підвищення продуктивності розробки: завдяки застосуванню SQLAlchemy, Pydantic, bcrypt та інших популярних бібліотек Python забезпечено зручну роботу з базою даних, валідацію вхідних даних та захист користувацьких паролів.

Отже, реалізація застосунку відповідає сучасним вимогам до безпеки, масштабованості та зручності підтримки. Це створює надійну основу для функціонування всієї системи, включаючи мобільний клієнт, і дозволяє в майбутньому легко розширювати її можливості.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було проведено аналіз сучасного стану розробки соціальних мереж для обміну рецептами. Здійснено огляд найпоширенішої платформи Cookpad, визначено основні тенденції у її розвитку та виявлено недоліки, такі як недостатня персоналізація, низька інтерактивність, що зумовило актуальність розробки нової спеціалізованої мережі.

На основі проведеного аналізу були чітко сформульовані функціональні й нефункціональні вимоги до мобільного застосунку. Серед ключових функцій виокремлено реєстрацію користувачів, створення, перегляд, оцінювання та коментування рецептів, а також можливість формувати персональні добірки. Особливу увагу було приділено таким нефункціональним характеристикам, як швидкість роботи, безпека даних та зручність використання.

Було виконано порівняльний аналіз сучасних технологій для розробки кросплатформних застосунків, включаючи Flutter, React Native та Xamarin для клієнтської частини; FastAPI, Django та Node.js для серверної частини; PostgreSQL, MySQL та MongoDB як варіанти бази даних. В результаті аналізу технологій обрано оптимальний стек: Flutter для клієнтської частини завдяки високій продуктивності та гнучкості, FastAPI для створення ефективного REST API, і PostgreSQL за надійність та підтримку складних реляційних запитів.

На основі вибраних технологій було розроблено архітектуру мобільного застосунку, яка включає детальне проектування бази даних та структуру REST API. Розроблена архітектура забезпечує чіткий поділ відповідальностей між компонентами, що дозволяє ефективно керувати функціональністю платформи, спрощує підтримку та масштабування додатку.

У рамках практичної реалізації було створено серверну частину з повним функціоналом авторизації та автентифікації, управління рецептами, коментарями та персональними добірками користувачів. Впроваджено

механізми безпеки із застосуванням JWT-токенів, які забезпечують надійний захист персональних даних користувачів.

Реалізований серверний додаток було успішно розгорнуто на хмарній платформі Heroku, що дозволило легко керувати застосунком у веб-середовищі. Це забезпечило доступність API для клієнтської частини та стабільну роботу сервісу з мінімальним простоєм і високою швидкістю обробки запитів.

Клієнтську частину застосунку створено на базі Flutter, що дозволило реалізувати всі необхідні функції: реєстрацію, публікацію рецептів, коментування, оцінювання і керування персональними добірками. Використання сучасних підходів у дизайні та UX забезпечило інтуїтивно зрозумілий інтерфейс і комфортну взаємодію користувачів з платформою.

Таким чином, усі завдання, окреслені на початку роботи, були виконані в повному обсязі, що підтверджує ефективність обраного підходу та доцільність використаних технологій для реалізації спеціалізованої соціальної мережі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cookpad (cookpad.com). Data Scraping Website. Data Scraping Encyclopedia. AI-Powered Web Scraping Tool & Web Data Extractor ScrapeStorm. URL: https://www.scrapestorm.com/tutorial/cookpadcookpad-com/ (дата звернення: 02.03.2025).

2. Зберігайте, публікуйте та діліться рецептами з кулінарами з усього світу. Cookpad. URL: https://cookpad.com/ua (дата звернення: 02.03.2025).

3. Building a Recipe App with Flutter. Walturn. URL: https://www.walturn.com/insights/building-a-recipe-app-with-flutter (дата вернення: 12.04.2025).

4. React Native vs Xamarin. Which One to Choose. PixelCrayons. URL: https://www.pixelcrayons.com/blog/dedicated-teams/react-native-vs-xamarin/ (дата звернення: 14.04.2025).

5. React Native vs Xamarin. Яка платформа для розробки краща у 2022 році? Wezom. URL: https://wezom.com.ua/ua/blog/react-native-vs-xamarin (дата звернення: 14.04.2025).

6. Django vs. Flask vs React vs Node.js vs FastAPI vs Rails. Ritza. URL: https://ritza.co/articles/django-vs-flask-vs-react-vs-node-vs-fastapi-vs-rails (дата звернення: 15.04.2025).

