Хмельницький національний університет
Факультет програмування
та комп'ютерних і телекомунікаційних систем
Кафедра інженерії програмного забезпечення

ДИПЛОМНИЙ ПРОЕКТ

Програмна система для автоматизації публікації,
поширення та пошуку кулінарних рецептів

Назва теми

Рівень вищої освіти Перший (бакалаврський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДПІПЗ.170101.01.01.ПЗ

Виконав студент IV курсу група ІПЗ-17-1 _____ (Підпис) В.О. Бойко (Ініціали, прізвище)

Керівник д-р фіз.-мат. наук, професор (Науковий ступінь, звання) _____ (Підпис) Л.П. Бедратюк (Ініціали, прізвище)

Нормоконтролер канд. техн. наук, доцент _____ (Підпис) Г. І. Радельчук (Ініціали, прізвище)

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ (Підпис) Л. П. Бедратюк (Ініціали, прізвище)

2 червня 2021 р.

Хмельницький 2021

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
05 02 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Бойку В'ячеславу Олександровичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Програмна система для автоматизації публікації поширення та пошуку кулінарних рецептів

Керівник проєкту (роботи) Бедратюк Леонід Петрович, д-р фіз.-мат. наук, професор
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 05.02.2021 р. № 11
2. Строк подання студентом проєкту (роботи) на кафедру 01.06.2021 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі, проектування програмного забезпечення, програмна реалізація, тестування програмного забезпечення
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали (слайди)

6 Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Радельчук Г. І., канд. техн. наук, доцент		
Антиплагіат	Гурман І. В., канд. техн. наук, доцент	1.06.2021	1.06.2021

7. Дата видачі завдання «05» лютого 2021 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1 Ознайомлення з тематикою дипломного проектування (ДП), визначення та узгодження індивідуальних там ДП	01.12 – 30.12.2020	
2 Дослідження предметної області, в якій планується використання програмного засобу (ПЗ), визначення задач та вимог, розробка технічного завдання	02.01 – 31.01.2021	
3 Проектування програмного забезпечення	01.02 – 28.02.2021	
4 Програмна реалізація	01.03 – 10.04.2021	
5 Тестування програмного забезпечення	11.04 – 30.04.2021	
6 Написання вступу, загальних висновків, оформлення джерел посилання та додатків. Оформлення пояснювальної записки ДП згідно вимог стандартів	01.05 – 25.05.2021	
7 Попередній захист ДП	Травень 2021 (згідно графіка)	
8 Перевірка ДП на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки)	26.05 – 30.05.2021	
9 Підготовка до захисту та захист ДП	з 01.06.2021	

Студент _____ В. О. Бойко
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

АНОТАЦІЯ

Тема дипломного проекту: «Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів».

Автор проекту: Бойко В'ячеслав Олександрович.

Керівник проекту: Бедратюк Леонід Петрович.

Пояснювальна записка: 86 с., 15 рис., 6 табл., 10 дод., 32 джерела.

Графічна частина: 15 презентаційних слайдів.

СОЦІАЛЬНА МЕРЕЖА, MVP-ПРОДУКТ, REST, WEB API, ASP.NET CORE, POSTGRESQL, REDIS, TYPESCRIPT, ANGULAR.

Метою проекту є розробка програмної системи, яка дозволяє автоматизувати публікацію та поширення кулінарних рецептів серед користувачів, а також забезпечує швидкий та зручний їх пошук та має сучасний інтуїтивно зрозумілий користувацький інтерфейс.

У дипломному проекті визначено специфіку соціальних мереж; виконано порівняння монолітної та мікросервісної архітектури; проведено аналіз методів та інструментів по управлінню високонавантаженими системами та визначені шляхи їх оптимізації; проаналізовано використання реляційних та NoSQL баз даних, встановлено їх переваги та недоліки; встановлено особливості фронт-енд фреймворків при реалізації клієнтської частини програми.

Для реалізації програмної системи використано серверну платформу ASP.NET Core та мову програмування C#, систему керування бази даних PostgreSQL, систему кешування Redis та фронтенд-фреймворк Angular.

В результаті проектування здійснена програмна реалізація системи для автоматизації публікації, поширення та пошуку кулінарних рецептів, а також проведено практичну апробацію отриманих результатів.

01.06.21.
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К-сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	ДППЗ.170101.01.01.ПЗ	Пояснювальна записка	86		
2	А4		Завдання на дипломний	1		
			проект			
3	А4		Анотація	1		
			Графічні документи			
4	А4		Презентаційні матеріали	15		

Змн.	Арк.	№ докум.	Підпис	Дата	ДППЗ.170101.01.01.ПЗ		
Виконав		Бойко В.О.		1.06	Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів Відомість документів	Літ.	Арк. 1 / Аркушів 1
Керівник		Бедратюк Л.П.		2.06			
Н. контр.		Радельчук Г. І.		2.06		ХНУ, ІПЗ-17-1	
Зав. каф.		Бедратюк Л.П.		2.06			

ЗМІСТ

ВСТУП

На сучасному етапі розвитку людства Інтернет виступає в ролі потужного інструменту з пошуку та надання інформації. Джерелами інформації можуть виступати звичайні веб-сайти, блоги, лендінг-сторінки тощо. Та найбільш популярними сервісами на сьогодні є соціальні мережі, де людина проводить більше часу, ніж на інших інтернет-ресурсах. Поява і розвиток соціальних мереж сприяє розвитку нової культури і усього суспільства. Такий спосіб комунікації виконує велику кількість функцій, дозволяє людині самореалізуватися, отримувати нову корисну інформацію.

Зараз соціальна мережа – це поєднання інформаційно-розважального контенту та реклами. Тепер користувач в основному не створює контент, а споживає його. З роками інформації стає все більше і вона починає об’єднуватися у категорії. Користувачам стає все складніше обирати із запропонованого контенту щось корисне та потрібне саме йому. Тому і починають набирати популярність соціальні мережі, діяльність яких обмежена певною тематикою. Користувачам пропонується тільки той контент, який вони бажають споживати. Хоча такі сервіси стають все популярнішими, вони не охоплюють деякі важливі та актуальні теми, наприклад, такі, як їжа та приготування страв. В інтернеті є безліч такого контенту, проте він зазвичай не структурований, що робить його не зовсім зручним для споживання.

Актуальність теми полягає у тому, що на сьогодні у мережі Інтернет є велика кількість кулінарних веб-сайтів, проте повноцінних соціальних мереж із такою тематикою практично не існує. Соціальна мережа – не лише інструмент обміну інформацією серед користувачів, а й готова платформа для ведення власного бізнесу. Крім того, приготування страв є одним із пріоритетних занять більшості людей. І якщо поєднати це – отримаємо потужну систему, призначену для пошуку та обміну кулінарних рецептів. Така система повинна бути у першу чергу швидка та оптимізована, а так як вона розрахована на велику кількість

користувачів – відповідно і високонавантажена. Технологій та практик для побудови і підтримки таких систем існує багато, проте вибрати найоптимальнішу із них – не завжди просто. Тому ми пропонуємо реалізувати таку програмну систему і на основі неї провести аналіз та дослідити основні принципи оптимізації високонавантажених систем.

Метою проекту є розробка програмної системи, яка дозволяє автоматизувати публікацію та поширення кулінарних рецептів серед користувачів, а також забезпечує швидкий та зручний їх пошук та має сучасний інтуїтивно зрозумілий користувацький інтерфейс.

Для реалізації проекту необхідно виконати наступні завдання:

– встановити особливості та визначити специфіку предметної області соціальної мережі;

– виконати порівняння монолітної та мікросервісної архітектури;

– провести аналіз методів та інструментів по управлінню високонавантаженими системами та шляхи їх оптимізації;

– проаналізувати використання реляційних, документо-орієнтованих та графових баз даних при реалізації програмної системи та реалізувати серверну частину програмної системи;

– встановити особливості фронт-енд фреймворків та реалізувати клієнтську частину веб-додатку;

– провести апробацію програмної системи, реалізувати тестування програмного забезпечення.

Результатом вирішення завдань дипломного проекту є повноцінна програм-на система для автоматизації, публікації та поширення кулінарних рецептів.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Соціальна мережа – це сервіс, що ґрунтується на об'єднанні людей зі спільними інтересами або діяльністю для обміну інформацією один з одним. Сьогодні користувачами соціальних мереж є сотні мільйонів людей по всьому світу. Люди проводять в них все більше часу: там вони спілкуються, шукають потрібну інформацію, радяться, знайомляться і встановлюють ділові відносини. Крім того, за допомогою сучасних соціальних мереж можна створювати рекламу. Більше того, із розвитком можливостей інтеграції рекламного контенту у соціальні мережі з’явився новий напрямок, що називається SMM (Social Media Marketing), який набирає обертів з кожним днем і є інструментом ведення бізнесу.

Розрізняють такі соціальні мережі:

– для спілкування, такі, як Facebook, Вконтакте;

– для обміну медіа контентом. Такі соцмережі дають широкі можливості для обміну відео і фото. До них відносяться, наприклад, Instagram та YouTube;

– для авторського запису. До даного виду відносяться сервіси для публікації текстово-медійного контенту. Наприклад, Twitter;

– для колективних переговорів. В основі цього виду соціальних мереж лежить потреба в обміні знань. Приклади: Reddit, Stack Exchange.

Особливими є тематичні соціальні мережі. Проблеми, що зазвичай вирішують такі сервіси обмежені певною тематикою. Наприклад, соціальна мережа Goodreads дозволяє користувачам шукати та публікувати книги, анотації, цитати та відгуки. Та хоч вони і популярні лише у конкретній сфері, вони повинні реалізовувати базовий функціонал соціальних мереж, бути зручними для користувача, а система повинна бути достатньо захищеною та продуктивною.

Схожими за реалізацією на тематичні соцмережі є кулінарні веб-сайти. Ця

тема є популярною, оскільки їжа – одна з основних потреб людини, а її приготування може витрачати багато часу. Проте зазвичай вони реалізовані у вигляді авторських блогів та звичайних сайтів, що відрізняються за своїм функціоналом та дизайном один від одного. Це не досить зручно для звичайних користувачів. Пошук конкретного підрозділу сайту, а відповідно і самого рецепту може займати більше часу.

Тому я пропоную реалізувати програмну систему для автоматизації публікації, поширення та пошуку кулінарних рецептів.

Для опису та формалізації основних бізнес-процесів програмної системи використано методологію функціонального моделювання IDEF-0. На рисунку 1.1. зображена контекстна діаграма роботи соціальної мережі. Вхідні дані – запити та дані користувачів, та дозвіл на редагування розділів програмної системи, таких як категорія рецепту, національна кухня, список інгредієнтів тощо. Відповідно, вихідними даним є створені користувачами публікації кулінарних рецептів, інформація про наявні рецепти та зареєстровані користувачі, що успішно авторизувались у системі. Механізмами на діаграмі є система управління та адміністратори та модератори.

На рисунку 1.2 зображена діаграма декомпозиції першого рівня, що розкриває основні функціональні особливості системи. Відповідно, можна виділити базовий функціонал програмної системи. Це редагування розділів адміністраторами та модераторами, перегляд публікацій, авторизація користувачів та створення рецептів.

Крім того, як і більшість соціальних мереж, дана програмна система також надає користувачам можливість оцінювати публікації інших користувачів, ділитися та коментувати їх, підписуватися на новини інших користувачів. Також, так як дана соціальна мережа є тематичною, було прийнято рішення реалізувати систему модерації контенту з метою уникнення шкідливих, нецензурних публікацій або публікацій, що не відповідають кулінарній тематиці.

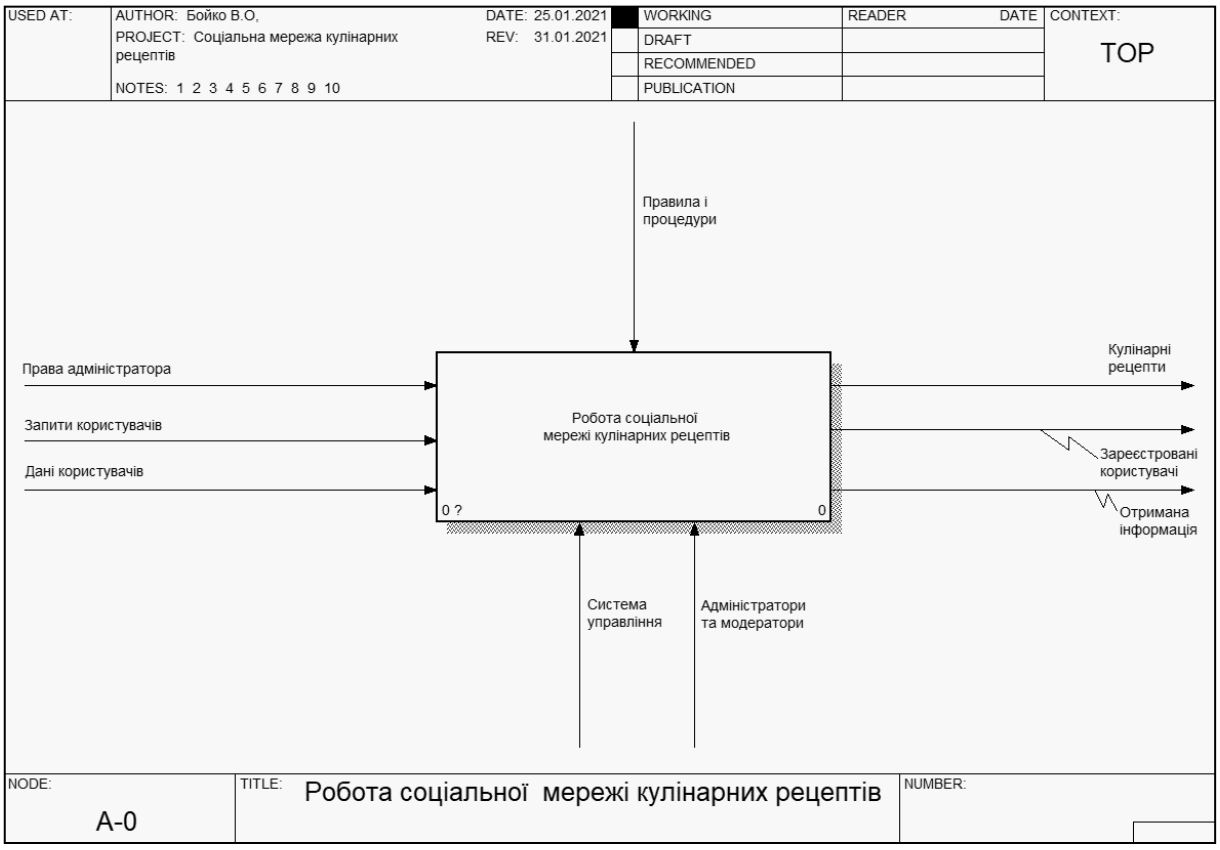


Рисунок 1.1 – Контекстна IDEF0-діаграма роботи програмної системи

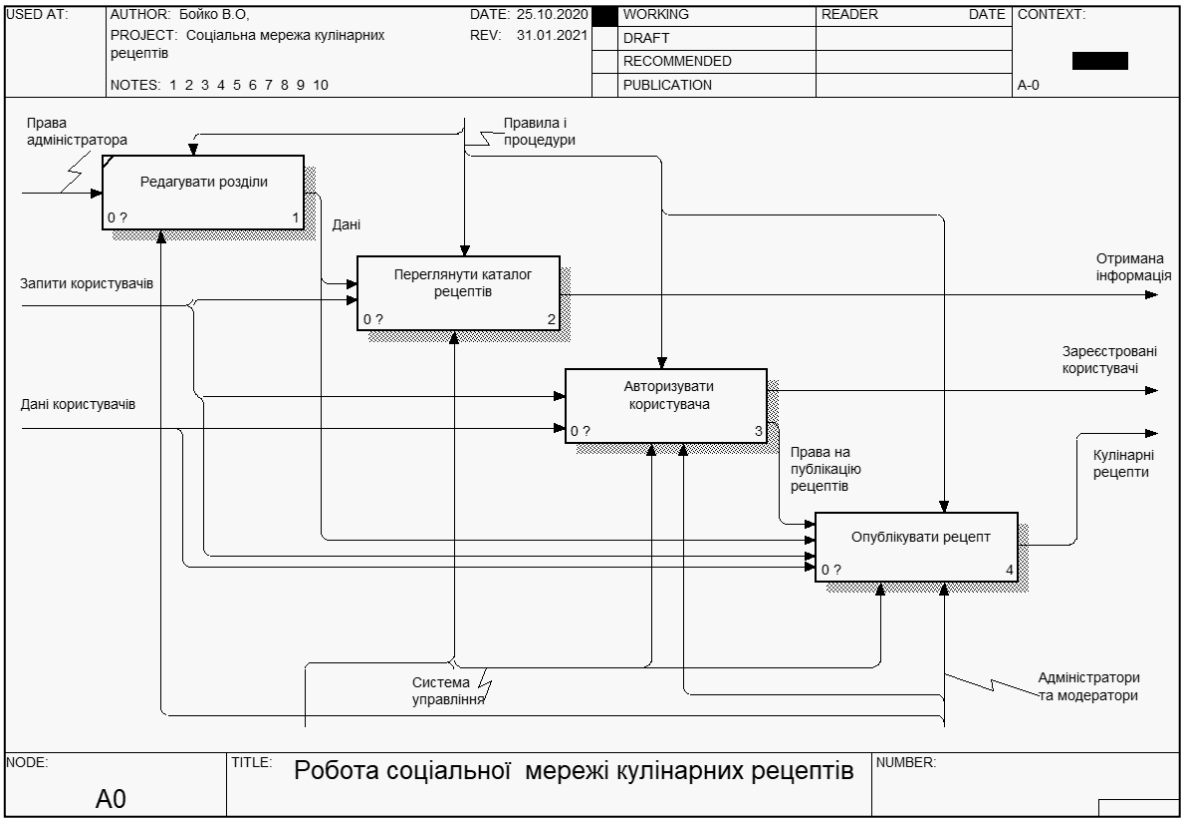


Рисунок 1.2 – IDEF0-діаграма декомпозиції першого рівня

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

На сьогодні є досить багато подібних програмних систем, що дозволяють користувачам переглядати та публікувати кулінарні рецепти. Тому важливо, щоб розроблювана програмна система була функціональною, зручною, продуктивною та привабливою для майбутніх користувачів.

Розглянемо деякі популярні веб-сайти кулінарних рецептів, та проведемо порівняльну характеристику. Основними елементами для оцінювання є: функціональність, продуктивність, зручність, зрозумілість та привабливість користувацького інтерфейсу веб-додатку.

На рисунку 1.3 зображено головну сторінку сайту [5]. Це один з популярних сайтів у сфері кулінарії. На сайті запропоновані різноманітні рецепти, які поділені на категорії (перші страви, гарніри, салати, випічки, закуски, десерти тощо). На цьому веб-сайті можна поділитись своїми кулінарними рецептами із іншими користувачами. Рецепти подані українською та російською мовами. Деякі із них мають фото. Також є примітивний функціонал оцінки кулінарних рецептів. Присутнє зручне навігаційне меню, а категорії, які найчастіше вибирають користувачі для перегляду відображаються на головній сторінці сайту. Є можливість пошуку рецептів по їх назві та опису.

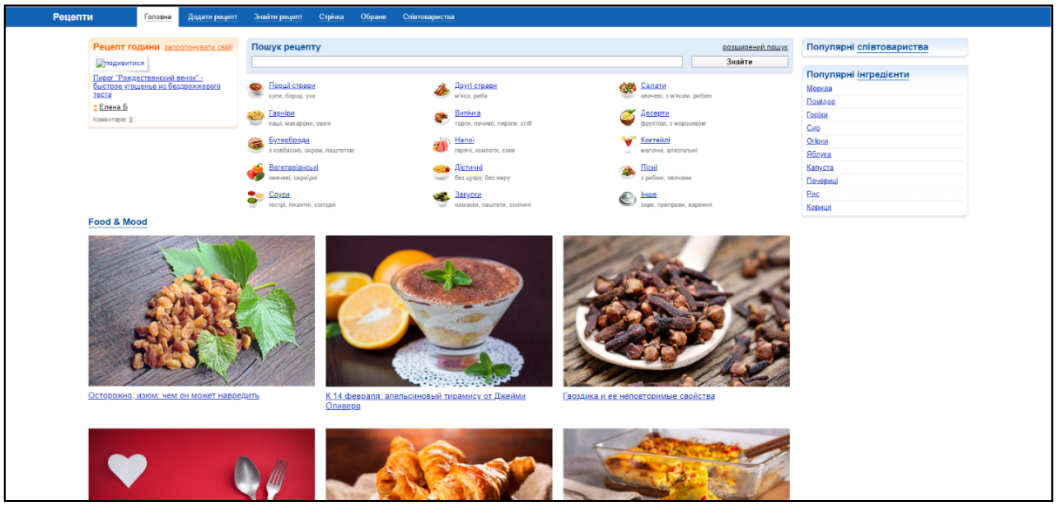


Рисунок 1.3 – Головна сторінка сайту [5]

Система працює швидко, функціонал інтуїтивно зрозумілий, проте дизайн є застарілим. Відсутній пошук по інгредієнтам та системи фільтрів. Реєстрація та авторизація відбувається через логін та пароль. Крім того стиль не зовсім відповідає стилю сучасних соціальних мереж, проте основний функціонал, що присутній у соціальних мережах, реалізовано.

На рисунку 1.4 зображено одну зі сторінок сайту «Simply Recipes» [6]. Це популярний кулінарний блог, написаний Еліза Бауер, яка регулярно публікує нові рецепти на веб-сайті. Інтерфейс виглядає мінімалістичним та сучасним. Є можливість збереження рецептів, пошуку та фільтрації. Усі рецепти зручно розподілені по категоріям, по яким також можна здійснювати пошук. Крім того, переглядати необхідні рецепти страв можна за допомогою пошуку по назві інгредієнтів. Наявні відео-рецепти та детальні інструкції з приготування. Проте немає можливості у фільтрах вказати декілька категорій для пошуку страв. Авторизуватися можна тільки через соцмережі, відповідно не має налаштування особистого кабінету. Також це не є соціальна мережа, а звичайний веб-сайт, блог, де немає можливості публікувати свої рецепти.

Рисунок 1.4 – Одна зі сторінок сайту «Simply Recipes»

На рисунку 1.5 зображено сторінку веб-додатку «Merry Kitchen» [7]. Цей сайт позиціонує себе як соціальна мережа кулінарних рецептів. Працює швидко, інтерфейс функціонально зручний, є поле пошуку рецептів по назві, присутній пошук по інгредієнтам, є усі необхідні фільтри. Вибрати рецепти можна відразу за декількома категоріями. Є можливість оцінки та коментування рецептів інших користувачів, а також їх збереження. Реалізована система push-сповіщень та присутня модерація.

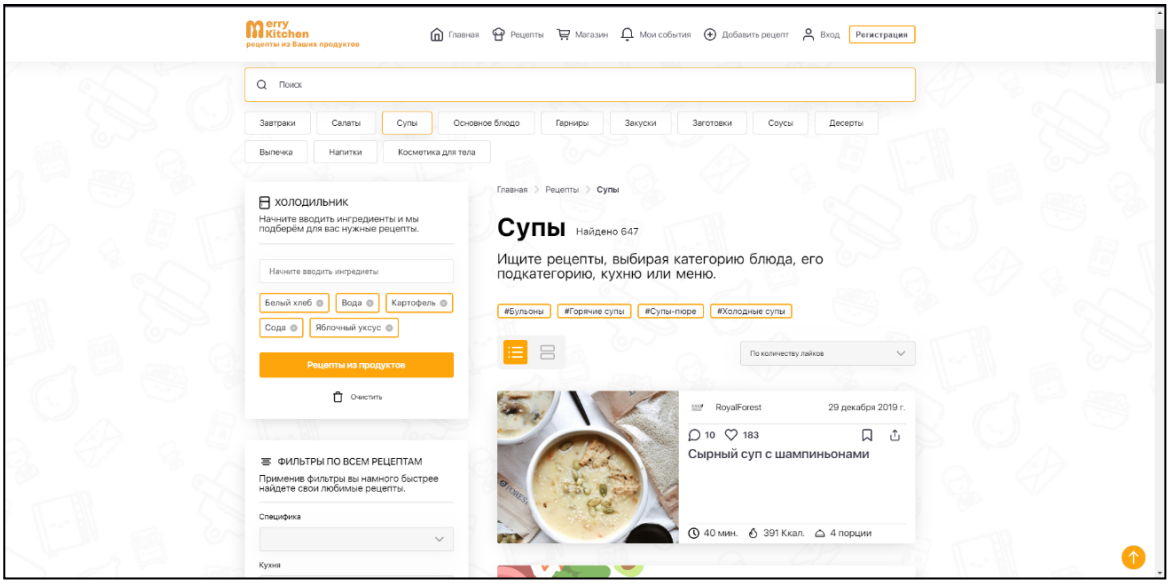


Рисунок 1.5 – Розділ «Супи» веб-додатку «Merry Kitchen»

Інтерфейс веб-додатку сучасний, мінімалістичний, зручний та зрозумілий та функціонально подібний до більшості популярних соціальних мереж. Популярні категорії рецептів виведені на головний екран для зручності пошуку. Є окремі додатки під систему IOS та Android.