7. Що таке Node JS простими словами – застосування Node JS у програмуванні – DAN-IT. URL: https://dan-it.com.ua/uk/blog/chto-jeto-takoe-node-js-prostymi-slovami/ (дата звернення: 21.04.2025).

8. Node JS vs Django. Which One is Better? TatvaSoft Blog. URL: https://www.tatvasoft.com/outsourcing/2022/10/node-js-vs-django.html (дата звернення: 21.04.2025).

9. MongoDB vs MySQL. A Detailed Unbiased Guide (2025). Astera. URL: https://www.astera.com/type/blog/mongodb-vs-mysql (дата звернення: 01.05.2025).

10. Comparing The Differences – MongoDB Vs MySQL. URL: https://www.mongodb.com/resources/compare/mongodb-mysql (дата звернення: 10.05.2025).

11. MongoDB vs MySQL – Simplilearn. URL: https://www.simplilearn.com/tutorials/mongodb-tutorial/mongodb-vs-mysql (дата звернення: 15.05.2025).

12. RESTful API with Python, SQLAlchemy & FastAPI + PostgreSQL. GitHub. URL: https://github.com/wpcodevo/python_fastapi (дата звернення: 20.05.2025)

13. OAuth2 with Password (and hashing), Bearer with JWT tokens. FastAPI documentation. URL: https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt (дата звернення: 21.05.2025).

14. Recipe Finder Application in Flutter. GeeksforGeeks. URL: https://www.geeksforgeeks.org/recipe-finder-application-in-flutter (дата звернення: 21.05.2025).

ДОДАТКИ

Додаток А – Взаємодія з API

```
class APIService {
  static const String baseUrl = 'https://diplom-recipe-app-0efe2d79f24e.herokuapp.com'; // Change this to your
API URL
  // Auth endpoints
  Future<Map<String, dynamic>> login(String email, String password) async {
    final response = await http.post(
      Uri.parse('$baseUrl/login'),
      headers: {
        'Content-Type': 'application/json',
      },
      body: jsonEncode({
          'email': email,
          'password': password,
        }),
    );
    final data = json.decode(response.body);
    final token = data['access_token'];
    await SharedPreferences.getInstance().then((prefs) {
      prefs.setString('token', token);
    });
    return data;
  }
  Future<Map<String, dynamic>> register(String email, String password, String username) async {
    final response = await http.post(
      Uri.parse('$baseUrl/register'),
      headers: {
        'Content-Type': 'application/json',
      },
      body: jsonEncode({
        'email': email,
        'password': password,
        'name': username,
      }),
    );
    return json.decode(response.body);
  }
  // Recipes endpoints
  Future<List<dynamic>> getRecipes() async {
    final response = await http.get(Uri.parse('$baseUrl/recipes/'));
    return json.decode(response.body);
  }
  Future<Map<String, dynamic>> getRecipe(int id) async {
    final response = await http.get(Uri.parse('$baseUrl/recipes/$id'));
    return json.decode(response.body);
  }
  Future<Map<String, dynamic>> createRecipe(Map<String, dynamic> recipe) async {
    final prefs = await SharedPreferences.getInstance();
    final token = prefs.getString('token');
    final response = await http.post(
      Uri.parse('$baseUrl/recipes/'),
      headers: {
```

```
      'Content-Type': 'application/json',
      'Authorization': 'Bearer $token',
    },
    body: json.encode(recipe),
  );
  if (response.statusCode == 200 || response.statusCode == 201) {
    return json.decode(response.body);
  } else {
    throw Exception('Failed to create recipe: ${response.statusCode}\n${response.body}');
  }
}
// Saved Recipes endpoints
Future<List<dynamic>> getSavedRecipes() async {
  final prefs = await SharedPreferences.getInstance();
  final token = prefs.getString('token');
  final response = await http.get(
    Uri.parse('$baseUrl/saved-recipes'),
    headers: {'Authorization': 'Bearer $token'},
  );
  return json.decode(response.body);
}
Future<Map<String, dynamic>> saveRecipe(int recipeId) async {
  final prefs = await SharedPreferences.getInstance();
  final token = prefs.getString('token');
  final response = await http.post(
    Uri.parse('$baseUrl/saved-recipes/$recipeId'),
    headers: {'Authorization': 'Bearer $token'},
  );
  return json.decode(response.body);
}
Future<bool> removeSavedRecipe(int recipeId) async {
  final prefs = await SharedPreferences.getInstance();
  final token = prefs.getString('token');
  final response = await http.delete(
    Uri.parse('$baseUrl/saved-recipes/$recipeId'),
    headers: {'Authorization': 'Bearer $token'},
  );
  return response.statusCode == 204;
}
// Comments endpoints
Future<List<dynamic>> getComments(int recipeId) async {
  final response = await http.get(Uri.parse('$baseUrl/comments/recipe/$recipeId'));
  final data = json.decode(response.body);
  if (data is List) {
    return data;
  } else {
    if (kDebugMode) {
      print('Unexpected response for comments: $data');
    }
    return [];
  }
}
```

```
Future<Map<String, dynamic>> addComment(int recipeId, String content) async {
  final prefs = await SharedPreferences.getInstance();
  final token = prefs.getString('token');
  final response = await http.post(
    Uri.parse('$baseUrl/comments/'),
    headers: {'Authorization': 'Bearer $token'},
    body: {
      'recipe_id': recipeId.toString(),
      'content': content,
    },
  );
  return json.decode(response.body);
}
// отримати поточного користувача
Future<User?> fetchCurrentUser(String token) async {
  final res = await http.get(Uri.parse('$baseUrl/users/me/'),
      headers: {"Authorization": "Bearer $token"});
  if (res.statusCode == 200) {
    return User.fromJson(jsonDecode(res.body));
  }
  return null;
}
// отримати один рецепт
Future<Recipe?> fetchRecipeById(String token, int id) async {
  final res = await http.get(Uri.parse('$baseUrl/recipes/$id/'),
      headers: {"Authorization": "Bearer $token"});
  if (res.statusCode == 200) {
    return Recipe.fromJson(jsonDecode(res.body));
  }
  return null;
}
// Delete a recipe by ID
Future<bool> deleteRecipe(String token, int recipeId) async {
  final res = await http.delete(
    Uri.parse('$baseUrl/recipes/$recipeId/'),
    headers: {"Authorization": "Bearer $token"},
  );
  return res.statusCode == 200;
}
// Edit an existing recipe
Future<bool> editRecipe(
  String token,
  int recipeId,
  String title,
  String ingredients,
  String instructions,
  int cookingTime,
  int difficulty,
) async {
  final res = await http.put(
    Uri.parse('$baseUrl/recipes/$recipeId/'),
    headers: {
```

```
      "Authorization": "Bearer $token",
      "Content-Type": "application/json"
    },
    body: jsonEncode({
      "title": title,
      "ingredients": ingredients,
      "instructions": instructions,
      "cooking_time": cookingTime,
      "difficulty": difficulty,
    }),
  );
  return res.statusCode == 200;
 }
}
```