Зареєструватися та авторизуватися можна через електронну пошту або використовуючи інші соцмережі. Також є можливість редагувати дані облікового запису користувача, проте усі ці діє можна виконувати, якщо додаток на мобільний пристрій. Веб-версія має обмежений функціонал.

На основі отриманих даних складено порівняльну таблицю наявного програмного забезпечення та подано у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика наявного ПЗ

Назва або посилання	Можливість публікації рецептів	Зручний пошук рецептів	Дизайн користувацького інтерфейсу	Реєстрація та функціонал особистого кабінету
http://cook.i.ua/	Так	Ні	Застарілий, мінімалістичний, інтуїтивно зрозумілий	Через логін та пароль, є редагування особистого кабінету
«Simply Recipes»	Ні	В цілому зручний	Сучасний, інтуїтивно зрозумілий	Тільки через соцмережі, немає можливості редагування особистих даних
«Merry Kitchen»	Так	Так	Сучасний, інтуїтивно зрозумілий, мінімалістичний	Через електронну пошту або інші соцмережі, є можливість налаштування особистого кабінету

Проведений аналіз аналогічного програмного забезпечення показав, що розроблювана програмна система повинна мати:

– можливість публікації та редагування рецептів;
– зручний пошук рецептів, їх фільтрацію по категоріями;
– сучасний, інтуїтивно зрозумілий дизайн інтерфейсу;
– функціонал, подібний до функціоналу сучасних соцмереж;
– зручну форму авторизації та реєстрації;
– налаштування особистого кабінету.

1.3 Визначення вимог до програмного забезпечення та розробка технічного завдання

Для визначення вимог до програмного забезпечення використовується UML – мова моделювання, яка призначена для побудови діаграм, компоненти яких представляють об’єкти програмного забезпечення та документують і

відображають вимоги до ПЗ. Ця мова є загальноприйнятим стандартом для графічного опису програмного забезпечення. Уніфікована мова моделювання допомагає наочно переглядати компонування системи і полегшує співпрацю з іншими її розробниками.

Діаграма варіантів використання – це загальне представлення функцій системи. Вона описує залежності і взаємозв'язки між варіантами використання і акторами. Ця діаграма показує, що має робити система. У таблиці 1.2 подано опис акторів програмної системи.

Таблиця 1.2 – Опис акторів програмної системи

Актор	Короткий опис
Незареєстрований користувач (гість)	Має можливість переглядати та шукати кулінарні рецепти.
Зареєстрований користувач	Додатково до можливостей незареєстрованого користувача, має можливість створювати та редагувати свої публікації, ставити оцінки, додавати у збережене, коментувати публікації інших людей, підписуватися на інших користувачів, керувати своїм обліковим записом.
Адміністратор та модератор	Може додавати нові розділи у програмній системі, виконувати модерацію користувацького контенту, тобто перевіряти публікації користувачів на відповідність цензурі та іншим політикам популярних соціальних мереж блокувати та видаляти користувацькі облікові записи.

Як видно із таблиці 1.2, у системі існує три головних актори. Це гість, зареєстрований користувач та модератор. Усі вони взаємодіють на різних рівнях та мають різні права, що обмежуються їхніми ролями.

У таблиці 1.3 описано основні варіанти використання програмної системи, що відповідають прецедентам на відповідній діаграмі.

Таблиця 1.3 – Опис варіантів використання програмної системи

Актор	Найменування ВВ	Опис ВВ
1	2	3
Незареєстрований користувач	Реєстрація в системі	Користувач може зареєструватися в системі за допомогою своєї електронної пошти та паролю
	Пошук та перегляд інформації	Користувач може шукати та переглядати публікації
Зареєстрований користувач	Керування обліковим записом	Користувач може редагувати облікові дані особистого кабінету
	Авторизація в системі	Користувач може авторизувати в системі, використовуючи свою електронну пошту
	Публікація рецептів та керування ними	Користувач може публікувати свої рецепти, редагувати та видаляти опубліковані рецепти.
Зареєстрований користувач	Оцінювання, коментування, додавання в збережені	Користувач може оцінювати рецепти інших користувачів, коментувати їх та зберігати у свою книгу рецептів
	Створення та видалення підписок на інших користувачів	Користувач може шукати та підписуватися на інших користувачів, а також скасовувати підписки
Користувач	Пошук та перегляд інформації	Користувач може шукати, переглядати, фільтрувати та сортувати публікації кулінарних рецептів

Кінець таблиці 1.3

1	2	3
Адміністратор та модератор	Керування розділами системи	Адміністратор може додавати, редагувати і видаляти розділи програмної системи
Адміністратор та модератор	Модерація користувацького контенту	Адміністратор може не допустити пост до публікації у випадку нецензурного контенту, контенту, що належить іншим користувачам або контенту, що не відповідає кулінарній тематиці
	Блокування та видалення користувацького облікового запису	У випадку підозрілих дій з боку користувацького облікового запису адміністратор може заблокувати акаунт на певний термін або видалити його

Відповідно до описаних акторів та варіантів використання побудовано діаграму (рисунок А.1) та створено технічне завдання (додаток Б).

Отже, було проведено аналіз предметної області та встановлено, що соціальні мережі – популярні веб-сервіси, що відіграють значну роль у сучасному житті людини, забезпечують комунікацію між людьми, виступають у якості основи для ведення бізнесу та об’єднують людей за спільними інтересами. Усі вони повинні бути продуктивними, захищеними та легкими у користуванні.

Також було проведено аналіз існуючого програмного та інформаційного забезпечення предметної області, в результаті якого визначено, що існує багато подібних програмних систем і додатків, усі вони відрізняються своїм зовнішнім виглядом, функціональністю та зручністю у користуванні.

Визначені функціональні та нефункціональні вимоги до програмного забезпечення та виконано опис основних прецедентів системи.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Архітектурне проектування

2.1.1 Аналіз клієнт-серверної архітектури сучасних веб-сервісів

Усі веб-додатки та будь-які Інтернет-сервіси, побудовані за принципом клієнт-серверної архітектури. В основі такої архітектури лежать два основних компоненти: клієнт та сервер. Під клієнтом зазвичай розуміється програмне забезпечення, що має здатність надсилати запити на віддалений комп'ютер – сервер. Також клієнтом можна вважати користувача, якому необхідно за допомогою відповідного програмного та апаратного забезпечення отримати доступ до інформації. Сервер – більш потужне апаратне забезпечення, що призначене для вирішення великого роду задач, наприклад, доступу до даних, що запитує клієнт, виконання складних обчислень тощо. Проте задача серверу одна – належно обробити дані користувача та надіслати коректну відповідь.

Отже, архітектура «Клієнт-сервер» полягає у тому, що клієнт відправляє запит на сервер, де він обробляється, і готовий результат відправляється клієнту. Сервер може обслуговувати декілька клієнтів одночасно. Якщо одночасно надсилається більше одного запиту, вони встановлюються у чергу і виконуються сервером послідовно.

Для того, щоб сервер та клієнт мали здатність коректно розпізнавати вхідні та формувати вихідні дані, комунікація між ними відбувається за допомогою протоколу передачі даних – HTTP (Hyper Text Transfer Protocol). Це протокол передачі гіпертексту, тобто HTML (Hyper Text Markup Language). Проте, на сьогодні, у зв'язку зі стрімким розвитком серверних API – наборів чітко визначених методів для взаємодії компонентів серверу – все більше використовують дані формату відмінного від гіпертексту. Тому можна стверджувати, що HTTP – протокол передачі даних у форматі, що строго визначений прикладним програмним інтерфейсом серверу. Внаслідок популярності серверних API з'явились та розвиваються різні архітектурні підходи

для створення прикладних програмних інтерфейсів та керування ними. Найпопулярніший із них – REST (Representational State Transfer) – передача стану представлення. Відмінною особливістю сервісів REST є те, що вони дозволяють найкращим чином використовувати протокол HTTP. Обробка запитів при такій архітектурі має виконуватись у відповідності до того, яким методом були надіслані дані. Метод запиту вказує, яку дію необхідно виконати над надісланими даними (створити, оновити, видалити тощо). Наприклад, метод отримання даних не повинен виконувати додаткові дії, такі як, наприклад, оновлення чи видалення. Він призначений лише для читання ресурсу, а не його зміни. Також вимогою RESTful архітектури є надсилання коректних статус-кодів користувачу у відповідь. Наприклад, якщо ресурс було створено успішно, необхідно надіслати код «201 Created», якщо ресурс не знайдено – «404 Not Found» тощо. REST не накладає жодних обмежень на формати представлення ресурсів.

Отже, для реалізації мети даного дипломного проекту було обрано клієнт-серверну архітектуру як основу для побудови взаємодії. Також запропоновано використати деякі методики архітектурного шаблону REST, що можуть покращити використання протоколу HTTP.

2.1.2 Аналіз підходів до проектування серверної архітектури

Існують різні підходи до реалізації серверної архітектури у сучасних веб-додатках. Проте виділяють основні та найбільш поширені два архітектурних стилі: монолітний та мікросервісний. Усі вони мають свої переваги та недоліки.

Монолітна архітектура розглядається як традиційний метод розробки веб-додатків. Додаток на основі такої архітектурі розробляється як єдиний пакет, що містить усю необхідну функціональність виділену в окремі модулі, що взаємодіють між собою у рамках одного рішення. Ці модулі і складають багаторівневу архітектуру у монолітному сервісі. Її стандартна структура складається із чотирьох основних рівнів.

Рівень бізнес-логіки – відповідає за роботу та спосіб організації моделей проекту, що описують сутності відносно предметної області. Зазвичай це може бути сукупність класів, що поєднані між собою або ієрархічно, або за допомогою інших відношень, таких як агрегація чи композиція. Модель повинна коректно описувати сутності у відповідності до їх реального призначення. Рівень бізнес логіки – є одним із важливих у роботі моноліту.

Не менш важливий – рівень доступу до сховища даних. На цьому ріні відбувається робота із базою даних та реалізуються основні операції над ними. Їх скорочено називають CRUD. Цей термін відбиває призначення кожної із операції: Create – створити нові дані, Read – отримати інформацію, Update – оновити дані, Delete – їх видалити.

Рівень представлення – графічний інтерфейс користувача, який обробляє запити протоколу HTTP. Це може бути статично або динамічно згенеровані набори веб-сторінок, що представляють різні розділи веб-додатку. Графічний інтерфейс скорочено прийнято називати UI (User Interface).

Рівень інтеграції не є обов’язковим у реалізації повноцінної монолітної архітектури, проте він є дуже корисним при інтеграції з іншими програмними системами. На цьому рівні надається загальний інтерфейс для керування даними інтегрованих веб-сервісів. Зазвичай це відбувається за допомогою різних адаптерів, що трансформують представлення даних інтегрованого сервісу у представлення основного і навпаки.

Незважаючи на те, що монолітна архітектура має логічну багаторівневу архітектуру, кінцеві додатки будуть упаковані в один моноліт. Великою перевагою моноліту є те, що його легше реалізувати. У такій архітектурі можна швидко почати реалізовувати бізнес-логіку.

Ще однією важливою перевагою моноліту є незначна кількість наскрізних проблем. Більшість додатків залежать від міжкомпонентних задач, наприклад, таких, як ведення логів та контрольних журналів. Монолітні додатки набагато легше вирішують такі задачі завдяки єдиній кодовій базі. Тому такі наскрізні тести (end-to-end) виконати просто. Також важливо зазначити, що моноліт

простий у розгортанні та легко масштабується за рахунок єдиної кодової бази.

Проте є і недоліки. Одним з основних є те, що з часом кодова база розростається та стає все складніше знаходити побічні ефекти та залежності, які можуть суттєво вплинути на терміни розробки. Із розростанням кодової бази поступово втрачається гнучкість, тобто для кожного невеликого оновлення необхідне повторне розгортання. Також стає все важче інтегрувати нові технології, оскільки всі модулі у такій архітектурі міцно зв’язані один з одним і розрив цих зв’язків може коштувати новими потенційними проблемами та дефектами, що в свою чергу змушує розробників писати все більше тестів та більше витрачати часу саме на тестування замість розробки.

Інший стиль проектування серверної архітектури – мікросервіси. Це варіант сервіс-орієнтованої архітектури (SOA), у якій слабо пов'язані ізольовані компоненти (сервіси) взаємодіють один з одним для спільного виконання завдань, проте кожен сервіс у такій системі повинен мати можливість самостійного виконання задач., які надають різні послуги та взаємодіють один з одним за допомогою методів прикладного програмного інтерфейсу (API). У порівнянні з монолітом, у мікросервісах є декілька одиниць розгортання, тобто кожен сервіс розгортається самостійно.

Сервісний модуль у такій архітектурі фактично представляє невеликий моноліт, що також має багаторівневу архітектуру представлену чотирма основними рівнями. Тут рівень інтеграції є обов’язковим та вже більше відіграє роль ніж у моноліті, оскільки зазвичай сервіси комунікують між собою. Крім того, вони можуть бути реалізовані за допомогою абсолютно різних технологій, сховищ даних, шаблонів проектування тощо.

Одною з переваг мікросервісів є те, що вони невеликих розмірів. Це допомагає зменшити час компіляції, запуску та час тестування, оскільки усі ці фактори впливають на продуктивність та тривалість розробки. Мікросервіси не прив'язані до технології, яка використовується у інших моудлях. Це означає, що можна використовувати кращі та сучасніші технології, тому старі сервіси можна швидко та легко переписати з використанням нових технологій.

Так як мікросервіси – це зазвичай окремі невеликі модулі, тестувати їх набагато зручніше, ніж моноліт. Крім того, якщо виникнуть якісь дефекти, то вони залишаться лише у межах одного мікросервісу. Більшість інших сервісів продовжать працювати без помилок. Тобто мікросервіси є відмовостійкими, що також є великою їх перевагою.

Проте є і недоліки. Найчастіше це пов’язано із транзакціями. Транзакція є логічною одиницею роботи з даними. Вона може бути виконана або цілком і успішно, при цьому дотримуючись цілісності даних, або не виконана зовсім. У разі виникнення помилок транзакція не справляє ніякого ефекту. У мікросервісах здатність підтримувати цілісність даних між різними модулями – не проста задача, тому транзакції легше виконувати у моноліті. Рішення для мікросервісів – використання дистрибутивних (розподілених) транзакцій. Це досить громіздке, проте дієве рішення. За рахунок громіздкості, продуктивність мікросервісів може знизитись, на відміну від моноліту.

Ще один недолік мікросервісів – складність їх тестування. Вона виражається у тому, що спочатку потрібно окремо проводити тестування з кожним сервісом окремо, а потім тестувати коректну взаємодію між ними.

Підсумовуючи вищесказане, можна виділити основні особливості стилів побудови серверної архітектури.

Отже, монолітна архітектура краща для розробки невеликих, простих і легких додатків. Мікросервісна ж архітектура у свою чергу призначена для розробки складних та розподілених систем. Тому, порівнюючи ці два стилі, не можна однозначно стверджувати, що один краща за інший. Стиль архітектури необхідно обирати, керуючись досвідом, термінами розробки та складністю програмної системи.

Соціальні мережі – зазвичай великі та комплексні системи, що складаються із різних сервісів та компонентів. Монолітний стиль для таких систем не зовсім вдалий вибір, оскільки із нарощуванням функціоналу буде збільшуватися кодова база, створюватися нові зв’язки та, взагалі, програмну систему буде складно підтримувати, тестувати та виконувати її масштабування і розгортання. Проте у

більшості випадках новий продукт на ринку не може бути відразу реалізований повністю. Такі додатки мають назву MVP (Minimum viable product) – мінімально життєздатний продукт – продукт із мінімальним функціоналом, який може задовольнити потреби потенційних користувачів. Такі продукти розробляються ітеративним методом, тобто функціонал нарощується поступово у вигляді оновлень. Тому для таких додатків було обрано монолітний стиль серверної архітектури з огляду на терміни розробки та не високу архітектурну складність системи. Крім того декомпозувати невелику систему на мікросервіси – це затратно та не вигідно для впровадження, розгортання і підтримки.

2.2 Детальне проектування

Для реалізації програмної системи існує два основних типи веб-служб: Process-Based – на основі процесів та Event-Based – на основі подій.

У веб-службах на основі процесів кожен запит обробляється в окремому потоці або процесі. Кожен потік або процес вимагає певної кількості серверних ресурсів (апаратних засобів). Ресурси сервера простоюють та не використовуються до моменту відкладеного запиту і відправки клієнту відповіді. Це негативно позначається на продуктивності при високих навантаженнях, коли обраного потоку процесів не вистачає для обробки всіх запитів. До цієї категорії відносяться такі веб-служби, як IIS та Apache. Це не зовсім підходить для роботи таких систем, як соціальні мережі.

Веб-служби на основі подій використовують всі ресурси серверного обладнання. «Loop-event-endless» цикл відстежує статус запиту від клієнта. Тоді використовуються усі ресурси веб-сервера, що дозволяє обробляти запити максимально швидко, а у випадках затримок – працювати з іншими запитами із черги подій (Event Queue) тобто асинхронно. До таких веб-служб належать платформи Node.js та Nginx. Такі веб-сервери є вдалим вибором для розробки високонавантажених систем, тому для реалізації даної програмної системи було

обрано веб-службу Nginx.

Із огляду на зручність, постійний розвиток, хорошу підтримку компанією Microsoft та велику різноманітність сторонніх бібліотек, фреймворків та модулів було обрано об’єктно-орієнтовану статично-типізовану мову програмування C# та ASP.NET Core – крос-платформний фреймворк для створення веб-додатків.

Серверна частина програмної системи побудована на основі видозміненого шаблону проектування MVC (Model-View-Controller), де замість представлення (View) використовуються повноцінні фронтенд-системи, що розгортаються як окрема одиниця та відповідають лише за відображення користувацького інтерфейсу. Вони можуть бути ніяк не зв’язані із сервісною системою (яку ще називають Web API), а можуть розгортатися, використовуючи її можливості.

Відповідно до монолітного архітектурного стилю, програмна система складається із трьох взаємопов’язаних слоїв: Data Layer, Service Layer і Presentation Layer.

Data Layer – шар представлення даних. Він відповідає за впорядкованість інформації та управління базами даних на рівні запитів до них. В основному класи, що знаходяться на такому рівні називаються контекстами даних. Також на цьому рівні містяться основні моделі програмної системи, що описують предметну область.

Service Layer – рівень абстракції програми, на якому містяться модулі керування шаром даних, що реалізовуються у вигляді основних операцій роботи з даними – CRUD. Шар сервісів використовує патерни проектування «Адаптер» та «Команда» для з'єднання із наступним, презентаційним рівнем для передачі оброблених даних у контролери.

Presentation Layer – найвищий рівень монолітної системи. Він обмінюється даними із сервісним шаром та використовує їх для відображення користувачу або за допомогою генерації веб-сторінок, або надсилає запити клієнту у REST-стилі, виступаючи при цьому повноцінним прикладним програмним інтерфейсом (Web API). Схема трьох-шарової монолітної архітектури подана на рисунку 2.1.

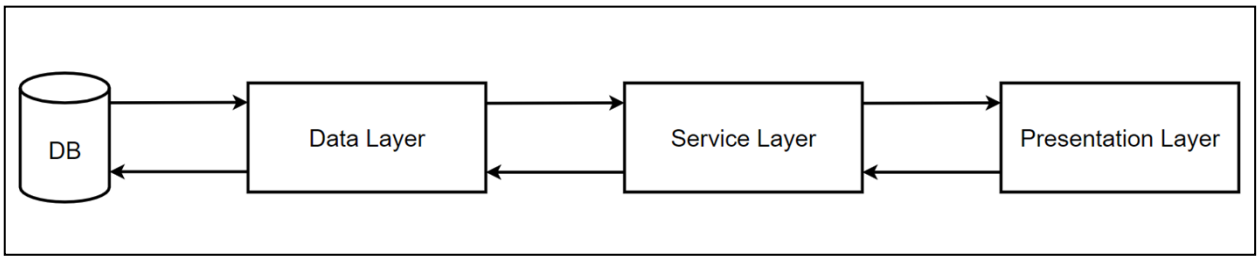


Рисунок 2.1 – Монолітна архітектура програмної системи

Кожен шар представляє собою набір класів, що виконують певні функції відповідно до рівня розташування. Усі ці класу комунікують між собою, передаючи дані.

2.2.1. Аналіз та вибір типу бази даних

Усі веб-додатки працюють з інформацією, яку треба не тільки коректно обробити, а і надійно зберегти. Для цього використовуються різні системи для збереження даних, що називаються базами даних. Одна з основних проблем, яку необхідно вирішити при проектуванні програмної системи – це правильний та зважений вибір сховища даних, адже від цього залежить швидкість роботи програми, терміни розробки, способи масштабування, підтримки тощо.

На сьогодні існує досить багато систем керування даними. Усі вони поділяються на три великі групи: реляційні бази даних, «NoSQL» та комбіновані програмні рішення.

Реляційні бази даних. Цей тип є досить поширеним завдяки успішним реалізаціям реляційних моделей у системах керування базами даних. Збережена інформація структурується як набір пов'язаних таблиць, кожна з яких містить інформацію про об'єкти певного типу. Рядок таблиці містить дані про один об'єкт, а стовпці таблиці описують різні характеристики цих об'єктів – атрибутів. Записи мають однакову структуру – вони складаються з полів, що зберігають атрибути об'єкта. Кожне поле, описує тільки одну характеристику об'єкта і має строго визначений тип даних. Усі записи мають одні і ті ж поля, тільки у них

відображаються різні інформаційні властивості об'єкта.

У реляційній базі даних кожна таблиця повинна мати первинний ключ – поле або комбінацію полів, які єдиним чином ідентифікують кожен рядок реляційної таблиці.

Одна з основних переваг реляційних баз даних полягає у тому, що вони дозволяють користувачам легко класифікувати та зберігати дані, які згодом можуть бути певним чином оброблені для отримання конкретної інформації. Реляційні бази даних також легко розширюються і не залежать від фізичної організації. Зазвичай для доступу до реляційної бази даних використовуються спеціалізовані програми, що називаються системами керування базами даних (СКБД). Це ще одна перевага такого типу баз даних, оскільки СКБД в основному мають дуже широкий функціонал, та забезпечують базу даних додатковими корисними функціями, наприклад, такими, як індекси, очищення таблиць, підтримка JSON-у та взагалі мають свої ідеології упорядкування даних, що називаються рушіями БД. Також, реляційні БД мають чітко визначену структуру, тобто її проектування відбувається завчасно, що у майбутньому запобігає невизначеності у прийнятті архітектурних рішень та надає повне уявлення про сутності предметної області та їх взаємодію.

Також, відповідності реляційних баз даних вимогам ACID (Atomicity, Consistency, Isolation, Durability – атомарність, цілісніть, ізольованість, довговічність) дозволяє зменшити ймовірність несподіваної поведінки системи та забезпечити її цілісність.

Проте, з іншого боку, реляційні бази даних можуть ускладнюватись зі збільшенням обсягу даних, що, як наслідок, впливає на продуктивність програми.

Так як ПЗ для автоматизації публікації, поширення та пошуку кулінарних рецептів має чітко визначені сутності предметної області та є MVP-продуктом, тому оптимальне рішення – використання реляційної бази даних, як основного сховища системи.

Бази даних NoSQL – група типів БД, що пропонують підходи, відмінні від стандартного реляційного шаблону. Такі бази спеціально створені для певних

моделей даних і володіють гнучкими схемами, що дозволяє швидко розробляти сучасні програми. Бази даних NoSQL набули широкого поширення у зв'язку з простотою розробки, функціональністю і продуктивністю при будь-яких масштабах. Проте за рахунок гнучкості та високої продуктивності нереляційні бази даних втрачають стовідсоткову цілісність даних, тому їх рекомендовано використовувати як додаткове сховище даних до реляційних БД. Такий спосіб використання не тільки зберігає усю цілісність даних, надану РБД, а ще й може підвищити продуктивність системи. Часто NoSQL бази даних, а саме прості за структурою сховища «ключ-значення» використовують у вигляді шару кешування додатку.

До сховищ NoSQL відносяться ще й графові БД. У таких сховищах використовуються вузли для зберігання сутностей; і ребра – для зберігання взаємозв'язків між сутностями. Ребро завжди має початкову і кінцеву вершину, тип та напрямок. Обмеження на кількість і тип взаємозв'язків, які може мати вершина, відсутні.

Обхід графа у графовій базі даних можна виконувати або за певними типами ребер, або по всьому графу. Він виконується досить швидко, оскільки взаємозв'язки між вершинами не обраховуються під час виконання запиту, а зберігаються у базі даних. Графові бази даних мають ряд переваг у таких системах, як сервіси рекомендацій, системи виявлення шахрайства – тоді, коли потрібно створювати взаємозв'язки між даними і швидко їх обробляти. Проте, графові БД важкі у масштабуванні, оскільки спроектовані відповідно до однорівневої архітектури. Така архітектура передбачає, що клієнт, сервер і база даних розташовані на одній машині. Приклади графових БД – це Neo4J, OrientDb, Neptune, ArangoDb тощо.

Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів є соціальною мережею, а отже, зв’язки типу «користувач – підписники» і подібні було б краще зберігати у вигляді зв’язаного орієнтованого графу. Проте більшість функціоналу системи не передбачає використання даних, що ефективно будуть оброблені саме за допомогою їх упакування у граф або

дерево. Крім того використання ще одного типу бази даних значно ускладнить систему, її підтримку та збільшить терміни розробки, що є не зовсім доцільно для програмного продукту типу MVP.

Документні бази даних спільно використовують базову семантику доступу і пошуку сховищ ключів та значень. Такі БД також використовують ключ для унікальної ідентифікації даних. Документо-орієнтовані бази зберігають дані у структурованих форматах – JSON, BSON або XML. Вони не мають фіксованої та чітко визначеної структури даних. Вони є вдалим вибором для швидкої розробки невеликих додатків, у яких схема даних може часто змінюватись.

Ще однією перевагою є горизонтальне масштабування таких баз даних, тобто додавання фізичних серверів та розподілення даних між ними. Один із недоліків полягає у тому, що оновлення даних – повільний процес у документних БД, оскільки дані можуть дублюватися і бути розподіленими між декількома серверами. Також відсутня підтримка атомарних транзакцій, що відповідають за цілісність даних. Приклади таких баз даних: MongoDb, CouchDb, MarkLogic.

Бази даних колонного типу також належать до сімейства NoSQL, але ззовні схожі на реляційні. Як і реляційні, подібні сховища зберігають дані, використовуючи рядки і стовпці, але із іншим зв'язком між елементами. У реляційних БД всі рядки повинні відповідати фіксованій схемі, що визначає, які стовпці будуть у таблиці, типи даних та інші критерії. У базах колонного типу замість таблиць є структури – «колонне сімейство». Сімейства містять рядки, кожен з яких визначає свій власний формат. Рядок складається із унікального ідентифікатора, що використовується для пошуку, за яким слідують набори імен та значень стовпців.

Такі бази даних є надзвичайно продуктивними у записі даних. Усі таблиці у таких БД є не фіксованими та не визначеними завчасно. Вони проектуються відносно запитів і дублювання даних або ж денормалізація – це природньо для таких сховищ даних. Використовується, коли необхідно постійно записувати нову інформацію набагато частіше, ніж виконувати запити читання, наприклад для зчитування показу датчиків у реальному часі.

Проте, для програмної системи, яку реалізовано у дипломному проекті використання колонних баз є не зовсім вдалий вибір, оскільки у системі присутня велика кількість запитів фільтрування та сортування, що надзвичайно складно досягти при використанні таких БД. Крім того, як недолік таких баз – це те, що при видалені дані фактично не видаляються, а «затираються», утворюючи блоки, що можуть нагромаджуватись із часом і для їх очищення необхідні додаткові операції, що означає збільшення складності підтримки такої програмної системи, що є недоцільно для MVP-продукту. Приклади колонних БД: Cassandra, ScyllaDb.

Отже, розглянувши та порівнявши основні типи бази даних було зроблено висновок, що для реалізації дипломного проекту було обрано реляційні бази даних у сукупності із сховищами «ключ-значення», що виконують функції кешування даних із метою пришвидшення роботи системи та усунення одного із недоліків ACID-системи – періодичних затримок у обробці даних.

2.2.2. Проектування логічної моделі «Сутність-зв’язок»

Модель «Сутність-зв’язок» відбиває концептуальну модель світу, яка буде представлена у базі даних. Вона заснована на сприйнятті реального світу, який складається із сукупності об'єктів, що називаються сутностями, та взаємозв'язків між цими об'єктами. Модель ER досягає високого ступеня незалежності даних. Схема бази даних у ER-моделі може бути спроектована у вигляді діаграми «Сутність-зв’язок».

Основними будівельними блоками у ER-моделі є:

– Entity – сутність, що є проекцією об’єкту із предметної області;

– Attribute – атрибут, це властивості сутностей, тобто те, що їх описує;

– Relationship – відношення між сутностями.

Відношення у ER-моделі – це один із найважливіших компонентів, що застосовується для опису зв'язків між об’єктами. Існує три основних типи відношень між сутностями.

«Один-до-багатьох» – відношення, що асоціюють одну сутність із кількома залежними. Прикладом такого відношення – користувачі, що мають опубліковані пости. Для одного користувача може існувати кілька опублікованих ним постів, отже це приклад відношення.

«Один-до-одного» – це окремий випадок відношень «один-до-багатьох». Наприклад, користувач що має лише один набір налаштувань програми і так кожен користувач відповідно, має лише свій власний набір налаштувань. Такі відношення використовуються для розділення однієї великої сутності у якості винесення атрибутів і утворюючи при цьому окрему сутність. Таким чином утвориться два невеликих об’єкта, зв’язані відношенням «Один-до-одного».

«Багато-до-багатьох» – відношення, що асоціюють декілька сутностей із декількома залежними сутностями. Наприклад, це може бути відношення між користувачами і підписниками. Один може мати декілька підписників і один підписник може бути підписаний на декількох користувачів. Зазвичай такі відношення не існують у чистому вигляді, а розділяються двома «Один-до-багатьох» та «Багато-до-одного».

Побудова коректної ER-моделі і, відповідно, проектування реляційної моделі бази даних вимагає розбиття одного відношення, та формування декількох відношень на базі функціональної залежності. Це називається нормалізацією даних. Нормалізація дозволяє зменшити кількість повторюваних даних, покращити загальну організацію бази даних, таким чином досягається гнучкість архітектури та найголовніше – забезпечити узгодженість даних.

Існують три основні нормальні форми. При першій НФ відбувається виключення повторюваних груп в окремих сутностях, створюються нові сутності для кожного набору зв’язаних даних та кожен набір зв’язаних даних ідентифікується за допомогою первинного ключа. Первинний ключ (Primary Key, PK) – це один атрибут, або набір атрибутів, що однозначно ідентифікує кортеж (рядок) відношення, тобто є унікальним.

При другій НФ створюються окремі сутності для наборів значень, які застосовуються до кількох записів. Ці сутності зв’язуються за допомогою

зовнішнього ключа. Зовнішній ключ (Foreign Key, FK) – це атрибут у відношенні, який відповідає первинному ключу іншого або цього ж відношення. При необхідності записи не повинні залежати від первинного ключа таблиці.

При третій НФ виключаються ті поля, що не залежать від ключа. Зазвичай третьої нормальної форми достатньо для коректної побудови логічної моделі.

На початковому етапі проектування логічної моделі програмної системи при аналізі предметної області було виділено декілька сутностей.

User – представляє зареєстрованого користувача програмної системи. Атрибути сутності User:

- First name – ім’я користувача;
- Last name – прізвище користувача;
- User name – коротке ім’я користувача;
- Country – країна, де живе користувач;
- Email – електронна адреса;
- Hashed password – хеш паролю;
- Picture key – шлях до фото профіля;
- Locale – локалізація користувача;
- User Type – роль користувача (Superuser, модератор, користувач).

Post – представляє пост із рецептом, що може опублікувати користувач. Атрибути цієї сутності:

- Title – назва рецепту;
- Description – опис рецепту;
- Preview key – шлях до фото попереднього перегляду рецепту;
- Youtube link – посилання на відео приготування рецепту;
- Cook time – час приготування страви;
- Complexity – складність приготування;
- Calorific – калорійність страви;
- Potions – кількість порцій;
- Healthy food – описує, чи рецепт відноситься до здорової їжі;
- Step by step – описує, чи є рецепт покроковим;

– With multicooker – атрибут, що визначає, чи приготування відбувається за допомогою мультиварки;

– Status – описує статус публікації: 0 – активна, 1 – на модерації, 2 – заблокована публікація;

– Reports – кількість скарг на публікацію;

– Private – описує, чи рецепт є у публічному доступі.

Comment – коментар користувача до публікації іншого користувача, має атрибут Content – текстовий вміст коментарю.

Recipe Book – книга рецептів, що може містити як рецепти власника так і рецепти інших користувачів. Має такі атрибути:

– Name – назва книги рецептів;

– Description – опис книги рецептів.

Post Action – описує дію, що виконали користувачі над постом. Це може бути вподобання або додавання у збережені. Має два атрибути: Preffered і Bookmarked, що означають, чи була публікація вподобана чи додана до збережених іншими користувачами відповідно.

Post Detail – покроковий опис публікації рецепту. Містить атрибути назви етапу приготування (Title), та його детального опису (Description).

Ingredient – інгредієнт рецепту. Містить такі атрибути:

– Name – назва інгредієнту;

– Optional – чи обов’язково його додавати у страву;

– Units – одинці міри кількості.

Сутності Category (Категорія), Subcategory (Підкатегорія), Specific (Специфікація – безглютенова дієта, наприклад), Assignment (Призначення – на обід, для дітей, наприклад), Cuisine (Національна кухня) мають атрибут Name.

User Settings описує користувацькі налаштування програми. Містить атрибути Language – налаштування мови інтерфейсу, Show LastActiveFollowings – показувати останніх користувачів, на яких підписаний користувач, ShowDayTimeRecipes – показувати рецепти відповідно до часу доби. Також усі сутності містять час створення запису, що необхідно для роботи із пагінацією –

розділення даних на невеликі частини із метою уникнення сильного навантаження на систему, оскільки даних може бути велика кількість.

Як видно із таблиці у користувацьких налаштуваннях є атрибут Locale та Language, що означає, що програмна система має багатомовний інтерфейс. Отже, було прийнято рішення додатково до довідкових сутностей, такі як Ingredient, Category і т.д. створити сутності, що будуть їх локалізувати за допомогою атрибутів Locale.

Також облікові дані користувачів, такі як електронна адреса, пароль, тип було винесено у окрему сутність UserIdentity, що із сутністю User має зв'язок «один-до-одного». Створено нові атрибути, такі як RefreshToken та RefreshTokenExpired – для реалізації системи захисту додатку за допомогою пари захисних токенів. Атрибут MarkedForDelete визначає, коли користувач помітив свій обліковий запис як видалений. За правилами соціальних мереж, повинна бути можливість відновлення користувацьких даних, а отже при видаленні акаунта він не видаляється фізично, а помічається спеціальним значенням, що дозволить системі за допомогою процедур видалити його, якщо користувач не відновить дані через певний час. EndBlockDate показує, коли користувач буде розблокований у системі.

Отже, провівши нормалізацію даних відповідно до третьої нормальної форми, було декомпозовано систему сутності, що зображені у вигляді ER-діаграми (рисунок В.1).

На діаграмі CK (Composite Key) – композитні ключі, які входять до складу первинного ключа та однозначно визначають відношення. Також з'явились додаткові сутності, що є проміжними при відношеннях «Багато-до-багатьох». Наприклад, сутність UserFollow визначає підписників та підписок користувача, сутність PostIngredient – набір інгредієнтів певного рецепту та їх кількість.

Так як кожна соціальна мережа вимагає реалізацію певної статистики, наприклад, кількість підписників, кількість вподобань, публікацій на конкретну категорію тощо. Для цього можна скористатися стандартними запитами, що є у всіх популярних реляційних СКБД. Проте, так як системи бази даних при

підрахунку кількості перевіряють дані на узгодженість відповідно до принципу ACID, цей процес може йти неприпустимо довго для високонавантажених систем. Тому прийнято рішення виконати незначну денормалізацію даних, що полягає у додаванні додаткових атрибутів-лічильників, що визначають кількість.

До сутності User було додано атрибути кількості, такі як FollowersCount – кількість підписників, FollowingsCount – кількість підписок, BookmarksCount – кількість збережених рецептів, LastPublicationTime – останній час публікації – для того, щоб не виконувати зайвих запитів при читанні даних. До сутності Post було додано: PreferencesCount – кількість вподобань, BookmarksCount – кількість разів додавання у збережені, Comments Count – кількість коментарів до публікації.

Отже, керуючись логічною моделлю можна проектувати реляційну фізичну модель. Проте для цього необхідно обрати правильну СКБД для реалізації цієї концептуальної моделі.

2.2.3 Вибір реляційної системи керування базами даних

Для вибору реляційної бази даних було розглянуто три популярних варіанта: MySQL, MS SQL Server та PostgreSQL.

Система керування базами даних MySQL – найпопулярніша безкоштовна для використання база даних з відкритим кодом. Вона підтримує декілька рушіїв бази даних, що відрізняються за своєю архітектурою та продуктивністю. Наприклад, механізм зберігання даних MyISAM використовуються в основному для читання, оскільки має високу продуктивність. Тому цю СКБД можна налаштувати під свої потреби. Ще однією перевагою є велика інтернет-спільнота та кількість технічної документації у вільному доступі. Також MySQL не потребує великої кількості комп'ютерних ресурсів. Для цієї системи керування базами даних є дуже багато інструментів моделювання та створення даних, середовищ виконання SQL-запитів та візуалізації даних. Є крос-платформною та може працювати на будь-якій операційній системі.

MS SQL Server – реляційна система керування базами даних, розроблена компанією Microsoft. В основному вона використовується для розробки комерційних додатків. Має багатофункціональний клієнт баз даних зі зручними інструментами розробника. Ще одною перевагою є те, що дана СКБД дуже добре інтегрована із платформою ASP.NET та швидко і легко налаштовується для використання. Є зручна та зрозуміла онлайн-документація від Microsoft. Продуктивність цієї СКБД висока. Має надійний механізм відновлення даних. Проте, як недолік – ціна версії Enterprise, що підтримує багато функцій для обслуговування високонавантажених систем. Вона занадто висока для впровадження у MVP-продукт, а у версії Express, що безкоштовна, функціонал має певні обмеження, наприклад, невеликий розмір сховища, слабкий рівень захисту даних користувачів тощо. Ще один недолік полягає у тому, що Microsoft SQL Server призначений лише для роботи на серверах під керуванням Windows. Тому можна зробити висновок, що дана СКБД не зовсім підходить для впровадження у програмну систему для автоматизації публікації, поширення та пошуку рецептів.

PostgreSQL – об’єктно-реляційна система керування базами даних. Відрізняється від усіх попередньо розглянутих варіантів тим, що повноцінно підтримує не реляційні дані та ефективно оброблює і зберігає їх. Наприклад, для пошуку даних у форматі JSON, що містяться у колонці відповідного типу, використовуються спеціальні оператори. Це дозволяє виконувати пошук даних такого типу набагато швидше, ніж це відбувається, наприклад у MySQL. Тому PostgreSQL є гнучкою у використанні. Також добре розвинена система індексації таблиць, що позитивно впливає на продуктивність при читанні із великих таблиц. За допомогою спеціальних індексів, повнотекстовий пошук виконується набагато швидше, ніж у аналогів PostgreSQL, що для соціальних мереж – одна з основних вимог швидкого отримання даних. Є кросплатформеною – може запускатись та розгортатись на будь-якій платформі. Одна із її особливостей – наявність підтримки сторонніх розширень, що робить цю СКБД максимально гнучкою. Це

дає змогу використовувати розширення, що ще збільшать продуктивність цієї бази даних. PostgreSQL має кращий, ніж у MySQL оптимізатор запитів, що ефективно виконує складні запити. Дуже розвинена система транзакцій. Також має велику інтернет-спільноту, створено багато технічної літератури, є зручна онлайн-документація. Дозволяє виконувати реплікацію бази даних – тобто створення копій БД із метою розвантаження системи. Також має драйвери для багатьох платформ та фреймворків, включаючи і ASP.NET.

Проте є невелика кількість недоліків. Збільшення швидкості обробки запитів, вимагає більше роботи, ніж у MySQL, оскільки PostgreSQL зосереджується на сумісності. Також при не коректному використанні можливостей даної СКБД, наприклад, індексів або автоматичне очищення від зайвих кортежів (Vacuuming), може значно сповільнитися виконання запитів, що негативно позначиться на продуктивності, тому налаштування, адміністрування та підтримка PostgreSQL набагато складніша, ніж у інших реляційних системах.

Отже, проаналізувавши найпопулярніші системи керування базами даних, для реалізації програмної системи було обрано СКБД PostgreSQL, оскільки вона має дуже багато переваг над іншими БД, постійно вдосконалюється та надає широкі можливості.

2.2.4 Проектування модульної архітектури

Після вибору системи керування базами даних виконується проектування модульної архітектури програмного забезпечення.

Система спроектована на основі семи зв’язаних між собою модулів. Усі вони належать до відповідних слоїв багато-шарової монолітної архітектури.

Основний модуль – Application. Він належить до презентаційного рівню. Тут міститься основна логіка програми, реєстрація функціональних сервісів конфігураційні файли, набір контролерів для комунікації із клієнтською частиною додатку, міграційні класи для керування історією змін структури баз даних. Саме

модуль Application відповідає за виконання додатку та поєднує у собі усі інші модулі програмної системи.

Модуль Contracts – не належить ні до якого шару архітектури, а представляє собою загальний набір інтерфейсів, що реалізовуються у інших модулях.

Entities – модуль, що містить моделі, які описують відображають реляційні дані. Він знаходиться на рівні керування даними (Data Layer). Також там знаходиться пакет класів DTO (Data Transfer Object). Він містить об’єкти, які використовуються у ролі представлення ресурсів запитів. Пакет Contexts складається із класів-контекстів, що виступають абстракцією над запитами до бази даних.

Модуль Services використовується для централізованого доступу до керування даними, тобто виконання над даними CRUD операцій (створення, читання, оновлення та видалення). Він належить до шару сервісів Service Layer.

Модуль Utility представляє допоміжний функціонал системи, наприклад, містить службу надсилання листів на електронні адреси. Схема основних пакетів подана на рисунку Г.1.

Шар даних містить об’єктні моделі, що є проекцією сутностей ER-діаграми. Кожен клас моделі представляє собою окрему сутність. Діаграма класів, що відображає реляційну модель подана на рисунку Д.1.

Майже усі класи дублюють інформацію сутностей ER-діагарми, проте створено нові, об’єднуючі класи, що містять базові атрибути для інших моделей. Усі класи успадковуються від PostgresEntityBase, оскільки усі сутності мають атрибут створення запису. Цей атрибут у майбутньому необхідний для виконання пагінації даних – розбиття даних на частини. PostgresRelatedEntity об’єднує класи, що зв’язані із об’єктом Post сутності. В основному це таблиці-зв’язки, які у ER-моделі виконують функцію розбиття відношень «багато-до-багатьох». У об’єктній моделі такі відношення називаються асоціаціями. Вони є проекцією відношень «один-до-багатьох» або «один-до-одного».

Для проекції реляційної моделі у об’єктну використовується технологія, що має назву ORM (Object-Relational Mapping). Компанія Microsoft пропонує таку

технологію – бібліотека Entity Framework Core (далі EF Core) – дозволяє працювати із СКБД, оперуючи даними на більш високому рівні абстракції, тобто дозволяє абстрагуватися від самої бази даних та її таблиць і працювати із даними незалежно від типу сховища. Доступ до даних EF Core виконує через спеціальні класи, що називаються контекстами.

Модуль DTO містить набори класів, що використовують лише ті дані, які необхідні для представлення моделі. Вони також можуть описувати лише окремі частини моделі. Сенс використання такого патерну проектування у тому, що зазвичай користувач не надсилає інформацію на сервер, що повністю б відповідала певній моделі, а лише її частину, або не потребує повної інформації моделі. І за допомогою модуля Mappings використовується співставлення DTO та об'єктної моделі. Класи DTO описані у 3 розділі записки, оскільки вони не є сталими та спроектувати їх заздалегідь не можна.

Модуль QueryObjects є основною одиницею шару сервісів у програмній системі. Він використовується для керування даними та виконання запитів. Є обгорткою над стандартними методами EF Core.

Взагалі розповсюджені два патерни проектування, що дозволяють отримати доступ до даних та інкапсулювати виконання стандартних CRUD-операцій над ними. Це «Репозиторій» (Repository) та «Об'єкт Запиту» (Query Object). Патерн «Репозиторій» представляє собою шар абстракції, що інкапсуює методи для роботи з даними. Використовується для відокремлення бізнес-логіки від деталей реалізації методів доступу до даних. Гарною практикою є використання на кожну бізнес-одиницю власний репозиторій, оскільки зазвичай у великих проектах неможливо охопити весь функціонал системи лише за допомогою базових CRUD-операцій. Тому, при такому проектуванні використовується ще один патерн, що називається «Одиниця Роботи» (Unit Of Work). Він слугує для зберігання усіх репозиторіїв, утворюючи єдину точу доступу до них та реєструється у якості синглтона при запуску програми. Синглтон – це один із найрозповсюдженіших

твірних шаблонів проектування, що використовується для створення лише одного екземпляру об'єкта. На рисунку 2.2 показано архітектуру програми із використанням патерну «Репозиторій» на прикладі трьох моделей програмної системи: User, Post та Comment.