Кінець додатку А

Додаток Б – Провайдер рецептів.dart

```
class RecipeProvider with ChangeNotifier {
 final APIService _apiService = APIService();
 List<Recipe> _recipes = [];
 List<Recipe> _savedRecipes = [];
 bool _isLoading = false;
 List<Recipe> get recipes => _recipes;
 List<Recipe> get savedRecipes => _savedRecipes;
 bool get isLoading => _isLoading;
 Future<void> fetchRecipes() async {
  _isLoading = true;
  notifyListeners();
  try {
   final response = await _apiService.getRecipes();
   _recipes = response.map((json) => Recipe.fromJson(json)).toList();
  } catch (e) {
   print('Error fetching recipes: $e');
  }
  _isLoading = false;
  notifyListeners();
 }
 Future<void> fetchSavedRecipes() async {
  _isLoading = true;
  notifyListeners();
  try {
   final response = await _apiService.getSavedRecipes();
   _savedRecipes = response.map((json) => Recipe.fromJson(json)).toList();
  } catch (e) {
   print('Error fetching saved recipes: $e');
  }
  _isLoading = false;
  notifyListeners();
 }
```

```
Future<void> createRecipe(Recipe recipe) async {
  try {
    await _apiService.createRecipe(recipe.toJson());
    await fetchRecipes(); // Refresh the recipe list
  } catch (e) {
    print('Error creating recipe: $e');
    rethrow;
  }
}
Future<void> saveRecipe(int recipeId) async {
  try {
    await _apiService.saveRecipe(recipeId);
    await fetchSavedRecipes(); // Refresh saved recipes
  } catch (e) {
    print('Error saving recipe: $e');
    rethrow;
  }
}
Future<void> removeSavedRecipe(int recipeId) async {
  try {
    await _apiService.removeSavedRecipe(recipeId);
    await fetchSavedRecipes(); // Refresh saved recipes
  } catch (e) {
    print('Error removing saved recipe: $e');
    rethrow;
  }
}
```

кінець додатку Б