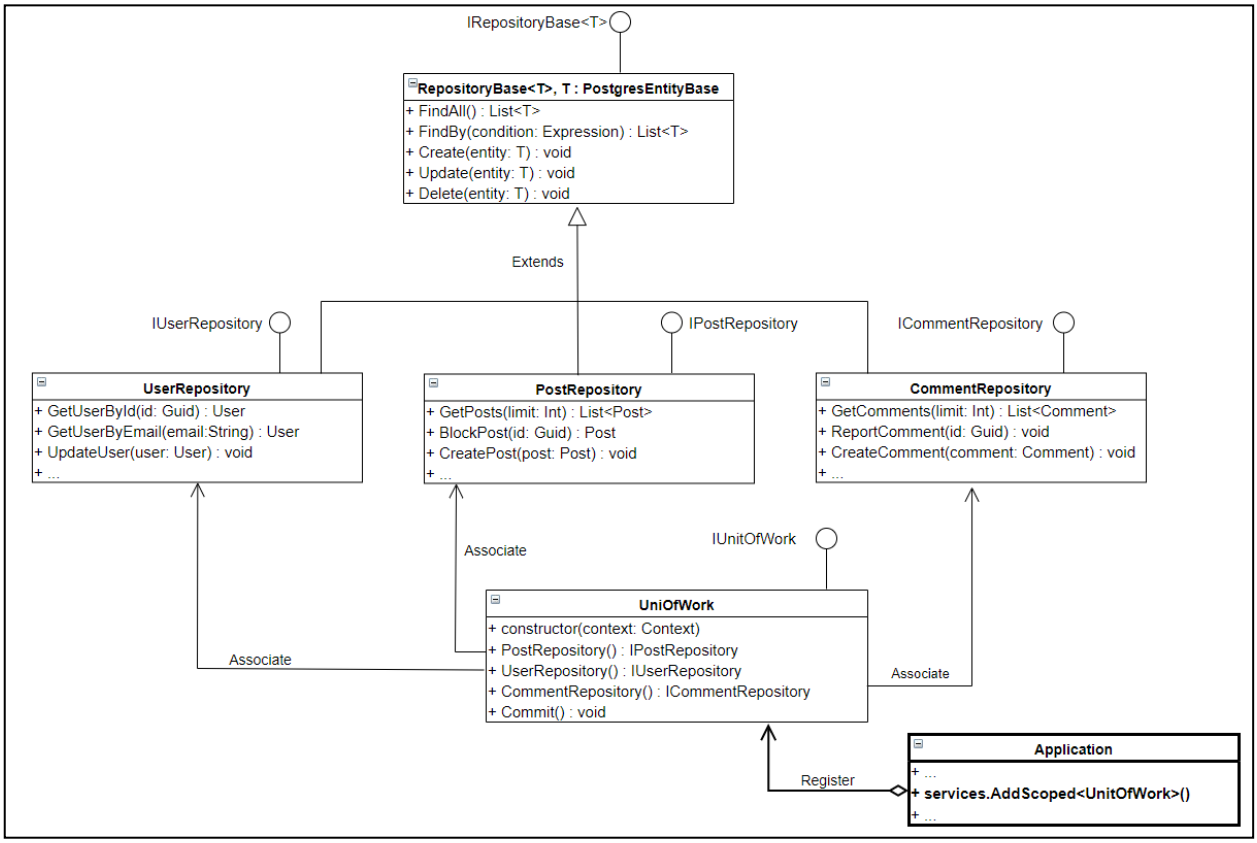


Рисунок 2.2 – Діаграма патерну проектування «Репозиторій»

Як видно зі схеми (рис. 2.2) усі класи супроводжуються відповідними інтерфейсами. Класи UserRepository, PostRepository та CommentRepository розширюють додатковими методами базовий клас RepositoryBase, що має лише основні CRUD-методи для роботи з даними. А клас UnitOfWork виступає єдиною точкою доступу до репозиторів та зареєстрований у основному класі програми Application у якості синглтона. Метод Commit зберігає усі зміни у базі даних. Великою перевагою патерну Repository є те, що за рахунок розділення на локальні репозиторії легше шукати дефекти у коді та проводити юніт-тестування. Також даний патерн проектування – є прикладом слідування парадигмі DDD

(Domain Driven Design) – предметно-орієнтоване програмування – створення моделей, що відповідають певній предметній області. Наприклад, кожний зі створених репозиторіїв виконує роботу з даними лише конкретної моделі.

Проте недоліком є те, що при великій кількості моделей зростає і кількість репозиторіїв та інтерфейсів, які ці репозиторії імлементують, що негативно впливає на LOC-метрики (Lines Of Code), суттєво уповільнює розробку програмного забезпечення та ускладнює його тестування. Крім того у проекті присутні нетривіальні запити до бази даних, такі, як, наприклад, курсорна пагінація чи повнотекстовий пошук із використанням вбудованого механізму сканування слів та фраз. Виконання таких запитів не підтримується на рівні стандартних методів EF Core, а отже такі запити тільки можливо виконувати без стандартних засобів бібліотеки, тобто напряму, через інтерполятор SQL-запитів, що також вбудований у EF Core. Це означає, що CRUD-методи, які реалізовані у базовому репозиторії практично застосовуватися не будуть, що автоматично призведе до надлишковості усіх локальних репозиторіїв.

Альтернативний патерн проектування – «Об’єкт Запиту» (Query Object). Він представляє собою набір об'єктів запитів. Кожен такий клас відповідає або лише за один запит, який необхідно виконати (наприклад, отримання списку користувачів), або групу запитів, що виконують спільну роботу на даними бізнес-логіки (наприклад, створення користувача вимагає відразу виконання трьох запитів, оскільки дані необхідно занести відразу у три зв'язані таблиці). Усі об’єкти запиту передаються у спеціальний клас – «Query Executor», що виконує ці запити та повертає результат виконання об’єкту, що його викликав.

Отже, перевагою такого підходу є те, що кожен об’єкт запиту – це окрема ізольована стурктура, що як і репозиторій, відповідає парадигмі DDD, тобто виконує лише обробку даних конкретної моделі. Над такими об’єктами легше провеводити юніт-тестування та виявлення дефектів, ніж над репозиторіями. Крім того, уся логіка виконання складних запитів інкапсульована у об’єкті запиту.

Проте недоліком є те, що навіть для звичайних запитів, які можна виконати стандартними методами EF Core, необхідно створювати об’єкти запиту. Тому

було прийнято рішення об’єднати патерн «Об’єкт запиту» та частину патерну «Репозиторій», тобто реалізувати лише методи для виконання базових CRUD-операцій. Результат подано на рисунку 2.3 на прикладі трьох об’єктів запиту: «GetUsersByName», «CreatePost» та «UpdateUserCredentials» – «Отримання списку користувачів по імені», «Створення публікації» та «Оновлення облікових даних користувача» відповідно.

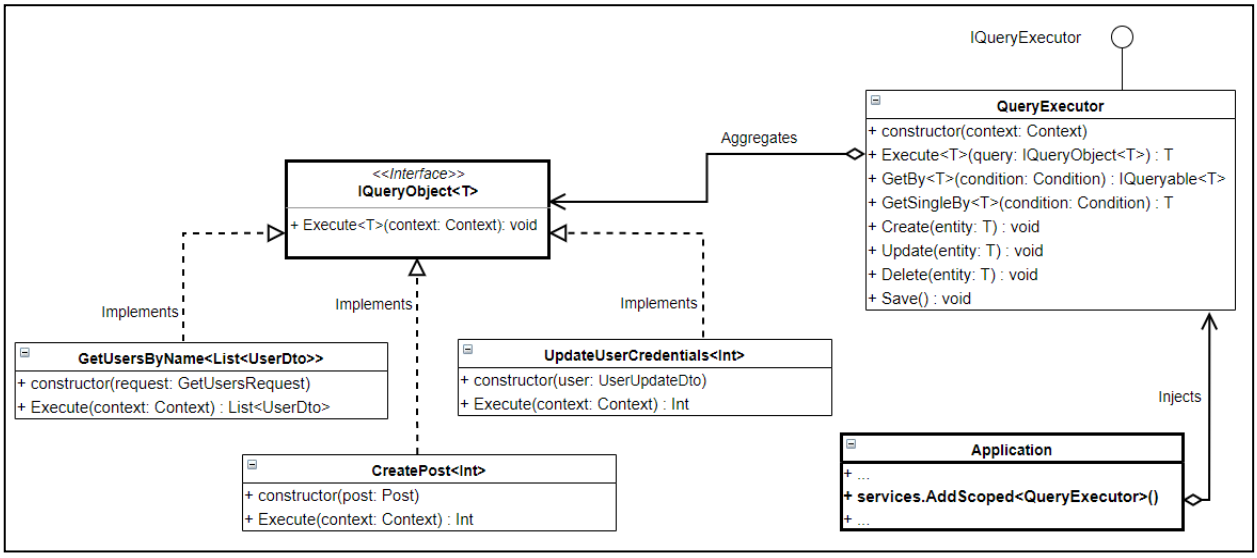


Рисунок 2.3 – Діаграма класів розширеного патерну «Об’єкт запиту»

Як видно зі схеми на рисунку 2.3 основним елементом у розширеному патерні проектування «Об’єкт запиту» виступає інтерфейс IQueryObject що містить єдиний метод Execute, який імплементують усі об’єкти запиту та у якому міститься логіка його виконання. Клас QueryExecutor використовується подібно класу UnitOfWork у патерні «Репозиторій», проте, на відміну від нього, він не утворює єдину точку доступу до усіх об’єктів запиту, а виступає у ролі їх виконавця, що відображає відношення агрегації між класом та інтерфейсом IQueryObject. Також цей клас містить усі базові методи та реєструється у головному класі програми у якості синглтону. Проте передбачити усі об’єкти запитів на етапі проектування архітектури практично не можливо. Їх реалізація проводиться паралельно до програмної реалізації системи, тому опис імплементації кожного об’єкту запиту подано у розділі 3 дипломного проекту.

Отже, розширений патерн проектування «Об'єкт запиту» є якісною заміною громіздкого «Репозиторію», легко тестується, кожен об'єкт є ізольованим та виконує лише строго визначені операції із даними конкретної бізнес-логіки.

Модуль менеджерів також відноситься до шару сервісів та виконує службові функції. Зазвичай менеджери покривають вбудовані у бібліотеки функції, розширюючи їх функціонал з метою уникання повторювальних ділянок коду у контролерах. Це дозволяє повторно використовувати методи та основний функціонал як вбудованих, так і сторонніх бібліотек. Менеджери реєструються у головному класі програми у якості синглтонів.

Презентаційний шар додатку – найвищий рівень тришарової монолітної архітектури веб-додатку. Він відповідає за відображення даних – рендеринг веб-сторінок. У випадку Web Api – відповідає за коректну обробку запитів та видачу ресурсів сервера у коректному форматі та надсилання їх клієнту. Основною одиницею є контролер – спеціальний клас, приймає запити, та за допомогою методів дії (Action methods) його обробляє та надсилає відповідь клієнту. У контролерах реалізована основна функціональність системи. Для обробки запитів вони використовують сервісний шар програмної системи, який надає для цього необхідний функціонал. Усі сервіси передаються у контролер шляхом механізму ін’єкції (впровадження) залежностей (Dependency Injection, DI) Це шаблон проектування, що використовується для передачі об’єктів іншим об’єктам замість повторного створення їх екземплярів. ASP.NET Core використовує цей патерн для реєстрації сервісів у якості синглтонів, а потім впровадження їх у інші сервіси або контролери. Патерн відокремлює створення залежностей сервісу від власної логіки сервісу, що дозволяє компонентам бути слабко зв'язаними. Це полегшує проведення модульного тестування, оскільки сервіси стають більш незалежними.

Крім того, презентаційний шар додатку поділяється на серверну та клієнтську частину. Саме контролери формують серверну частину шару представлення. Клієнтську частину представляє автономний веб-додаток, який надсилає запити та відображає клієнтську частину програмної системи.

2.2.5. Проектування функціональної архітектури

Функціональна архітектура програмної системи дещо відрізняється від модульної, оскільки представляє собою проектування саме функціональних одиниць програми. Один такий модуль може використовувати одразу декілька логічних модулів, що, виконуючи спільну роботу, досягають одного результату. Функціональна архітектура – це детальний опис і структура функціональності створюваної системи, спроектованої з урахуванням технологічних, бізнес-вимог, а також ієрархії функцій, їх залежності один від одного і використання в компонентах системи. Така архітектура побудована у відповідності до функціональних вимог, що моделюються і документуються за допомогою варіантів використання. Тому програмну систему для автоматизації публікації, поширення та пошуку кулінарних рецептів було декомпозовано на декілька основних функціональних одиниць.

Автентифікація – основа безпеки системи, один із найважливіших функціональних модулів. Полягає у перевірці достовірності даних користувача на сервері та є другим елементом захисту інформації. Перший – це ідентифікація – перевірка логіну та паролю користувача. Третій елемент – авторизація – перевірка прав користувача та визначення можливості доступу до захищених ресурсів.

У сучасних додатках для автентифікації використовується JWT (JSON Web Token) – це відкритий стандарт (RFC 7519) для створення токенів доступу у форматі JSON. Токени створюються сервером, підписуються секретним ключем і передаються клієнту, який в подальшому використовує їх для підтвердження своєї особи.

Токен складається із трьох частин: заголовок – там міститься інформація про алгоритм шифрування та тип; корисні дані – дані користувача, що необхідні для перевірки доступу до ресурсів та час завершення дії (інвалідація) токена; сигнатура – для підтвердження, що токен не був змінений. Проте використовувати один токен для доступу до ресурсів не надійно, оскільки, якщо його видобудуть зловмисники за допомогою, наприклад, атаки типу міжсайтового

скриптингу (XSS), то у нового користувача з’явиться можливість до постійного доступу до ресурсів. Тому прийнято використовувати пару токенів, один з яких – JWT – називається Access Token або токеном доступу, а інший, Refresh Token, токен оновлення, що може складатися із будь-якої допустимої кількості символів та бути зашифрованим будь-яким алгоритмом. Використання пари токенів підвищує безпеку, оскільки, навіть якщо буде вилучено, наприклад, токен оновлення, то без нього використання токена доступу буде неможливо. Якщо буде вилучено токен доступу, то користувач зможе ним скористатись певний час, проте, як тільки термін дії токена завершиться, без токена оновлення його використання також буде неможливим для отримання захищених ресурсів. Якщо термін дії токена доступу завершиться – необхідно виконати запит на сервер із надсиланням пари Access- та Refresh-токенів для отримання нової пари. Це дозволить користувачу не авторизовуватися повторно, а продовжувати користуватися системою неперервно. Повна діаграма послідовності автентифікації подана на рисунку Е.1.

Проте перед виконанням автентифікації, користувача необхідно створити та записати його дані у БД. При створенні користувача сучасні веб-додатки використовують різні способи підтвердження. У дипломному проекті запропоновано проводити верифікацію даних користувача за допомогою надсилання спеціального коду доступу на електронну пошту.

Так як програмна система – потенційно високонавантажена, зайві запити на додавання та видалення коду доступу із БД могли б негативно вплинути на швидкодію. Тому було прийнято рішення не створювати користувача у базі даних до тих пір, поки він не виконає верифікацію за допомогою коду доступу, а записувати дані користувача у кеш, де ключ – це захешована електронна адреса та пароль (верифікаційний токен), а значення – код доступу. Кеш – це сховище даних у оперативній пам’яті, що за рахунок цього оброблює запити набагато швидше, ніж БД. Крім того подібні сховища мають вбудований механізм інвалідації – видалення даних при завершенні терміну дії. При реалізації такого алгоритму створення користувачів було використано дистрибутивну систему

кешування Redis. Діаграма послідовності створення користувача системи подана на рисунку Е.2.

При створенні привілейованого користувача, тобто адміністратора та модератора, формується пароль, що також надсилається на електронну адресу, який користувач зможе використати для автентифікації у системі.

Зміна електронної адреси або паролю також супроводжується підтвердженням користувача за допомогою надсилання коду доступу на електронну пошту та використовує аналогічну послідовність операцій.

Після створення та автентифікації звичайного користувача, він окрім перегляду постів може виконувати такі дії, як створення нових публікацій та керування ними, оцінювання та зберігання та коментування публікацій інших користувачів. При створенні публікації, її вміст спочатку піддається модерації – тобто перевірки адміністраторами, чи публікація відповідає усім стандартним нормам соціальних мереж та у разі їх порушення – видалення публікації та надсилання користувачу відповідне попереджувальне повідомлення. Це попередня модерація. Існує також пост-модерація, що означає, що дані перевіряються уже після скарг інших користувачів і застосовуються необхідні дії. У даній програмній системі було вирішено реалізувати два види модерації. При публікації рецептів – використовується попередня модерація, при створенні коментарів та інші дії, наприклад, зміна профільного зображення використовується пост-модерація. Діаграма діяльності створення публікації подана на рисунку Ж.1.

Користувачі також можуть оцінювати публікації та додавати у збережені. Це часто виконувані операції, що також потребують оновлення лічильників (кількість вподобань та доданих публікацій у збережені), які використовуються для сортування рецептів, а оновлення у реляційній базі даних – не швидка операція, особливо, якщо на таблицю встановлено багато індексів, що використовуються для пришвидшення читання даних із бази.

Тому прийнято рішення використовувати кеш-сховище Redis. Принцип роботи аналогічний до вбудованого у ASP.NET Core Memory Cache. Відмінність

полягає у тому, що дані, у разі можливої їх втрати, наприклад, через переповнення кешу, можуть зберігатися на фізичне сховище. Також Redis може виступати я окремий сервер кешування, тобто не зв'язаний із основнимb процесами виконання програми та може розташовуватися взагалі на окремому фізичному сервері. Це дозволяє підвищити швидкодію додатку ще й за рахунок масштабування системи кешування та використовувати її як механізм для балансування навантаження та централізованого доступу.

Існує дві основні стратегії кешування: Cache-Aside та Write-Through. Cache-Aside – найпоширеніша із них. Основну логіку пошуку даних можна узагальнити таким чином:

а) коли програмі потрібно прочитати дані з бази, вона спочатку перевіряє кеш, щоб визначити, чи дані доступні;

б) якщо дані доступні (cache hit), кешовані дані повертаються, і відповідь надсилається клієнту;

в) якщо дані недоступні (cache miss), до бази надсилається запит отримання даних, якими потім заповнюється кеш, і дані надсилаються клієнту.

Першу стратегію на прикладі отримання облікових даних користувача ілюструє схема, подана на рисунку Ж.2.

У стратегії Write-Through на відміну від Cache-Aside кеш попередньо оновлюється відразу після первинного оновлення бази даних. Основну логіку читання даних можна узагальнити таким чином:

а) додаток оновлює постійне сховище даних;

б) відразу після цього дані оновлюються у кеші.

Зазвичай ці дві стратегії використовуються разом. Це значно прискорює роботу додатку.

Проте для оновлення даних було використано іншу стратегію кешування. Вона полягає у тому, щоб зменшити навантаження на базу даних у випадку високої активності та частоти надсилання запитів. Це може виникнути тоді, коли у користувача багато підписників і усі виконують запит, наприклад, додавання публікації у збережені. При оновленні лічильника виникне затримка у виконанні

операції, оскільки, як уже було вище зазначено, поле у реляційній БД оновлюється повільно. Тому логіку оновлення даних можна узагальнити таким чином:

а) клієнт надсилає запит на оновлення лічильника;

б) дані заносяться у кеш і усі наступні запити виконують збільшення (зменшення) значення цього лічильника у кеші;

в) за деякий час до визначеного терміну інвалідації кешу, усі дані заносяться однією транзакцією у базу даних, виконуючи запит оновлення таблиці у БД, а з кешу видаляються.

При цьому логіка читання відбувається за стратегією Cache-Aside, тобто цілісність даних повністю зберігається (рисунок Е.3).

Якщо при записі у базу даних виникне помилка та настав час інвалідації кешу, дані з кешу записуються у постійне сховище на фізичний диск і уже при наступній спробі запису з кешу до бази даних – спочатку перевіряється постійне сховище на диску, і, якщо там містяться дані, то вони заносяться у БД в першу чергу. Зазвичай термін інвалідації кешу встановлюється у межах від 5 до 10 хвилин. Так як не рекомендується використовувати механізм обробки події інвалідації кешу у Redis, оскільки цей процес не передбачуваний, і може відбутися як до інвалідації, так і після, тому було використано планувальник завдань (на діаграмі – Scheduler) та сторонню бібліотеку для його реалізації – Quartz.Net.

Видалення даних – також одна із функціональних вимог програмної системи. Відповідно до політики популярних соціальних мереж, видалення облікового запису виконується не одразу. Спочатку він помічається статусом «Готовий до видалення» та його можна відновити протягом пів-року або будь-якого терміну, що більше місяця. Якщо користувач не відновлює обліковий запис і термін відновлення завершився, обліковий запис видаляється назавжди. Реалізовано це також за допомогою планувальника завдань, який щодня перевіряє облікові записи та виявляє ті, термін відновлення яких завершився та видаляються. Аналогічний алгоритм використовується і при перевірці облікового запису на активність. Якщо користувач не активний, тобто не виконує вхід у

систему на протязі визначеного часу (6 місяців), то обліковий запис видаляється.

Для завершення терміну блокування облікового запису також використовується вище наведений алгоритм, та обліковий запис розблоковується автоматично. Дія розблокування облікового запису також доступна модераторам.

2.3 Проектування інтерфейсу користувача

Проектування інтерфейсу користувача включає такі етапи: розробка сценаріїв використання, проектування структури інтерфейсу, розробка прототипу інтерфейсу, визначення стилістики, вибір дизайн-концепції, оформлення екранів та анімація інтерфейсу.

Розробка користувацьких сценаріїв виконується на основі варіантів використання програмної системи. Опис варіантів використання подано у першому розділі дипломного проекту, тому можна перейти одразу до другого етапу проектування.

Відповідно до сценаріїв використання було розроблено основну структуру інтерфейсу користувача, а саме – визначено екрани, що відповідають розділам або модальним вікнам веб-додатку. Структура інтерфейсу зазвичай представлена діаграмою основних одиниць, що містить назву та список функцій, які визначені при проектуванні сценаріїв використання. Структура інтерфейсу зображена на рисунку 2.4. Позначка «Д» означає те, що даний елемент інтерфейсу – модальне або діалогове вікно, а «Р» – розділ.

Основні розділи системи – стрічка публікацій та адміністративна панель. Загалом уся функціональність веб-додатку представлена у вигляді модальних вікон. Так, наприклад, для створення нової публікації, авторизації або реєстрації, показу детальної інформації про рецепт, або для редагування особистих даних користувача використовуються модальні вікна. Для підтвердження дій користувачів, таких, як видалення, використовуються діалогові вікна із відповідним текстом.

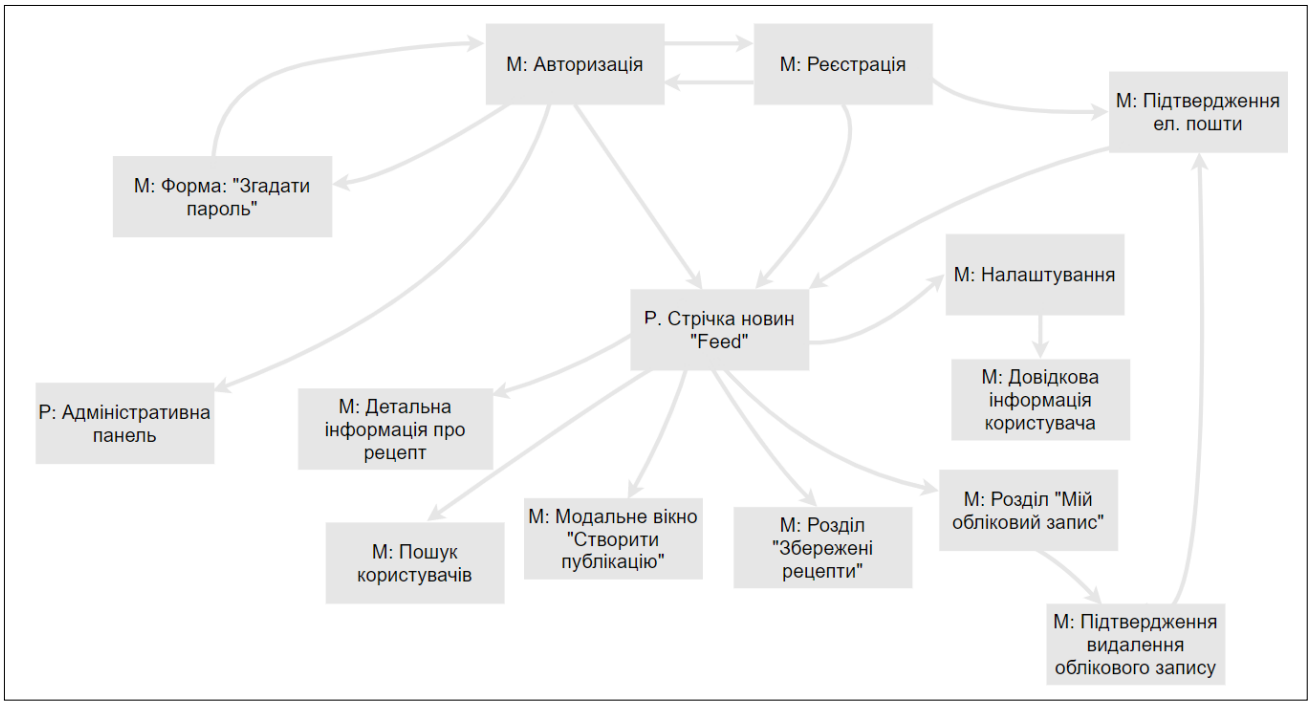


Рисунок 2.4 – Структура інтерфейсу програмної системи

Далі, відповідно до структури інтерфейсу будується прототип кожної структурної одиниці. В прототипах планується функціонал та розташування елементів сторінок відносно один одного. При виборі стилістики, що є наступним етапом у проектуванні, готуються набори зображень, іконки, шрифти та функціональні елементи (кнопки, поля вводу тощо). При виборі дизайн-концепції керуються в першу чергу темою додатку та вибирають сучасну тему, до якої уже звикли користувачі при використанні інших додатків. Крім того, тема повинна бути розроблена із врахуванням особливостей користувачів із фізіологічними відхиленнями. Наприклад, це може бути дальтонізм і необхідно використовувати такі кольори, які будуть чітко розрізнятися навіть користувачами із вадами зору.

У дипломному проекті запропоновано об’єднати ці три етапи проектування в один, із метою пришвидшення розробки додатку. У якості дизайну вибрано концепцію Material Design, що є популярною та зручною для користувача, оскільки містить зрозумілі елементи інтерфейсу.

Усі сторінки додатку мають навігаційне меню зліва. Головна сторінка також містить верхню панель та основну частину – стрічку новин.

Навігаційне меню містить п’ять пунктів: стрічка новин – головний розділ, розділ створення публікації, «Збережені рецепти», налаштування та обліковий запис користувача.

Стрічка новин містить публікації інших користувачів. На верхній панелі міститься поле пошуку рецептів. Пошук можна здійснювати по назві рецепта. Також поруч із полем пошуку розташована панель сортування рецептів. Можливе сортування за датою публікації, за кількістю вподобань, калорійністю, складністю та часом приготування. Є система фільтрів, розташована справа (рисунок И.1).

Зверху розташована панель найбільш активних користувачів та поруч є кнопка для пошуку інших користувачів. При натисненні на неї відкриється модальне вікно пошуку користувачів (риунок. И.2).

Є можливість пошуку по імені користувача. Справа розташована кнопка підписки (відписки). При натисненні на ім’я користувача відкриється його сторінка із публікаціями та публічна інформація про користувача.

Публікації оформлені у вигляді карток. Зверху картки міститься зображення профілю автора публікації та її назва. Також подано у скороченому вигляді інформацію про рецепт (час приготування, калорійність тощо). Нижче розташовується зображення рецепту та кнопки оцінки, коментарів та додавання у збережені, короткий опис рецепту, час публікації та кнопка «Детальніше», що відкриває модальне вікно, у якому подана детальна інформація про рецепт. Там міститься відео приготування страви, список інгредієнтів та категорії, до яких належить рецепт, етапи приготування, опис та коментарі (рисунок И.3).

У розділі створення публікації міститься форма для заповнення необхідних даних для публікації рецепту. Крім звичайного рецепту можна створити приватний рецепт та додати одразу у збережені, не публікуючи його при цьому. Інтерфейс розділу подано на рисунку И.4.

На рисунку И.5 подані форми авторизації, реєстрації, зміни електронної пошти та оновлення паролю відповідно.

Розділ «Мій обліковий запис» містить інформацію про користувача, а також його статистику (кількість підписників, рецептів тощо) та є можливість змінювати

дані та видалити обліковий запис, змінювати та видаляти публікації рецептів тощо. Мокап інтерфейсу подано на рисунку И.6.

При натисненні на посилання видалення облікового запису відображається діалогове вікно із підтвердженням дії (рисунок И.7).

Розділ «Збережені публікації» містить усі збережені користувачем рецепти. Крім того, публікації можна групувати по книгам рецептів. Книги можна сортувати за кількістю рецептів, що до них належать та датою створення. Проект інтерфейсу подано на рисунку И.8.

Інші розділи та елементи інтерфейсу майже не відрізняються від наведених.

Адміністративна панель доступна тільки модераторам після виконання авторизації із відповідною роллю. На ній містяться таблиці для створення, редагування та видалення категорій рецепту, а також панель керування публікаціями користувачів. Тобто присутній перегляд опублікованих публікацій і є кнопки, одна з яких відповідає за публікацію рецепту, а інша – за його видалення. Також присутня панель перегляду скарг на користувачів і присутні кнопки блокування користувачів у разі великої їх кількості.

На панелі навігаційного меню з’являється відповідна кнопка для адміністратора системи. Якщо користувач неавторизований, кнопки створення та збереження рецептів є неактивними. При натисненні на кнопку облікового запису – з’являється модальне вікно авторизації. Також це вікно з’являється тоді, якщо неавторизований користувач намагається оцінити публікації чи додати їх у збережені публікації.

2.4 Аналіз та вибір сучасних технологій для реалізації клієнтської частини веб-додатку

Для реалізації користувацького інтерфейсу – клієнтської частини програмної системи було проаналізовано декілька популярних технологій, а саме бібліотеку React та фреймворк Angular.

React – це розроблена компанією Facebook JavaScript-бібліотека для розробки інтерфейсу користувача односторінкових застосунків (SPA) – додатків, які не вимагають перезавантаження сторінки. Основною перевагою бібліотеки є швидкість розробки та наявність уже готових ресурсів та шаблонів, що можна застосовувати без розробки власних. React покращує продуктивність завдяки віртуальному дереву елементів веб-сторінки (DOM). Звичайний DOM оновлюється дуже повільно. React вирішив цю проблему, представивши віртуальний DOM. Віртуальний DOM повністю міститься у пам'яті і є поданням DOM веб-браузера. Завдяки цьому підвищується швидкодія додатку, оскільки при оновленні якого-небудь компонента, оновлюється тільки конкретний вузол документної моделі.

Оскільки React – це лише бібліотека, у комплексних додатках може призвести до ускладнення та збільшення термінів розробки. React охоплює лише шар інтерфейсу програми, не має чіткої архітектури побудови застосунків та не представляє повний набір інструментів (наприклад, сервіси, модулі, директиви тощо). Тому React – вдалий вибір лише для невеликих застосунків.

Angular – це інженерна платформа з відкритим кодом, що, як і React, використовується для побудови інтерфейсу користувача. Підтримується компанією Google. Одною з особливостей цієї платформи, є те, що замість JavaScript використовується об’єктно-орієнтована мова програмування TypeScript. Це значно полегшує підтримку продукту та його тестування. Також, на відміну від бібліотеки React, фреймворк Angular представляє комплексний вбудований функціонал у вигляді сервісів, модулів, компонентів та директив, що вибудовує базову архітектуру та принципи побудови веб-додатків і значно пришвидшує терміни розробки. Angular використовує ієрархічне впровадження залежностей та Ivy-візуалізатор, що перезавантажує стан компонента шляхом техніки «tree-shaking». Це позитивно впливає на швидкодію додатку. Також є вбудована підтримка RxJs – бібліотеки для обробки асинхронних викликів.

Недоліком Angular є складність його побудови та розуміння. Проте даний фреймворк використовується для складних додатків типу Enterprise, що

масштабуються, тому для реалізації інтерфейсу користувача програмної системи було обрано саме платформу Angular.

Отже, було проведено аналіз сучасної клієнт-серверної архітектури та встановлено, що сучасні веб-додатки усе більше використовують у якості серверної частини Web API.

Також було визначено переваги та недоліки монолітного та мікросервісного стилю побудови серверної архітектури і встановлено, що монолітний стиль є вдалим вибором для розробки мінімально життєздатного продукту.

У результаті аналізу типів сховищ даних, було обрано у якості основного – реляційну систему керування базами даних PostgreSQL, у якості шару кешування – сервер Redis. Також спроектовано концептуальну модель системи.

У процесі модульного проектування було виділено основні пакети програми, побудовано об’єктну модель та визначено «Об’єкт запиту» у якості основного патерну проектування на рівні сервісного шару монолітного стилю побудови серверної архітектури.

У результаті функціонального проектування було виконано аналіз функціональних вимог та на їх основі проведено опис основних процесів у системі, а також встановлено основні стратегії кешування у контексті конкретних проектних рішень.

Також було спроектовано основні елементи користувацького інтерфейсу, проаналізовано сучасні технології для його реалізації і вибрано платформу Angular для побудови клієнтської частини веб-додатку.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Створення бази даних програмної системи

Після проектування концептуальної моделі програмної системи та вибору реляційної системи керування базами даних, проектується та створюється власне база даних – основне сховище, що буде зберігати упорядковані дані. Перш за все на основі діаграми «Сутність-зв’язок» формуються таблиці та відношення між ними. Встановлюються первинні та зовнішні ключі. Виконавши усі необхідні дії у результаті ми отримали таблиці, що відповідають сутностям концептуальної моделі, поданої у додатку В. Після даного етапу створення бази даних проводиться аналіз таблиць, що можуть потенційно мати великий набір записів. У результаті аналізу було вибрано таблиці: Post (у ній зберігаються публікації користувачів, їх може бути необмежена кількість) та усі зв’язані таблиці із даною, що перебувають у відношенні «Багато до багатьох», наприклад, post_action, що характеризує дію користувача. Таблиця User, що зберігає дані користувачів, та зв’язані із нею таблиці також можуть мати потенційно велику кількість даних. Після виконання аналізу було обрано таблиці, на які будуть створюватися індекси. У реляційній базі даних, зокрема і у PostgreSQL, індекси – це спеціальні таблиці, що використовуються для прискорення пошуку даних у таблиці. При створені індексу та додавання на колонку усі дані, що уже існують у таблиці та ті дані, які у майбутньому будуть заноситись у таблицю будуть впорядковані за певними правилами в залежності від типу індексу. У PostgreSQL існує декілька основних типів індексів: B-tree (бінарне дерево), Hash (хеш-індекс), GiST, GIN (для повнотекстового пошуку). Для різних типів індексів застосовуються різні алгоритми впорядкування даних. Основною перевагою використання індексів – збільшення швидкості вибірки даних. Проте швидкість запису та оновлення може зменшитися, тому це необхідно врахувати при проектуванні індексної моделі. Для цього необхідно обрати стовпці таблиць та визначити, чи необхідно використовувати на ній індекс. І, якщо потрібно, обрати тип індексу. Індекси

також можуть створюватися, охоплюючи одразу декілька колонок. Тому було проведено додатковий аналіз та спроектовано індексну модель бази даних.

Запит для створення індексу B-tree наприклад на стовпець last_publication_time у таблиці User виглядає так:

```
CREATE INDEX last_publication_time_idx ON "User" USING btree
("LastPublicationTime" DESC, "UserId" DESC)
```

При створенні індексу задається його ім’я та визначаються стовпці, на які його потрібно встановити. При створенні B-tree індексу необхідно також порядок впорядкування даних (за зростаннями чи за спаданням) для пошуку. Аналогічно було створено інші індекс. Список стовпців таблиць та основних відповідних індексів подано у таблиці 3.1.

Таблиця 3.1 – Список індексів

Таблиця	Назви стовпців	Тип індексу
User	LastPublicationTime, UserId	B-tree (desc, desc)
	UserName, FirstName, LastName	GIN
Post	PostTitle	GIN
	CreatedAt, Id	B-tree (desc, desc)
	PreferenceCount, CreatedAt, Id	B-tree (desc, desc, desc)
RecipeBook	Name	GIN
Comment	CreatedAt, Id	B-tree (desc, desc)

Як видно із таблиці у БД присутні два типи індексів: B-tree та GIN. Перший використовується для пришвидшення вибірки відсортованих даних та прискоренні пагінації. GIN-індекс є специфічним та використовується для пришвидшення повнотекстового пошукую. У даній програмній системі він застосовується для пришвидшення пошуку рецептів по назві або користувачів по

імені, прізвищі та користувацькому імені.

Для покрокової вибірки даних використовується пагінація. Найпоширеніший тип пагінації – «офсетна». Вона виконується шляхом встановлення ліміту, тобто кількості рядків, що буде вибрана та кількості рядків, що буде пропускатися від початку вибірки. Крім того, дані повинні бути впорядковані. Такий вид пагінації не є оптимальним у випадку великої кількості даних, оскільки із кожним кроком збільшуватиметься кількість рядків, які необхідно пропустити.

Інший вид пагінації – Keyset. Вона передбачає встановлення умови, за якої будуть обиратися рядки, що увійдуть у блок пагінації. Дані повинні бути впорядкованими. Основна перевага у тому, що на відміну від «офсетної», даний тип пагінації не використовує механізм пропуску рядків. Це підвищує швидкодію вибірки. Також дана пагінація застосовується коли не має необхідності розділення блоків даних на конкретні чітко визначені сторінки. Це відповідає вимогам програмної системи. Тому її було обрано у якості основного механізму пагінації. Для збільшення швидкості вибірки даних було встановлено на відповідні стовпці B-tree індекси зі зворотнім порядком (табл. 3.1). Типовий запит вибірки даних із використанням keyset-пагінації виглядає таким чином:

```
SELECT * FROM "Post" WHERE ("Post"."CreatedAt", "Post"."Id") <
(dateMark, idMark) AND "Post"." CreatedAt" < initialDateTime
```

У даному запиті використано кортеж для порівняння даних, оскільки може виникнути ситуація, коли час створення двох публікацій однаковий, тому ще встановлена додаткова умова по унікальному ідентифікатору. Поля-маркери dateMark, idMark – дата на ідентифікатор останньої публікації із вибірки попередньої пагінації, а initialDateTime – час, коли був запущений процес вибірки, та усі наступні пагінації у даному циклі вибірки будуть обмежуватися саме цією датою із метою уникнення читання нових даних, що можуть не відповідати конкретним умовам і може виникнути ситуація «брудного читання», тобто

повторне читання одних і тих самих даних із БД.

Усі складні запити, та запити, що базуються на вхідних параметрах було об’єднано у процедури та функції. Це зменшує повторюваність коду. Функції та процедури створюються протягом реалізації функціональної архітектури програмної системи.

3.2 Реалізація серверної частини програмної системи

Після створення бази даних та відповідно до спроектованої модульної архітектури програмної системи, можна приступати до її реалізації. За допомогою інструментарію компанії Microsoft – редактору коду Visual Studio при виборі нового проекту WebApi створюється базовий шаблон. Основний файл – Startup.cs, що включає в себе усю необхідну базову логіку для запуску додатку. Файл appsettings.json містить основні налаштування проекту. Там розташовані строки підключення до бази даних та конфігурація Redis-серверу, налаштування токена авторизації тощо. Зазвичай використовується два файла: для розробки (appsettings.development.json) та для розгортання (appsettings.production.json). У першому використовуються локальні налаштування, у другому – налаштування, що будуть використовуватися на хостингу. У папці Controllers – містяться контроллери.

Для виявлення потенційних помилок у програмі було прийнято рішення встановити бібліотеку NLog, що застосовується для запису помилок (логів) та зберігати файли на фізичний диск.

Відповідно до модульної архітектури було створено відповідні пакети, що подані на рисунку Г.1, а саме: Application – основний модуль програми, що формується при створенні нового веб-проекту, Entities – сутності моделі бізнес логіки, Services – об’єкти сервісного шару програми, Utils – додатковий функціонал. Кожен пакет – це окрема бібліотека.

Реалізація серверної частини починається зі створення моделей бізнес-

логіки. Ці моделі є реляційним відображенням таблиць бази даних. Класи створюються відповідно до об'єктної моделі, поданої у додатку Д. Усі класи успадковуються від абстрактного класу PostgresEntityBase. Цей клас є об'єднуючим та не містить властивостей. Клас DatedEntity містить дату створення сутності. Від нього успадковуються усі класи, що використовують значення дати створення. Усі інші класи містять властивості, аналогічні рядкам таблиць у БД.

Після створення моделі бізнес-логіки було реалізовано контекст бази даних – точка доступу до керування запитами до БД. Контекст представляє клас PostgresContext та має такий вигляд:

```
public class PostgresContext : DbContext
{
public PostgresContext(DbContextOptions<PostgresContext>
options): base(options)
  {
    ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior
    .NoTracking;
  }
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)=> optionsBuilder.LogTo(Console.WriteLine);

protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<UserFollow>()
      .HasKey(u => new { u.FollowerId, u.FollowingId });

    /*..............*/
  }
  public DbSet<User> Users { get; set; }
  /*..............*/
}
```

За допомогою класу ChangeTracker, що вбудований у ORM-бібліотеку Entity Framework Core для підвищення продуктивності встановлено поведінку, що скасовує стеження за змінами у моделі.

У методі OnConfiguring встановлено механізм запису логів запитів, що виконує EF Core до бази даних. Це дозволяє виявити неправильно сформовані ORM-системою запити, що можуть негативно вплинути на продуктивність та переписати їх на більш ефективні.

Для сутностей, що мають складений первинний ключ встановлено явні вказівники на ці ключі для ORM-системи у методі OnModelCreating.

Нижче містяться набори даних, що заповняються значеннями з БД при створенні запитів до неї.

Створений контекст даних за допомогою механізму впровадження залежностей реєструється у головному файлі Startup.cs. Проте реєстрації усіх сторонніх сервісів було вирішено винести в окремий файл Extensions.cs задля уникнення нагромадження коду в основному файлі.

Далі було реалізовано кістяк патерну «Об'єкт запиту», що входить до сервісного шару серверної архітектури. Він складається із інтерфейсу IQueryExecutor та відповідного класу, що його реалізовує. Інтерфейс та його реалізацію також зареєстровано у головному файлі. Він має такий вигляд:

```
public interface IQueryExecutor
{
  Task<T> Execute<T>(IQueryObject<T> query, bool autoSave = true);

  Task<T> CachedRead<T>(IQueryObject<T> query, string cacheKey)
    where T : class;

  IQueryable<T> GetBy<T>(Expression<Func<T, bool>> condition)
    where T : PostgresEntityBase;

  Task<T> GetSingleBy<T>(Expression<Func<T, bool>> condition)
    where T : PostgresEntityBase;

  Task<bool> ExistsBy<T>(Expression<Func<T, bool>> condition)
    where T : PostgresEntityBase;

  Task<T> Create<T>(T entity, bool autoSave = true)
    where T : PostgresEntityBase;
   Task CreateFew<T>(IEnumerable<T> entities, bool autoSave = true)
     where T : PostgresEntityBase;

   Task Update<T>(T entity, bool autoSave = true)
     where T : PostgresEntityBase;

   Task Delete<T>(T entity, bool autoSave = true)
     where T : PostgresEntityBase;
}
```

Методи GetBy, GetSingleBy, ExistsBy, Create, CreateFew, Update, Delete – узагальнені методи із патерну «Репозиторій», що у своїй реалізації викликаються стандартні методи по управлінню операціями з БД, що вбудовані у EF Core. Метод Execute призначений для виконання нестандартних запитів до БД за допомогою спеціальних класів – об'єктів запиту, що реалізують інтерфейс IQueryObject. Усі об'єкти, що реалізують даний інтерфейс інкапсулюють логіку виконання запитів до БД. Вони виконуються за допомогою методів, що вбудовані у EF Core, такі, як FromSqlInterpolated та ExecuteSqlInterpolated. Використання їх дає повний контроль над запитом, оскільки він написаний за допомогою чистого SQL, а не згенерований автоматично ORM-системою. Це дозволяє гнучко оперувати даними у БД, оскільки можна виконувати, наприклад, запити виклику функцій чи процедур, створення таблиць або усі запити, що не підтримуються стандартними засобами EF Core (наприклад, запити повнотекстового пошуку, що містять оператори, властиві тільки СУБД PostgreSQL).

Далі наведено приклад об'єкта запиту оновлення профільного зображення у обліковому записі користувача:

```
public readonly struct UpdateUserPicture : IQueryObject<int>
{
    private readonly Guid id;
    private readonly string pictureBlobKey;

public UpdateUserPicture(Guid id, string pictureBlobKey = null)
{
    this.id = id;
    this.pictureBlobKey = pictureBlobKey;
}
 public async Task<int> Execute(PostgresContext context)
 {
   FormattableString sql = $@"UPDATE ""User"" SET ""PictureKey"" =
   {pictureBlobKey} WHERE ""UserId"" = {id}";

   return await context.Database.ExecuteSqlInterpolatedAsync(sql);
 }
}

await executor.Execute(new UpdateUserPicture(CurrentUserId,
fileName));
```

Замість класів у якості об’єктів запиту вирішено використовувати структури, що являються значеннєвим типом даним, а не типом-посилання. Це означає, що при створенні нового екземпляру структури вона буде розташовуватись у стеку, а не у купі, як класи. Внаслідок цього відсутні зайві виклики механізму очищення пам’яті (Garbage Collector), що збільшить продуктивність додатку.

Усі інтерфейси містяться в окремо створеному проекту Contract. Структура пакетів у цьому проекті відбиває модульну архітектуру програмної системи.

Для реалізації узагальнених функцій системи було створено менеджери. Це класи, що належать до сервісного шару модульної архітектури.

LoggerManager – використовується для запису логів можливих помилок.

AuthManager – відповідає за генерацію та налаштування авторизаційного токену доступу та оновлюючого токену. Тут встановлюється час завершення дії токена доступу. У цілях безпеки вибрано короткий термін – 15 хвилин. Час завершення дії оновлюючого токену може бути довше, так як він використовується для підтвердження валідності токену доступу (7 днів). Refresh Token (оновлюючий токен) зберігається у БД. Зашифровані дані токена, що називаються Payload, містять ідентифікатор користувача та його роль, за рахунок чого ну стороні клієнта визначатиметься, яка частина інтерфейсу має бути захищена, а яка доступна.

MemoryCacheManager – містить функції по управлінню тимчасовим сховищем даних в оперативній пам’яті Redis. Містить основні методи доступу до даних та операції читання, оновлення та видалення даних.

AWSS3BucketManager – виконує функції управління файлами у хмарному сервісі компанії Amazon – S3. Усі авторизаційні дані для захищеного доступу до S3 містяться у конфігураційних файлах appsettings.json.

Усі менеджери зареєстровані у головному класі програми шляхом впровадження залежностей.

Після вищенаведених операцій виконується реалізація функціональної архітектури програмної системи. Створено систему контролерів, що відповідають

бізнес-вимогам. Кожен контролер взаємодію із конкретною моделлю або зв'язаними моделями бізнес-логіки. У ньому реалізуються кінцеві точки та у кожному методі інкапсулюється логіка для роботи із даними. Зазвичай, усю цю логіку відділяють від контролерів ще одним шаром, що називається Business Services Layer, проте було вирішено не відокремлювати усю логіку в окремий шар абстракції, оскільки це негативно може вплинути на швидкодію додатку.

Метод контролера, що називається ActionMethod приймає клієнтський запит та повертає відповідь у вигляді даних та статус код, що свідчить про успішність запиту клієнта. Для методів, що мають параметри створено атрибут, що забороняє клієнтам надсилати порожні значення. Усі атрибути описані у Application-модулі та містяться у папці Attributes. Також на деяких методах встановлені атрибути максимальної кількості запитів за хвилину для захисту від занадто частих викликів.

Найчастіше параметри методів у контролері представляють об'єкти, що посилає клієнт для оновлення або отримання ресурсу, які не відповідають структурі моделі бізнес-логіки. І отримана відповідь також може відрізнятися від класів об'єктної моделі. Тому було реалізовано патерн, що називається DTO (Data Transfer Object). Він представляє тимчасовий об'єкт, який за необхідності або конвертується у модель для збереження у БД, або навпаки, надсилається клієнту. Для конвертування DTO у моделі бізнес логіки та навпаки використано сторонню бібліотеку Automapper.

Усі DTO-об'єкти умовно поділено на три типи, що знаходяться у відповідних пакетах: Input, Output та Request. Input-об'єкти – це класи, що використовуються як параметри методів контролера при надсиланні даних клієнтом. Вони серіалізуються у класи моделей бізнес-логіки та заносяться у БД. Output-об'єкти – це класи, що навпаки, посилаються клієнту у відповідь. Вони містять дані, отримані з БД. Request-об'єкти, або об'єкти-запити – це класи, що як Input-об'єкти посилаються клієнтом, проте вони ніяк не впливають на базу даних, а лише диктують умови для того чи іншого запиту вибірки із БД. Request об'єкти в основному можна поділити на прості та ті, що використовуються у пагінації.

Для таких об’єктів створено базовий абстрактний клас BasePaginationRequest, що містить три властивості, які характеризують запит вибірки, оснований на пагінації. Далі подано його реалізацію:

```
public abstract class BasePaginationRequest<T> where T : struct
{
   public T? IdMark { get; set; }

   [Required]
   [Range(3, 1000)]
    public int PagingLimit { get; set; }

    public virtual bool FirstPagination() => IdMark == null;
}
```

IdMark – це поле, що характеризує унікальний ідентифікатор останнього значення попереднього етапу пагінації. Застосовується для визначення наступного блоку пагінації. PaginLimit – установлює ліміт вибірки даних. Та метод FirstPagination, що визначає чи є етап пагінації першим у циклі вибірки та чи потрібно враховувати усі вище вказані умови під час цієї вибірки.

Усі DTO, що використовують пагінацію, що базується на даті створення сутності, розширюють клас DateTimeBasePainationRequest, що в свою чергу розширює базовий клас пагінації. Його реалізацію подано у наступному лістингу:

```
public class DateTimeBasedPaginationRequest<T> :
BasePaginationRequest<T> where T : struct
{
  [Required]
  [StringLength(maximumLength: 256, MinimumLength = 2)]
  public string SearchTerm { get; set; }

  public DateTime? InitialDateTime { get; set; }

  public DateTime? DateTimeMark { get; set; }

  public override bool FirstPagination()
            => base.FirstPagination() || InitialDateTime == null ||
               DateTimeMark == null;
}
```

InitialDateTime – ця властивість характеризує час, коли був запущений цикл вибірки, у який може входити декілька пагінацій, відповідно до ініційованих клієнтом запитів. Це унеможливить вибірку даних, які були створені пізніше, аніж користувач ініціював цикл отримання даних.

DateTimeMark – вказує на час публікації останнього запису попередньої пагінації з метою визначення наступної.

SearchTerm – представляє пошуковий запит користувача.

Усі об’єкти Request DTO лише містять вказівки, які дані відправляти користувачу. До них відносяться і фільтри рецептів.

Методи у контролерах можуть мати як вільний, так і захищений доступ. У випадку останнього метод помічається атрибутом Authorize, що означає, що тільки авторизований користувач має доступ до цього методу. Також створено атрибут фільтру, що визначає користувача, який виконав запит. Це може бути як простий користувач так і модератор. Модератор має права на редагування категорій, видалення користувацьких постів, що не відповідають політиці та правилам соціальної мережі, виявлення користувачів, на яких були надіслані скарги від інших користувачів.

Деякі методи у контролерах покриті механізмом кешування. Наприклад, метод отримання детальної інформації про рецепт використовує стратегію кешування Cache-Aside, тобто кешує дані на певний короткий термін. Для цього використовується метод CachedRead, лістинг якого поданий далі:

```
public async Task<T> CachedRead<T>(IQueryObject<T> query, string
cacheKey) where T : class
{
    T data = await cache.Get<T>(cacheKey, true);

    if (data == null)
    {
         data = await query.Execute(context);
         await cache.Set(cacheKey, data);
    }

    return data;
}
```

Якщо є закешовані дані, то метод повертає їх, якщо даних немає, то виконується запит до бази даних і дані заповнюються у кеш.

Інша стратегія кешування – відкладене оновлення даних. Використовується, наприклад для оновлення лічильників кількості вподобань. Коли користувач виконує дану дію, дані не заносяться одразу у БД, а спочатку записуються у кеш. Разом із нею використовується і вище згадана стратегія Cached-Aside для того, щоб синхронізувати дані, після запису їх у кеш.

Для відкладеного оновлення даних використовується планувальник завдань. Для його реалізації використано сторонню бібліотеку Quartz.net. Вона дозволяє зареєструвати об’єкти, що називаються задачами (Job), та виконувати їх із вказаною періодичністю. Для цього було створено фабрику об’єктів, що ініціалізують планувальник та самі об’єкти-задачі. Приклад задачі, що перевіряє користувачів, що не були активними та не авторизовувались протягом півроку та помічає їх як видалені подано у наступному лістингу:

```
[DisallowConcurrentExecution]
public class MarkUserAsRemovedJob : JobBase
{
   private readonly IQueryExecutor executor;
   private readonly ILoggerManager logger;

   public MarkUserAsRemovedJob(IQueryExecutor executor,
                               ILoggerManager logger)
   {
       this.executor = executor;
       this.logger = logger;
   }

   public override async Task Execute(IJobExecutionContext context)
   {
      try
      {
        await executor.Execute(new UpdateMarkUserAsRemoved());
      }
      catch (Exception ex)
      {
         logger.LogError($"{nameof(MarkUserAsRemovedJob)} |
{DateTime.Now} \n {ex.Message}");
      }
    }}
```

Уся логіка реалізована у методі Execute – посилається відповідний об’єкт запиту та виконується процедура у БД, що має назву mark_users_for_remove. Як параметр передається поточний час та порівнюється із часом останньої активності користувача. Ця задача реєструється у інверсійному контейнері шляхом впровадження залежностей. Для задач встановлюється проміжок часу у вигляді виразу Cron. Для даної задачі визначено, що перевірка буде виконуватись один раз на добу у час найменшої завантаженості сервера – вночі. Усі відкладені задачі знаходяться у пакеті ScheduleServices і також належать до сервісного шару модульної архітектури.

При реєстрації користувача відправляється на його електронну пошту із кодом підтвердження особи. Відправлення листів реалізовано за допомогою сторонньої бібліотеки MailKit. При реєстрації користувача, відновлення паролю та відновлення облікового запису формується короткий буквено-цифровий код, записується у кеш на короткий термін та відправляється користувачу на електронну пошту. Ключем у кеші виступає код доступу, а значенням захешована електронна пошта за допомогою бібліотеки хешування BCrypt. Після того, як користувач надсилає код підтвердження, дані звіряються з тими, що містяться у кеші і видаляються. Метод надсилання коду виглядає так:

```
[AllowAnonymous]
[HttpPost("sendAccessCode")]
[ServiceFilter(typeof(ValidationFilterAttribute))]
public async Task<IActionResult> SendAccessCode([FromBody]
                                UserForSendingAccessCodeDto userData)
{
   string accessCode = Hashing.GenerateCharSet();
   string locale = userData.Locale ?? userData.Language;

   string hashedEmail =
   BCrypt.Net.BCrypt.HashPassword(userData.Email)

   if (await cache.KeyExists(userData.Email))
   {

     ModelState.AddModelError(Constants.AccessCodeAlreadySentKey,
     Constants.AccessCodeAlreadySentMsg);
     return BadRequest(ModelState);
```

```
    }

var emailBody = Mailing.GetWelcomeEmailBody(locale, accessCode);

await Mailing.SendEmail(configuration, userData.UserName,
userData.Email, emailBody);

    if (!await cache.KeyExists(accessCode))
     {
       await cache.Set(userData.Email, string.Empty);
       await cache.Set(accessCode, hashedEmail);
     }

return Ok();
}
```

Вміст листа визначається відповідно до локалізації клієнта. Усі шаблони для надсилання листа містяться у пакет Utils. Також цей пакет містить засоби для формування коду доступу, містить усі константні вирази.

3.3 Реалізація клієнтської частини програмної системи

У якості інструменту розробки клієнтської частини (фронт-енду) веб-додатку було обрано платформу Angular. При реалізації використовується компонентна модель. Тобто кожен компонент – логічна одиниця – елемент, або група елементів користувацького інтерфейсу. Компонент складається із трьох частин: шаблон – це розмітка HTML, файл стилів CSS, або інший, оснований на CSS, препроцесор та клас компоненту, що використовує мову програмування TypeScript. Шаблон та клас компоненту тісно зв’язані за допомогою спеціальних директив фреймворку. Усі компоненти об’єднуються у логічно зв’язані модулі, що у свою чергу можуть бути також об’єднані.

Крім компонентів Angular пропонує використання сервісів. Це спеціальні класи, що можуть впроваджуватися у компоненти за допомогою розвиненого механізму впровадження залежносте та виконувати специфічні для сервісу функції. Зазвичай використовуються так звані http-сервіси, що приймають запити та повертають об’єкт, що називається Observable Object. Для того, щоб отримати

дані із серверу необхідно підписатись на цей об’єкт за допомогою методу subscribe. Цей функціонал надає бібліотека, вбудована у фреймворк, що називається RxJs. Це бібліотека для реактивного програмування, що дозволяє зручно організовувати роботу із подіями та асинхронним кодом шляхом реалізації логіки, використовуючи декларативний стиль. Вона побудована із використанням патерну проектування «Спостерігач», дає змогу одним об’єктам стежити й реагувати на події, які відбуваються в інших об’єктах. Ця бібліотека надає ряд корисних методів по управлінню підписками та порядком оновлення даних.

Реалізації клієнтської частини починається із визначення основних моделей та побудови основних сервісів. Усі моделі було прийнято поділити на два типи: Request та Response. Перший відповідальний за побудову запитів на сервер, останній – за серіалізацію відповіді сервера. Усі моделі представляють собою інтерфейси. Наприклад модель, що відповідає за зберігання даних розшифрованого токена доступу виглядає так:

```
export interface IUserAuthResponseDto {
  userId: string,
  userType: UserType
}
```

Вона містить такі поля, як ідентифікатор та тип користувача. Є моделлю типу Response. Моделі-запити у свою чергу побудовані аналогічно до серверних DTO типу Request.

Після створення моделей бізнес-логіки проектується модель роутингу. Це механізм маршрутизації у додатку, що дозволяє зіставляти запити до веб-додатку із певними ресурсами всередині програми. У даному веб-додатку було вирішено створити два маршрути: один, що доступний звичайним користувачам та інший – для модераторів. Для унеможливлення доступу звичайного користувача до адмін панелі, було реалізовано клас, що блокує перехід у адміністративний розділ програми, що називається GuardObject. Він зчитує дані із токена доступу та на основі типу користувача приймає рішення про доступ до розділу.

Далі було створено сервіси. Їх можна поділити на дві групи: Http Services – відповідають безпосередньо за надсилання запитів та обробку відповідей від сервера. У якості помилки зі статусом 500 сервер повертає одне й те ж повідомлення, тому, щоб уникнути повторення коду, було реалізовано клас, для перехоплення відповідей, та обробки її ще до надходження у сервіс. Такі класи називаються інтерсепторами. Аналогічно і працює інтерсептор, що виконує обробку у зворотньому порядку – він відповідає за встановлення спеціального заголовку, що містить Bearer Token – токен доступу, який посилається на сервер у вигляді заголовку запиту із метою отримання даних захищеного ресурсу. На кожен запит клієнта, що авторизований, цей інтерсептор встановлює такий заголовок та відправляє на сервер. Приклад частини http-сервісу наведено у наступному лістингу:

```
@Injectable({ providedIn: 'root' })
export class UserService {
   constructor(
      private httpClient: HttpClient,
      private tokenSettingsService: TokenSettingsService) {}

    getUserInfo(): Observable<ICurrentUserResponse> {
         return this.httpClient.get<ICurrentUserResponse>
                (environment.baseApiUrl + ApiUrls.UserInfo) }
}
```

Атрибут Injectable означає, що сервіс буде доступний для впровадження у будь-якому модулі додатку. Метод getUserInfo вертає об’єкт спостереження відповідного запиту на сервер. Коренева адреса серверу розташовано у файлах середовища (environment), тому що ця адреса на хостингу буде відрізнятися від локальної. Також введено перерахування, де містяться усі кінцеві точки серверу. Усі http-сервіси мають аналогічний функціонал.

Інший тип сервісів – Helpers Services. Вони виконують допоміжні функції у додатку. Наприклад, це засоби по виведення вікна-повідомлення користувачу – Alert Service, сервіс для взаємодії із локальним сховищем веб-браузера – Local Storage Service.

Існує ще один тип сервісів – Store Services. Вони відповідають за глобальний стан даних у програмі, оскільки, коли компонент видаляється із DOM-дерева, видаляються і усі дані відповідно. Також ці сервіси відповідають за доступ до даних із будь якого компонента. У веб-додатку було створено один глобальний сервіс для збереження даних зареєстрованого користувача, які мають бути доступні у всіх компонентах.

Після реалізації сервісів відповідно до проекту інтерфейсу виконується проектування та створення компонентів та модулів програми. Опис основних компонентів подано далі. На рисунку 3.1 зображено основні компоненти додатку, позначені цифрами.

Nav Bar (1) – компонент навігаційної панелі. Відповідає за її відображення та налаштування.

Search Input (2) – представляє поле пошуку. При введені тексту, користувачу посилаються відповідні до пошукового запиту дані. Проте з метою зниження навантаження на систему, дані надсилаються не одразу, а після того як набір тексту завершився. Для цього реалізовано допоміжний сервіс Debouncer.

Main Feed (3) – представляє стрічку публікацій користувачів. Контейнер прокрутки реалізований на основі технології віртуалізації даних, який поставляє стороння бібліотека Ngx-Scroller. Ця технологія дозволяє вміщувати у контейнері прокрутки велику кількість даних і при цьому не знижувати продуктивність додатку, оскільки дані кешуються та відображається лише їх видима частина. Крім того вона підтримує встановлення подій, за допомогою яких можна визначити, чи користувач досягнув кінця прокрутки. І якщо так, за допомогою уже реалізованого на сервері механізму пагінації виконати довантаження нових даних. Для цього було створено допоміжний сервіс, що виконує ці функції – Endless Scrolling Service.

Filter Panel (4) – панель, на якій розміщуються усі доступні фільтри. Є кнопки застосування фільтру або їх очищення. Вона розташовується у правій частині додатку.

Filter Multiple Select (5) – компонент множинного вибору. Ним представлені усі категорії програми.

Filter Chips (6) – компонент для пошуку інгредієнтів.

Post Card (7) – представляє картку із публікацією, де міститься зображення страви та коротка інформація про рецепт, кількість вподобань, збережень та коментарів до публікації.

Post Details – панель, що відображає детальну інформацію про рецепт, а саме: категорії, інгредієнти, кроки приготування, youtube-відео (якщо воно є) та коментарі. Контейнер коментарів також використовує технологію віртуалізації.

Далі реалізовано базовий компонент – діалогове вікно, та форми:

– Login Form – форма для авторизації користувача.
– Registration Form – форма для реєстрації
– Reset Password Form – форма відновлення паролю
– Restore Account Form – форма відновлення облікового запису
– Create Post Form – форма для створення публікації
– User Identity Form – форма для редагування облікового запису.

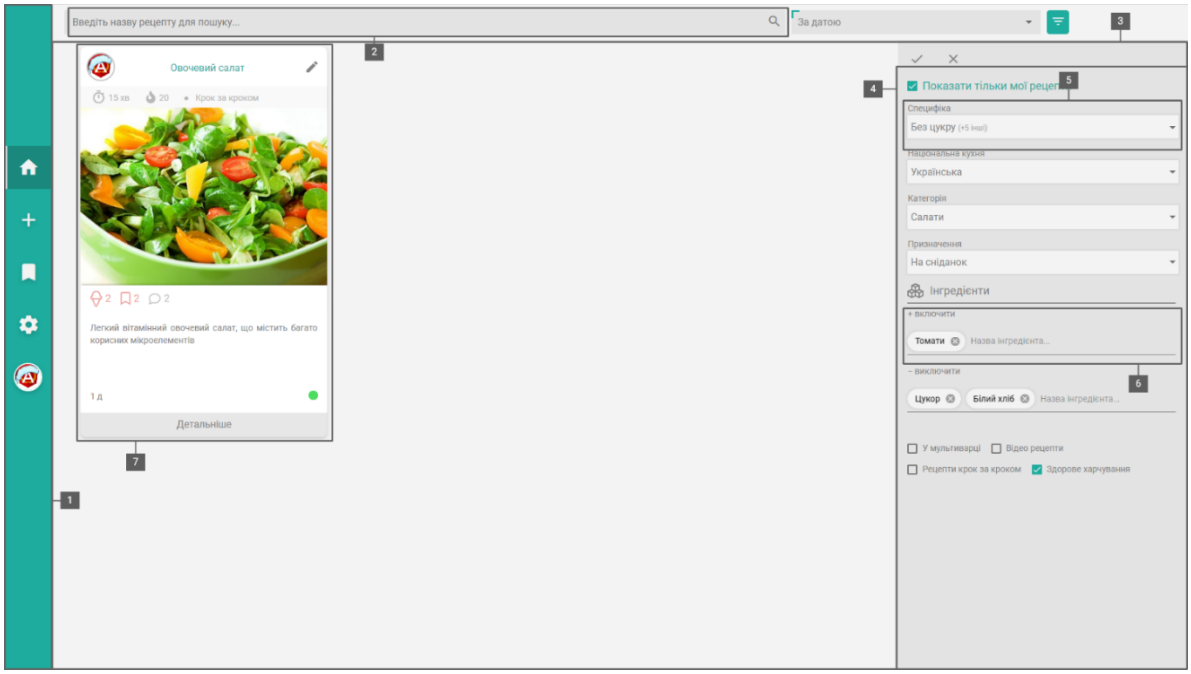


Рисунок 3.1 – Основні компоненти додатку

При натисненні на кнопку «Додати рецепт» на навігаційній панелі відкривається відповідне діалогове вікно, що показане на рисунку 3.2.

Далі на рисунках 3.2 – 3.4 наведено приклади основних форм.

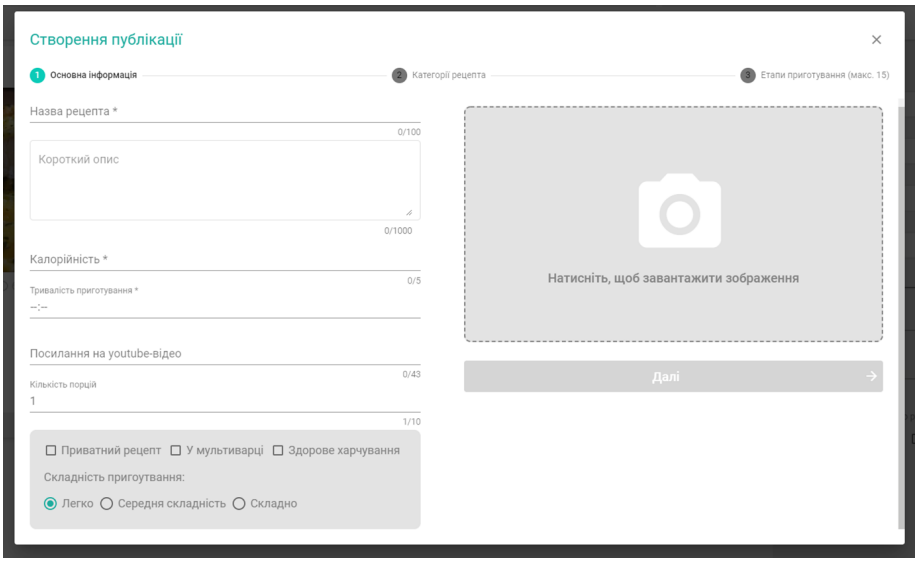


Рисунок 3.2 – Форма створення публікації

Якщо користувач не авторизований у системі, то усі кліки на елементи, що доступні тільки зареєстрованим користувачам будуть викликати діалогове вікно авторизації, що подане на рисунку 3.3.

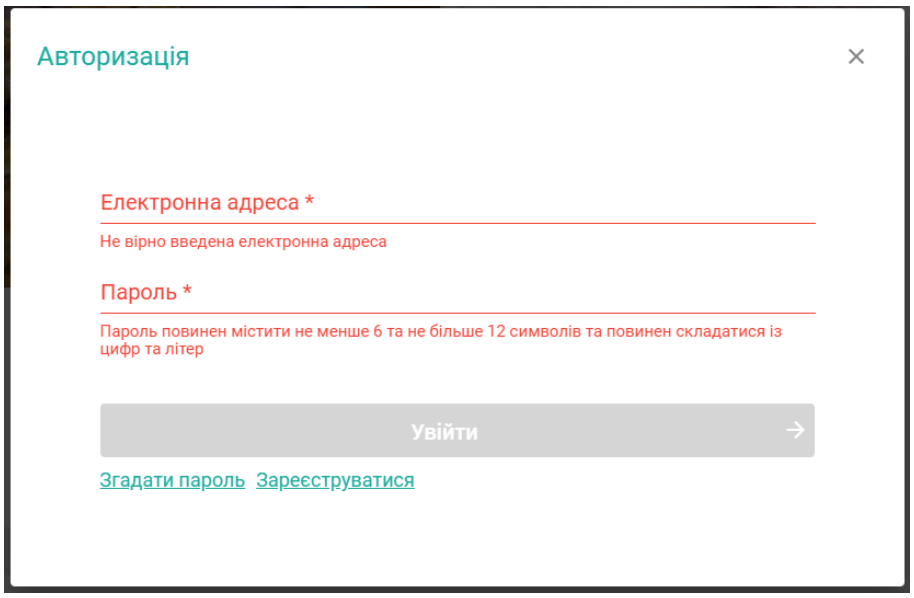


Рисунок 3.3 – Форма авторизації

Якщо користувач забув пароль від облікового запису, він може його змінити за допомогою натисненню на посилання «Змінити пароль». Якщо користувач ще не зареєстрований у системі, він може зареєструватись натиснуши на посилання «Зареєструватися». Відкриється вікно реєстрації, що подане на рисунку 3.4.

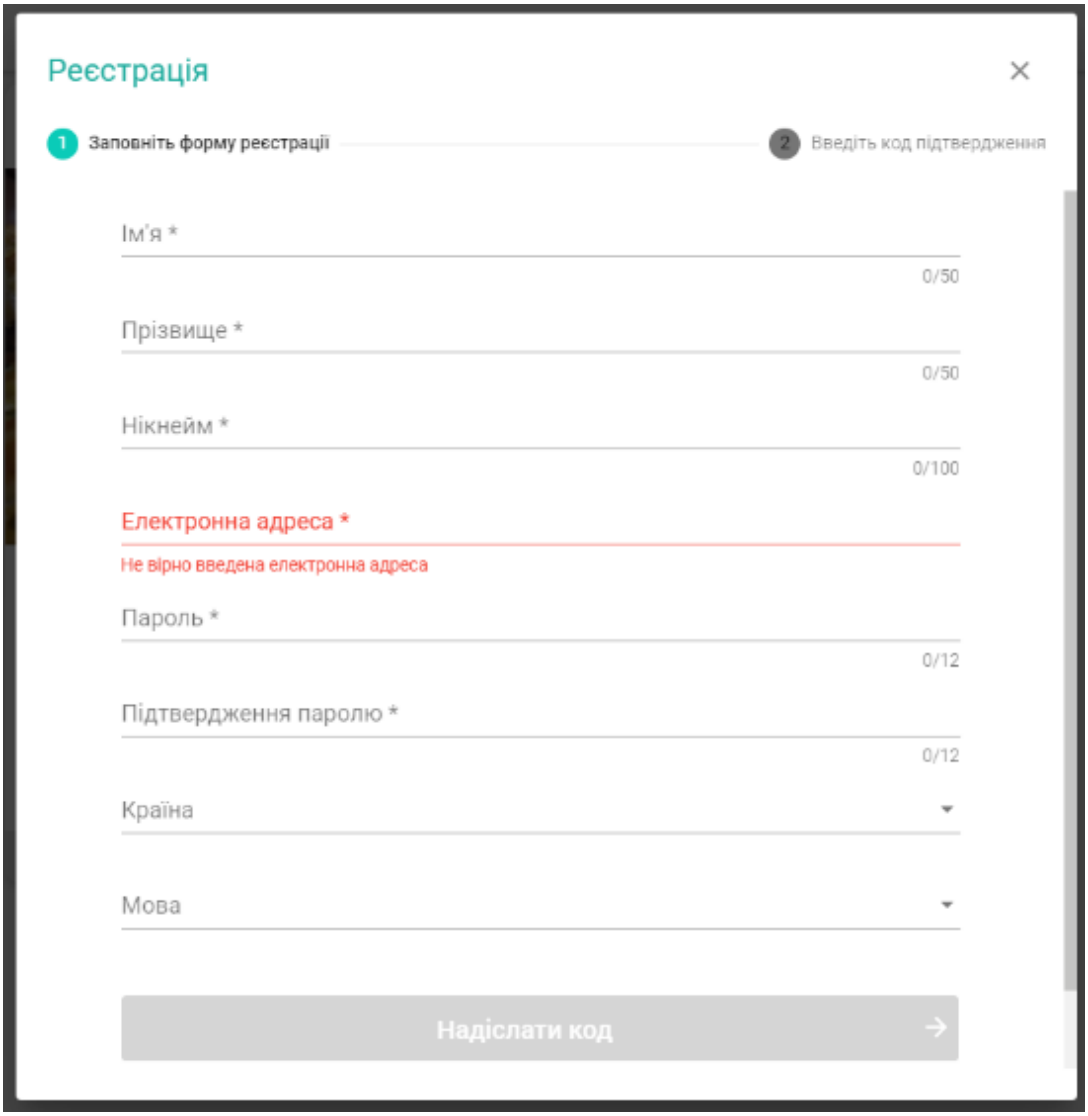


Рисунок 3.4 – Форма реєстрації користувача

Користувач має змогу редагувати облікові дані та змінювати зображення профілю. Для цього необхідно клікнути на іконку користувача, що знаходиться останньою на навігаційній панелі зліва. Відкриється модальне вікно, що подане на рисунку 3.5.

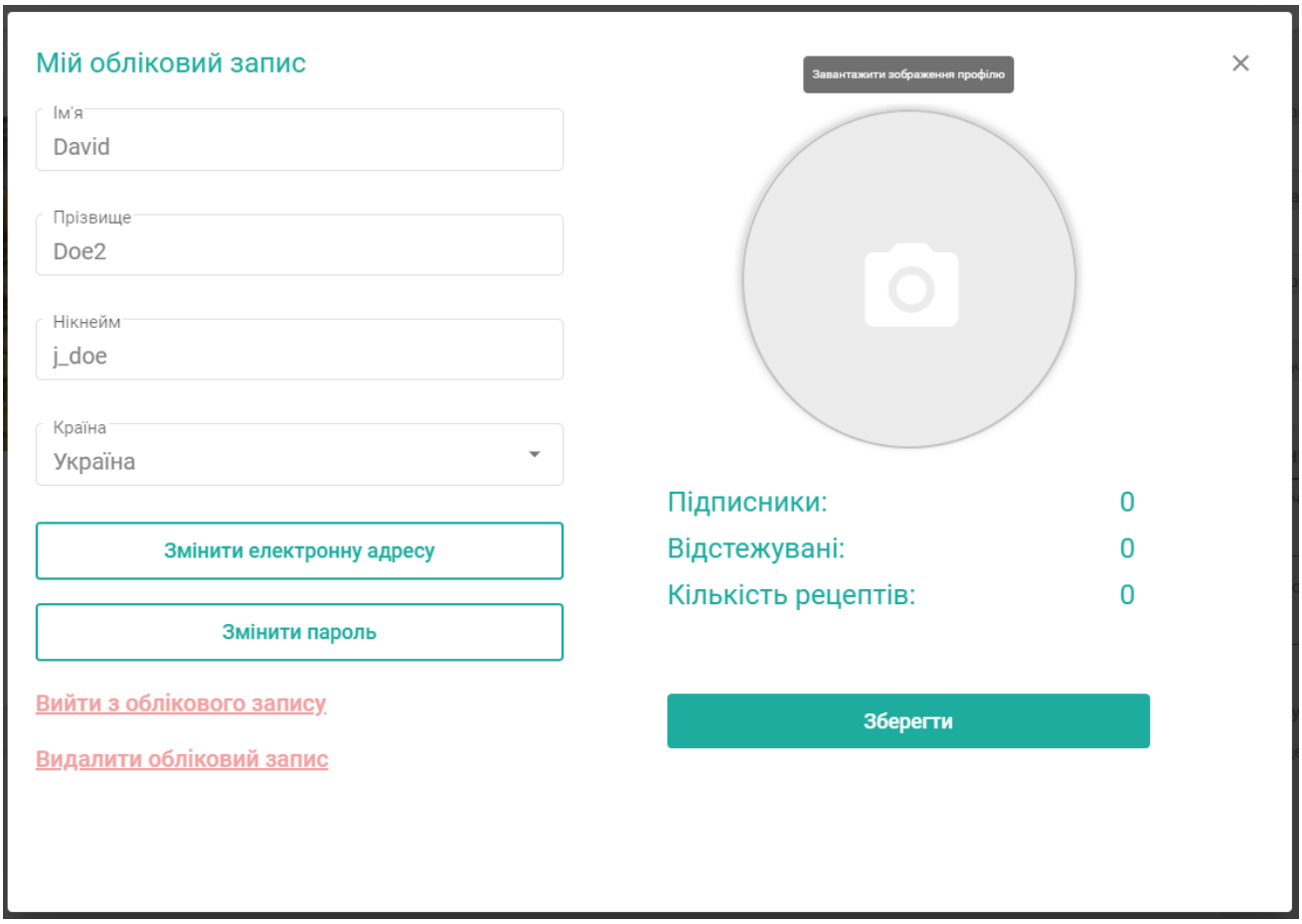


Рисунок 3.5 – Форма редагування облікового запису користувача

Користувач може вийти з облікового запису або видалити його, змінити електронну пошту та пароль.

Повернувшись на головну сторінку, при кліку на зображення або назву рецепту на картці рецепту відкриється вікно із детальною інформацією, що містить відео, якщо його завантажив користувач, список необхідних та опціональних інгредієнтів, короткий опис рецепту, етапи приготування та категорії, до яких належить рецепт. Також є можливість перегляду коментарів інших користувачів та створення своїх. Щоб створити коментар, необхідно написати текст та натиснути на кнопку із іконкою плюса. Для видалення коментарів достатньо натиснути на іконку кошика на самому блоці коментаря (рисунок 3.6). Крім того, якщо дуже часто створювати коментарі, система видасть попередження та тимчасово заблокує дію створення.

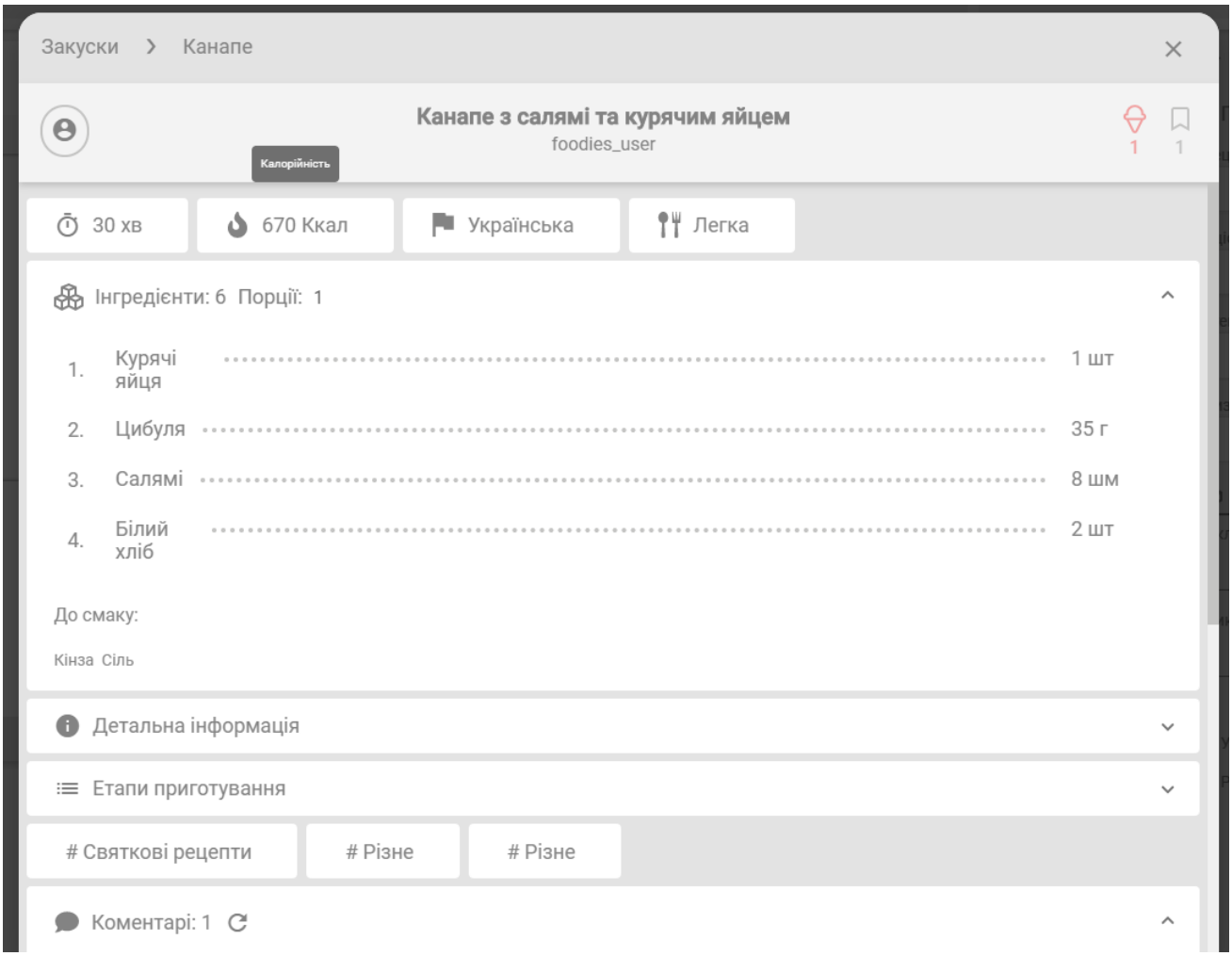


Рисунок 3.6 – Детальна інформація про рецепт

Інші форми та компоненти мають подібний вигляд.

Після завершення реалізації клієнтської частини, проект розгортається за допомогою серверу, усі відповідні мінімізовані файли розташовуються у спеціальній папці на сервері, що називається wwwroot.

Детальні фрагменти коду функцій та процедур бази даних та серверної частини програмної системи подано у додатку К.

Отже, після етапу проектування було реалізовано програмну систему та описано основні етапи її реалізації. Було створено базу даних відповідно до концептуальної моделі та виконано серверну реалізацію модульної та функціонально архітектури та створено клієнтську частину веб-додатку.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз та вибір методів тестування програмного забезпечення

Тестування програмного забезпечення – це процес виявлення дефектів програми, а також перевірки її працездатності у різних середовищах розгортання. Тестування є необхідною частиною процесу розробки програмної системи та дозволяє на критичних шляхах усунути небажані дефекти.

Виділяють функціональне та нефункціональне тестування. Функціональне тестування відповідає за працездатність усіх функцій програмної системи, що були визначені прецедентами. На основі діаграми варіантів використання створюються так звані тестові випадки (тест-кейси), що і формують набір тестів, які відповідають функціональної діяльності програмної системи.

До функціонального тестування також входить тестування безпеки додатку, що базується на перевірці основних принципів безпеки додатку. Це конфіденційність, цілісність та доступність даних. Також перевіряється ступінь захищеності додатку від шкідливих атак типу XSS, XSRF та можливих ін’єкцій у кодову базу програми. У наслідок тестування безпеки виявляються та усуваються критичні уразливості програмного забезпечення.

Функціональне тестування також може виконувати перевірку компонентів додатку на сумісність із іншими програмними модулями.

Нефункціональне тестування описує тести, що необхідні для виявлення характеристик програмного забезпечення, тобто як повинна працювати система. До нефункціональних тестів належить тестування навантаження, продуктивності, стрес-тести. Усі ці тести виміряють швидкодію програмної системи та дозволяються виявити та усунути ділянки, де відбувається зниження продуктивності додатку.

Окрім цього, тестування можна розподілити за рівнями.

Модульне тестування (або юніт-тестування) відповідає за перевірку роботи окремо взятих ізольованих модулів програмної системи. Це можуть бути класи,

функції та процедури у базі даних тощо. Модульне тестування здійснюється протягом усього життєвого циклу програмного забезпечення.

Системне тестування виконується для перевірки як функціональних, так і нефункціональних вимог у цілому. Програмна система піддається комплексному тестування, тобто відбувається тестування як окремих компонентів (юніт-тестування), так і зв'язаних компонентів, тестуючи при цьому використання ресурсів системи, комбінації даних користувацького вводу, зручність використання тощо. При такому тестуванні складається список тест-кейсів, що відповідають прецедентам системи.

На рівні інтеграційного тестування програмного забезпечення найбільше приділяють увагу взаємодії між компонентами системи. Наприклад, якщо серверний стиль архітектури – мікросервіси, то кожен із сервісів тестується спочатку окремо і у разі успішного проходження тестів виконується тестування на те, як мікросервіси працюють разом. Крім того взаємодія модулів системи одразу тестується на двох рівнях: компонентному – тобто тестується взаємодія компонентів усередині одної системи, та системному – тестується взаємодія уже окремих систем, як у випадку мікросервісної архітектури. Також інтеграційне тестування дозволяє виявити проблеми взаємодії системи із операційною системою або обладнанням.

Порядок інтеграційного тестування також буває різний. Існує тестування «Знизу вверх», «Зверху вниз» та «Великий вибух». Тестування «Знизу вверх» передбачає компонування усіх процедур та функцій та тестування їх незалежно одну від одної. Після цього створюється наступний рівень – компонуються структури та класи програмної системи і також тестуються разом. Але такий підхід є корисним лише у тому випадку, коли практично усі модулі є розробленими та готовими до використання та взаємодії. Якщо існує багато компонентів, що лише на стадії проектування, тобто не є реалізованими, то для такої системи краще застосувати тестування «Зверху вниз». Воно виконується шляхом перевірки роботи спочатку модулів верхнього рівня, тоді як низькорівневі модулі замінюються «заглушками» – тобто об'єктами, що імітують

функціонал справжніх модулів. Після цього, по мірі реалізації модулів об’єкти-«заглушки» замінюються реальними компонентами.

«Великий вибух» передбачає проведення тестування усіх модулів разом, що дозволяє зекономити час. Однак, якщо тест-кейси та їх результати записано не правильно, то процес інтеграції може значно ускладнитись.

Останній рівень – приймальне тестування – виконується з метою виявлення чи задовольняє система приймальним критеріям та чи досяг продукт, що тестувався, якості необхідного рівня.

Так як розроблювана програмна система є клієнт-серверним додатком, то проводити тестування необхідно як серверної, так і клієнтської частини та системи в цілому. Для цього було вибрано метод тестування «Знизу вверх», тобто перед тестуванням усіх компонентів системи разом, виконується тестування кожного компоненту окремо. Це дозволяє при створенні нового модуля програми бути впевненим, що усі модулі, які були створені раніше, працюють нормально, що скорочує час на перевірку працездатності системи у разі введення нових логічних компонентів, що потенційно можуть призвести до дефектів програми.

4.2 Тестування серверної частини програмної системи

При тестуванні програмної системи для автоматизації публікації, поширення та пошуку кулінарних рецептів було запропоновано використовувати тестування «Знизу вверх», оскільки практично усі модулі програми реалізовані.

Тестування серверної частини проводиться з метою виявлення дефектів у критично найважливішій частині клієнт-серверного додатку, оскільки основна функціональність, засоби безпеки та алгоритми керування навантаженням реалізовані саме на сервері.

Тестування було проведено поступово: від низькорівневих модулів до високорівневих компонентів. При тестуванні було використано бібліотеку NUnit та базу даних у пам’яті для симуляції реальної БД.

Для усіх тестів у проекті створено окрему директорію Test Projects, де містяться проекти для тестування функцій серверної частини додатку. Також там містяться основні налаштування тестового середовища, до якого входять об'єкти-«заглушки», наприклад, класи, що імітують поведінку ORM-системи Entity Framework. Такі об'єкти називаються «моками», а функціональність, що вони виконують ідентична функціональності реальних об'єктів. Службові класи оформлені у вигляді синглтонів. Також створено базовий клас BaseTestClass, де міститься усі загальні налаштування для тестів.

Спочатку було виконане модульне тестування об'єктів запиту. Для цього було створені мок-об'єкти, що симулювали поведінку реальних сутностей бізнес-моделі. Лістинг типового тест-кейсу модульного тестування об'єкту запиту створення користувача подано у лістингу:

```
[Test]
public async Task CreateUsersWithCorrectData()
{
  Guid id = Guid.NewGuid();

  User user = UserMock.Instance.UserForAdd;
  UserIdentity userIdentity = UserMock.Instance.UserIdentityForAdd;
  user.UserId = id;
  userIdentity.Id = id;

  await QueryExecutor.Execute(new CreateUser(userIdentity, user));

  User createdUser = await QueryExecutor.GetSingleBy<User>(x =>
                              x.UserId == id);

  Assert.IsNotNull(createdUser);
  Assert.AreEqual(id, createdUser.UserId);
}
```

Визначення, чи метод відпрацював коректно лежить у перевірці ідентифікатора створеного користувача, що має відповідати ідентифікатору, який був згенерований раніше, та, взагалі, чи існує щойно створений користувач у базі даних. Якщо так, значить метод виконався без помилок. QueryExecutor та Context зареєстровані у базовому класі. Усі інші тести мають подібну структуру.

Після тестування сервісів, було виконано тестування контролерів, а саме, кінцевих точок програми.

Для цього було використано аналогічні засоби, окрім того було створено «заглушки» об’єктів DTO та класів, що відповідають за роботу із протоколом HTTP, такі, як HttpContext та подібні.

Для перевірки роботи контролерів, а саме кінцевих точок програми, використовується підхід симулювання запитів. Для цього було використано бібліотеку HttpMock. Вона надає функціонал по управлінню симульованими запитами до сервера, проте, з метою уникнанення впливу на дані з БД, було створено «заглушки» контексту даних та виконано перевірку методів дій у контролерах програмної системи.

4.3 Системне тестування та верифікація програмного забезпечення

Після проведення тестування серверної частини, було виконано за подібним принципом модульне тестування клієнтської частини та повне автоматизоване тестування системи.

При модульному тестуванні клієнтської частини у додатку із використанням платформи Angular зазвичай тестуються окремі компоненти та функціональні одиниці, такі як, сервіси, класи-трансформатори, допоміжні класи та класи бізнес-логіки. При цьому логіка роботи юніт-тестів на клієнті не відрізняється від роботи модульних тестів на сервері. Дані для компонента підміняються та перевіряється, чи компонент у робочому стані та виконує свої функції. Для сервісів, що використовують http-клієнт, виконується його заміна на копію, що не посилає дані на сервер, а лише симулює поведінку відправки http-запитів за допомогою вбудованих засобів. Модульне тестування проведене за допомогою фреймворку для запуску юніт-тестів Karma та бібліотеки для їх виконання – Jasmine. Усі тести починаються із блоку describe, що описує акт тестування та it, що у свою чергу описує сам тест. Приклад тесту компонента, який відповідає за відображення картки публікації, що перевіряє, чи компонент завантажений, поданий далі.

```
describe('PostCardComponent', () => {
   let component: PostCardComponent
   let fixture: ComponentFixture<PostCardComponent>


beforeEach(async () => {
    await TestBed.configureTestingModule({
        declarations: [ PostCardComponent ]
}).compileComponents()
       })


beforeEach(() => {
  fixture = TestBed.createComponent(PostCardComponent)
   component = fixture.componentInstance
        fixture.detectChanges()
   })


it('should create', () => {
      expect(component).toBeTruthy()
})
```

Метод beforeEach описує конфігурацію, що повинна бути встановлена перед кожним тест-кейсом шляхом впровадження залежностей.

Після виконання модульних тестів виконується реалізація тестів End-To-End (або e2e). Вони тестують систему повністю. Для цього використовується окреме середовище, призначене саме для тестування з метою уникнення змін даних у звичайному середовищі розробки. Для реалізації тестування e2e використано фреймворк Protractor. Він виконує автоматизоване тестування програмної системи за допомогою симуляцій дій користувача. Також при автоматизованому тестуванні використовується патерн проектування Page Object. Цей клас містить посилання на вузли html-дерева із метою отримання даних про елемент, що використовується у тестуванні. Об’єкти-сторінки створюються для кожного компонента, функціональність якого була перевірена шляхом проведення юніт-тестування. Для запуску тестів Protractor використовує спеціальний сервер, що має назву Selenium та може запускатися як локально, так і на віддаленому сервері.

Після проведення тестування було створено таблицю, що містить загальний вигляд структури тест-кейсів. Вона подана у табл. 4.1.

Таблиця 4.1 – Основні тест-кейси

Вимоги	Модуль додатку	Підмодуль додатку	Вихідні дані	Очікуваний результат
1	2	3	4	5
R1	Модальне вікно	Форма авторизації	1. Клацнути на кнопку авторизації. 2. Ввести логін та пароль 3. Натиснути на кнопку.	1. З’являється модальне вікно та форма авторизації. 2. Валідація даних 3. Вікно закривається, сторінка перезавантажується, доступні усі дії для зареєстрованого користувача в залежності від його ролі.
R2	Модальне вікно	Форма реєстрації	1. Клацнути на кнопку авторизації. 2. Перейти на форму реєстрації по відповідному посиланню 3. Ввести необхідні особисті дані. 4. Натиснути кнопку «Надіслати код». 5. Вставити надісланий код. 6. Натиснути на кнопку «Завершити реєстрацію».	1. З’являється модальне вікно, з’являється форма авторизації, 2. З’являється форма реєстрації. 3. Валідація даних 4. Сервер відправляє код на пошту користувача. 5. Валідація коду 6. Закривається вікно, перезавантажується сторінка, доступні усі дії для зареєстрованого користувача в залежності від його ролі.
R3	Модальне Вікно	Форма публікації рецептів	*Підготовка: зберегти картинку страви на робочому столі.* 1. Клацнути на пункт меню «Додати рецепт». 2. Заповнити форму основною інформацією про рецепт. 3. Клацнути на кнопку завантаження картинки рецепту. 4. Вибрати підготовлений файл зі списку. 5. Натиснути кнопку «Далі».	1. З’являється модальне вікно та форма з публікацією рецептів. 2. Валідація даних форми основних даних про рецепт. 3. Відкривається вікно із вибором зображення. 4. Зображення відображається у формі. 5. З’являється форма категорій рецепту. 6. Валідація даних. 7. Валідація таблиці інгредієнтів. 8. З’являється форма створення етапів пригоутвання.

Продовження таблиці 4.1

1	2	3	4	5
R3	Модальне Вікно	Форма публікації рецептів	6. Заповнити форму категорій рецепту. 7. Заповнити таблицю з інгредієнтами. 8. Натиснути кнопку «Далі». 9. Заповнити форму із етапами приготування. 10. Натиснути кнопку «Завершити публікацію».	9. Валідація даних. 10. Вікно закривається, втсановлюється фільтр «Мої рецепти» та оновлюється стрічка публікацій.
R4	Стрічка публікацій	Збережені рецепти	1. Клацнути на пункт меню «Збережені рецепти».	1. Панель фільтрів закривається, відкривається панель із книгами рецептів
R5	Модальне вікно	Редагування облікового запису	*Підготовка: зберегти профільне зображення на робочому столі.* 1. Клацнути на пункт меню «Мій обліковий запис». 2. Редагувати облікові дані. 3. Клацнути на кнопку завантаження профільного зображення. 4. Вибрати підготовлений файл. 5. Натиснути кнопку «Зберегти дані».	1. Відкривається модальне вікно із формою редагування облікових даних. 2. Валідація даних 3. Відкривається вікно для вибору зображення. 4. Вибраний файл відображається на формі. 5. Дані посилаються на сервер, вікно закривається та сторінка перезавантажується
R6	Стрічка публікацій	Панель фільтрів	1. Клацнути на кнопку «Показати панель фільтрів». 2. Вибрати фільтри. 3. Натиснути кнопку. «Застосувати фільтри». 4. Натиснути кнопку «Очистити фільтри». 5. Натиснути кнопку «Застосувати фільтри».	1. Відображається панель фільтрів справа. 2. Відображаються відповідні фільтри. 3. Стрічка фільтрується та оновлюється. 4. Поля на панелі фільтрів очищуються. 5. Публікації відображаються у звичайному режимі.

Кінець таблиці 4.1

1	2	3	4	5
R7	Адмін. панель	Створення категорій	1. Клацнути на кнопку «Мій обліковий запис» 2. Авторизуватися у ролі модератора 3. Вибрати потрібну категорію 4. Ввести дані нової категорії 5. Створити категорію	1. З’являється модальне вікно із формою авторизації. 2. Вікно закривається, відбувається переміщення на сторінку адміністративної панелі. 3. Таблиця із категоріями фільтрується відповідно до вибраної. 4. Валідація даних 5. Дані додаються у таблицю

Так як програмна система буде вивантажена на платформу Amazon Beanstalk, то для коректності роботи програми було проведено так зване поверхневе тестування. Воно має назву «Smoke Testing» та застосовується у тому випадку, коли необхідно протестувати лише критично важливі частини програмної системи. Зазвичай, достатньо запустити деякі автоматизовані e2e тести для визначення дефектів та їх виправлення у віддаленому середовищі.

Після виконання тест-кейсів було створено звіт про результати тестування, що подано у таблиці 4.2.

Таблиця 4.2 – Звіт про результати виконання деяких тест-кейсів

ID	Опис тест-кейса	Вхідні значення	Реальні вихідні дані	Пройдено
1	2	3	4	5
TC1	Авторизація у системі	1. Логін користувача 2. Пароль користувача	1. Токен доступу та оновлення не пусті та валідні 2. Наявна інформація про користувача вірна	Так, Очікувані та реальні вихідні дані однакові
TC2	Реєстрація користувача	1. Ім’я 2. Прізвище 3. Нікнейм 4. Ел. адреса 5. Пароль 6. Країна	1. Токен доступу та оновлення не пусті та валідні	Так, Очікувані та реальні вихідні дані однакові

Кінець таблиці 4.2

1	2	3	4	5
ТС2	Реєстрація користувача	7. Мова 8. Код доступу	2. Отримана інформація про користувача не порожня та відповідає даним з форми реєстрації	Так, Очікувані та реальні вихідні дані однакові
ТС3	Публікація рецепту	1. Назва рецепту 2. Короткий опис 3. Калорійність 4. Тривалість Приготування 5. Посилання на відео 6 Додаткова інф. 9. Зображення рецепту 10. Категорії рецепту 11. Інгредієнти 12. Етапи приготування	1. Створена публікація містить усю необхідну інформацію, що відповідає інформації, що створив користувач 2. Встановлений фільтр на відображення тільки рецептів поточного користувача	Так, Очікувані та реальні вихідні дані однакові
ТС1 Inv	Авторизація у системі	1. Неправильний логін користувача 2. Неправильний Пароль користувача	1. Токени доступу не надсилаються 2. Відображається помилка «Користувача не знайдено»	Так, Очікувані та реальні вихідні дані однакові

У таблиці 4.2 наведені деякі із основних результатів проходження тест-кейсів. Було проведено тестування із різними наборами даними. Усі тести виконались успішно.

Також було проведено оптимізаційне тестування системи, що полягає у вимірюванні навантаження на базу даних та окремі її компоненти, наприклад, таблиці. Для цього було використано планувальник запитів та досліджено більшість запитів та функцій на продуктивність. Планувальник показує, скільки часу виконується запит та дозволяє виявити неоптимальні запити. У процесі тестування було оптимізовано деякі запити із метою збільшення продуктивності програмної системи.

Також було протестовано роботу Redis-сховища даних, а саме, доведено ефективність та доцільність використання кешування у додатку.

Отже, у результаті виконання тестування програмної системи для автоматизації публікації, поширення та пошуку кулінарних рецептів виявлено, що усі тести успішно проходять. Дефекти, що було виявлено під час тестування було успішно усунено. Базова функціональність додатку реалізована у відповідності з вимогами до ПЗ, та більша частина є повністю працездатною.

ВИСНОВКИ

На сьогодні соціальні мережі є популярними сервісами, що дозволяють користувачам обмінюватися інформацією. В залежності від типу, соціальні мережі можуть виконувати різні функції, проте базові – це пошук та розповсюдження контенту. Вузько-тематичні соціальні мережі об’єднують людей за спільними інтересами та спеціалізуються на конкретній темі. Зазвичай вони не досить популярні, проте можуть бути корисними, особливо, якщо вони допомагають задовольняти повсякденні людські потреби.

Так, темою дипломного проекту є програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів. На підставі виконаного дослідження та комплексного аналізу у галузі соціальних мереж та програмних систем було визначено основні переваги і недоліки існуючого програмного забезпечення та встановлено, що на сучасному ринку є зовсім небагато кулінарних соціальних мереж, а наявні програмні рішення – звичайні веб-сайти із відсутністю характерного для соцмереж функціоналу та не зовсім привабливим і сучасним користувацьким інтерфейсом, що підкреслює актуальність теми розроблюваного програмного забезпечення. На основі отриманих даних були визначені функціональні та нефункціональні вимоги до програми та описано основні варіанти використання розроблюваного додатку.

На етапі проектування програмного забезпечення було проаналізовано та визначено переваги та недоліки монолітного і мікросервісного стилю архітектури ПЗ. Встановлено, що мікросервіси – вдалий вибір для реалізації потенційно високонавантажених системи, проте було обрано монолітний стиль серверної архітектури, керуючись типом продукту, термінами розробки та складністю впровадження системи.

При детальному проектуванні було досліджено переваги та недоліки різних типів сховищ даних, обрано реляційну систему керування базами даних у якості основного сховища, керуючись цілісністю, надійністю, узгодженістю та швидкістю доступу до даних.

У процесі проектування моделі «Сутність-зв’язок» було визначено сутності системи та їх атрибути, а також здійснено нормалізацію даних та приведення відношень до третьої нормальної форми, проте деякі відношення було частково денормалізовано із метою уникнення зниження продуктивності.

При аналізі наявних реляційних систем керування базами даних було визначено основні характеристики популярних засобів розробки та вибрано СКБД PostgreSQL, керуючись такими важливими критеріями, як швидкодія, цінова політика та різноманітність функцій по управлінню масивами даних.

Також було встановлено особливості тришарової архітектури, спроектовано систему пакетів даних та об’єктної моделі класів. Визначено основні переваги та недоліки популярних патернів проектування сервісного шару додатку та встановлено, що шаблон «Об’єкт запиту» – є вдалим вибором для його реалізації завдяки можливості виконання нетривіальних запитів, гнучкості, легкості у тестуванні.

При проектуванні функціональної архітектури було проаналізовано відповідні вимоги до програмного забезпечення, встановлено потенційні ділянки падіння продуктивності програми та їх усунено за допомогою інструментів по управлінню високонавантаженими системами. Для цього було використано дистрибутивну систему тимчасового сховища даних Redis та визначено основні стратегії кешування у контексті конкретних проектних рішень.

У процесі виконання проекту інтерфейсу користувача було виділено основні розділи та модальні вікна програмної системи, а також виконано порівняння сучасних засобів реалізації клієнтської частини додатку та вибрано платформу Angular у якості фронтенд-фреймворка.

У процесі реалізації програмної системи було створено базу даних відповідно до концептуальної моделі, систему індексів та спроектовано базові запити до БД. Після цього було виконано серверну реалізацію модульної та функціональної архітектури. У процесі створення клієнтської частини веб-додатку було реалізовано основні компоненти та сервіси програми, що відповідають спроектованій системі модальних вікон та розділів.

Під час тестування програмного забезпечення було підготовлено основні тест-кейси, виявлено та усунено дефекти роботи програми і в результаті сформовано звіт про результати виконання тестових випадків. Виконано модульне тестування серверної та клієнтської частини, а також повне системне тестування за допомогою створення автоматизованих тестів.

Отже, у результаті виконання дипломного проекту реалізовано програмну системи для автоматизації публікації, поширення та пошуку кулінарних рецептів. Базова функціональність додатку реалізована у відповідності з вимогами до ПЗ, а результати практичної апробації підтверджують працездатність програми.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Социальные сети [Електронный ресурс] / Интернет-маркетинг с нуля. – Режим доступа к ресурсу: https://tilda.education/courses/marketing/social-networks

2. «Третья волна» социальных сетей [Електронный ресурс] / Социальные сети. – Режим доступа к ресурсу: https://vc.ru/flood/1230-tretya-volna-sotsialnyih-setey

3. Реклама в социальных сетях [Електронный ресурс] / Агенство маркетинга. – Режим доступа к ресурсу: https://cyborg-studio.com/reklama-v-socialnyh-setjah

4. Ших К. Эра Facebook. Как использовать возможности социальных сетей для развития вашего бизнеса / Клара Ших. – М.: Манн, Иванов и Фербер, 2011. – 304 с.

5. Рецепти. Кулінарні рецепти з фото: перші і другі блюда, салати, випічка, десерти, коктейлі [Електронний ресурс] / Рецепти. – Режим доступу до ресурсу: http://cook.i.ua

6. Simply Recipes [Electronic resource] / Recipes // Elise Bauer. – Available from: https://www.simplyrecipes.com

7. Супы [Електронный ресурс] / 701 рецептов приготовления пошагово с фото на Merry Kitchen. – Режим доступа к ресурсу: https://merry-kitchen.com/recepty/supy

8. Клієнт-серверна архітектура [Електронний ресурс] / Техінчні статті. – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical-articles/client-server-architecture

9. Rodolfo Machado and Rodolphe el-Khoury. Monolithic Architecture (Architecture & Design). – M.: Prestel Pub; First edition, November 1, 1995

10. dblp: Advanced Transaction Models and Architectures [Electronic resource]/ Computer Science bibliography // Larry Kerschberg. – Available from: https://dblp.uni-trier.de/db/books/collections/JajodiaK97.html

11. Применение микросервисной архитектуры: плюсы, минусы, подводные камни [Електронный ресурс] / Блог Highload // Алиса Яковлева. – Режим доступа к ресурсу: https://simpleone.ru/blog/primenenie-mikroservisnoj-arhitektury-plyusy-minusy-podvodnye-kamni

12. Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen. Microservice Architecture: aligning principles, practices, and culture. – M.: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, June 2016

13. Server Architectures [Electronic resource] / Concurrent Programming for Scalable Web Architectures // Benjamin Erb. – Available from: http://berb.github.io/diploma-thesis/original/042_serverarch.html

14. S. Sumathi, S. Esakkirajan. Fundamentals of Relational Database Management Systems. – M.: Springer-Verlag, Berlin, Heidelberg, 2007

15. What is NoSQL? Nonrelational Databases, Flexible Schema Data Models [Electronic resource] / AWS. – Available from: https://aws.amazon.com/nosql/?nc1=h_ls

16. What Is a Graph Database? [Electronic resource] / AWS – Available from: https://aws.amazon.com/nosql/graph/?nc1=h_ls

17. A Comparison Of Relational Database Management Systems [Electronic resource] / DigitalOcean // Mark Drake. – Avalilable from: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems

18. S. Sumathi, S. Esakkirajan. Fundamentals of Relational Database Management Systems. – M.: Springer-Verlag, Berlin, Heidelberg, 2007

19. About PostgreSQL [Electronic resource] / PostgreSQL. – Avalilable from: https://www.postgresql.org/about

20. .NET Fundamentals [Electronic resource] / .NET Documentation. – Available from: https://docs.microsoft.com/en-us/dotnet/fundamentals

21. Introduction to ASP.NET Core [Electronic resource] / Microsoft Docs. – Available from:

https://docs.microsoft.com/en-us/aspnet/core/introduction-to aspnetcore?view=aspnetcore-5.0

22. Atul Gupta, Sudhanshu Hate, Andrew Siemer. ASP.NET 4 Social Networking . – M.: Packt Publishing, 2011

23. Is the repository pattern useful with Entity Framework Core? [Electronic resource] / The Reformed Programmer // Jon P Smith. – Avalilable from: https://www.thereformedprogrammer.net/is-the-repository-pattern-useful-with-entity-framework-core

24. Caching strategies [Electronic resource] / Amazon ElastiCache. – Available: https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html

25. Redis Documentation [Electronic resource] / Redis. – Available from: https://redis.io

26. Pros and Cons of ReactJS [Electronic resource] / javatpoint. – Available from: https://www.javatpoint.com/pros-and-cons-of-react

27. Pros and Cons of Angular Development Framework [Electronic resource] / AltexSoft. – Available from: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of- angular-development

28. Typed JavaScript at Any Scale [Electronic resource] / TypeScript. – Available from: https://www.typescriptlang.org

29. What is Angular? [Electronic resource] / Angular. – Available from: https://angular.io/guide/what-is-angular

30. The RxJS library [Electronic resource] / Angular. – Available from: https://angular.io/guide/rx-library

31. Different Testing Types with Details [Electronic resource] / Types of Software Testing. – Available from: https://www.softwaretestinghelp.com/types-of-software-testing

32. Protractor – end-to-end testing for AngularJS [Electronic resource] / Protractor documentation. – Available from: https://www.protractortest.org

ДОДАТОК А

(обов’язковий)

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

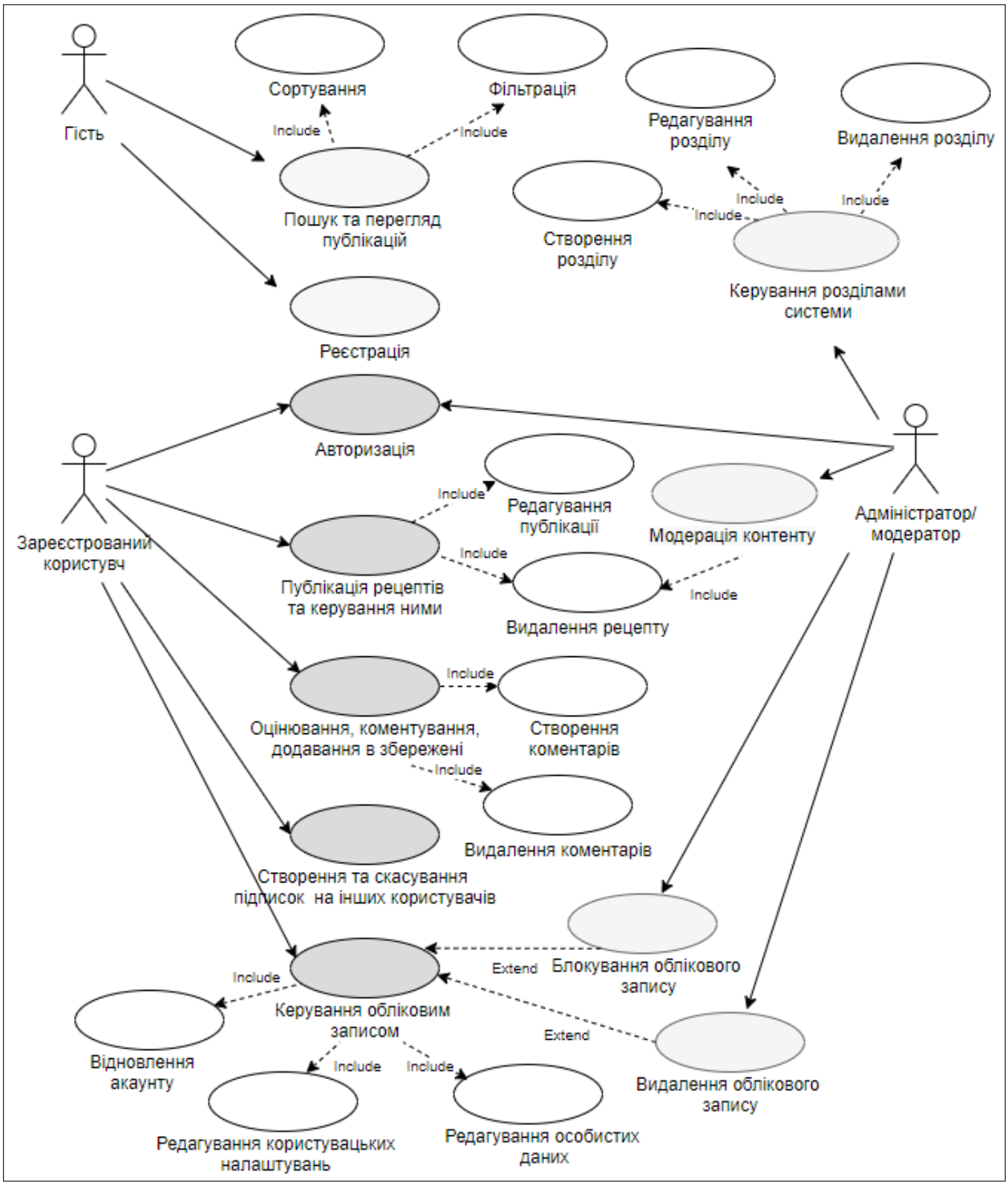


Рисунок А.1 – Діагарма варіантів використання

ДОДАТОК Б

(обов’язковий)

ТЕХНІЧНЕ ЗАВДАННЯ

Введення

Робота виконується в рамках проекту розробки соціальної мережі кулінарних рецептів, що дозволяє шукати, переглядати, створювати та поширюватипублікації рецептів, опублікованих іншими користувачами.

1 Підстава для розробки

Підставою для розробки є «Завдання на дипломний проект», затверджене завідувачем кафедри інженерії програмного забезпечення. Найменування розробки: програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів.

2 Призначення розробки

Соціальна мережа призначена для перегляду, публікації та поширення кулінарних рецептів.

Користувачами системи є звичайні користувачі Інтернету.

Функціональне призначення передбачає: пошук, фільтрацію та сортування рецептів, публікацію кулінарних рецептів, редагування та видалення створених публікацій, коментування, оцінювання рецептів інших користувачів, додавання рецептів у збережені, можливість формувати книги зі збережених рецептів, реєстрація та авторизація користувачів, редагування даних облікового запису, видалення та відновлення облікового запису.

Додавання нових розділів системи, редагування або видалення існуючих, модерація користувацького вмісту, тобто перевірка контенту на цензурність, доцільність та плагіат відбувається адміністраторами та модераторами. Крім того, адміністратор може заблокувати або видалити обліковий запис користувача.

Щодо експлуатаційного призначення, система може використовуватися на будь-якому персональному комп’ютері, у будь-якому встановленому браузері, ніяких додаткових налаштувань не потребує.

3 Вимоги до програми

3.1 Вимоги до функціональних характеристик

Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів має забезпечувати функції, подані нижче.

Функція публікації рецептів. Рецепт може бути опублікований тільки тоді, коли модератор перевірить відповідність публікації цензурі та кулінарній тематиці і допустить до публікації. Якщо вміст посту порушує ці правила, модератор повинен не допустити рецепт до публікації та видалити його.

Функція редагування та видалення опублікованих рецептів. Користувач має змогу редагувати рецепт не пізніше 15 хвилин після публікації. Видалену публікацію відновити неможливо.

Реєстрація та авторизація в системі.

Редагування особистих даних. Передбачити зміну паролю та електронної пошти користувача.

Видалення та відновлення акаунту. Акаунт, що видалив користувач можна відновити протягом 6 місяців.

Пошук, фільтрація та сортування кулінарних рецептів. Програмна система повинна забезпечувати користувача пошуком по назві рецепта. Фільтрація має бути доступна по таким категоріям: Специфіка (Вегетаріанство, Безглютенова дієта тощо), Категорія (Супи, Салати тощо), Підкатегорія (Борщ, Фруктовий салат тощо), Національна кухня, Призначення (Дітям, На обід тощо). Також передбачити функції «Включити інгредієнти», для пошуку рецептів, в яких містяться саме вибрані користувачем інгредієнти та «Виключити інгредієнти», для пошуку тих, в яких вибраних інгредієнтів немає. Реалізувати фільтри по часу приготування (швидко, довго), калорійності (слабко калорійний, сильно калорійний), складності приготування (легко, середня складність, важко) та реалізувати спеціальні фільтри «Здорова їжа», «Приготування у мультиварці», «Покроковий рецепт», «Відео-рецепт». Сортування рецептів має бути реалізоване за датою (спочатку нові), за кількістю вподобань (спочатку рецепти із найкращим рейтингом), за часом

приготування (спочатку рецепти, що займають мало часу, спочатку рецепти, що займають багато часу).

Збереження рецептів, та створення книг рецептів.

Оцінювання рецептів інших користувачів.

Коментування публікації користувачів.

Створення підписок на інших користувачів. Також передбачити пошук користувачів по імені.

Повинна бути реалізована адміністративна панель, до якої мають доступ тільки адміністратори та модератори. У цьому розділі системи повинна бути можливість створення, редагування та видалення розділів програми. Також має бути реалізована панель керування публікаціями користувачів.

Має бути реалізоване ліве навігаційне меню, що буде включати такі розділи веб-додатку: «Головна сторінка», «Створити публікацію», «Особистий кабінет», «Налаштування програми». Крім того, для не авторизованих користувачів мають бути доступні лише пункти «Головна сторінка» та «Авторизуватися у системі», а при здійснені кліку на пункти, що доступні лише авторизованим користувачам має відображатись діалогове вікно для авторизації у системі.

Центральна частина повинна містити верхнє прикріплене меню, де міститься поле пошуку рецептів, можливість сортування та кнопка відкриття панелі фільтрів, що при відкритті має розташовуватися справа. Публікація рецептів повинна бути оформлена у вигляді картки де міститься назва страви, зображення користувача, який опублікував рецепт, фото страви, скорочено поданий список властивостей страви (калорійність, час приготування тощо), кількість вподобань, коментарів та збережень, короткий опис рецепту, дата публікації та кнопка «Детальніше».

При кліку на кнопку «Детальніше» повинне відкриватися модальне вікно, де детально розписаний рецепт. Також там мають міститися коментарі та можливість створення і видалення своїх коментарів.

У розділі «Створити публікацію» реалізувати форму та валідацію даних форми. На формі повинні розміщуватись такі поля:

– Текстове поле «Назва рецепту»;

– Текстове поле «Короткий опис рецепту»;

– Поле для завантаження фото рецепту. Фото повинно бути розміром не більше 5Мб.

– Поля вибору, відповідні фільтрам, які були наведені вище;

– Поле для введення інгредієнтів та їх даних (назва, кількість)

– Кнопка «Опублікувати рецепт».

При кліку на кнопку опублікувати – виконати перевірку на правильність введених даних і відправити рецепт на модерацію, якщо він не помічений, як приватний рецепт.

У розділі «Особистий кабінет», що представляє собою модальне вікно, повинна бути інформація про авторизованого користувача та можливість її редагування. Також необхідно винести зміну пароля та електронної пошти у окреме модальне вікно. На формі особистого кабінету повинна бути кнопка для виходу з облікового запису та кнопка видалення облікового запису. При видаленні має надсилатися на електронну пошту код для підтвердження дії користувача.

При кліку на кнопку «Збережені рецепти» необхідно відфільтрувати центральну панель із публікаціями та показати користувачу лише ті, які він зберіг. Крім того, має відкриватися ліва панель та закриватися панель фільтрів. На лівій панелі мають міститися книги рецептів, що їх створив користувач та можливість створення нової книги рецептів і видалення існуючої.

3.2 Вимоги до надійності

Система має виконувати наступні вимоги до надійності:

– підтримання захисту від несанкціонованого доступу;

– розмежування прав користувачів у системі;

– обробка помилок;

– паролі повинні бути захешовані;

– валідація користувацьких даних як на стороні клієнта, так і на стороні сервера. Блокування некоректних дій користувача при роботі з системою.

3.3 Умови експлуатації

Умови експлуатації мають відповідати санітарним і технічним нормам експлуатації персонального комп’ютера, при температурі та відносній вологості навколишнього середовища, визначених для персональної обчислювальної техніки згідно з ГОСТ 15150-69.

Для обслуговування системи допускаються тільки спеціально навчені адміністратори або розробники. До користування системою допускаються звичайні Інтернет-користувачі.

3.4 Вимоги до складу та параметрів технічних засобів

Мінімальні вимоги для функціонування системи повинні відповідати вимогам будь-якого браузера. Нижче наведено приклад мінімальних вимог для браузера Google Chrome:

– платформа [OS]: Windows 10, 8, 8.1, 7;
– розрядність: x86 (32-bit) або x64 (64-bit);
– процесор Pentium 4 з SSE2;
– відеоадаптер [GPU]: 3D адаптер nVidia, Intel, AMD / ATI;
– вінчестер [HDD]: 350 Mb;
– оперативна пам'ять [RAM]: 512 Mb;
– аудіокарта [AUDIO]: Будь-яка;
– контролер: Клавіатура, Миша;
– інтернет: Стабільне з'єднання;
– роздільна здатність екрану: SVGA 800x600.

3.5 Вимоги до інформаційної та програмної сумісності

Для створення серверної частини будуть використовуватися такі технології:

– .NET C# - об’єктно-орієнтована мова програмування високого рівня;
– ASP.NET Core – фреймворк для створення продуктивних веб-застосунків;
– Redis – сховище ключ-значення, що розміщується в пам'яті, використовується для кешування даних;

– PostgreSQL – об'єктно-реляційна система керування базами даних;

– Entity Framework Core – ORM, технологія доступу та керування реляційними даним, та перетворення їх у об’єктно-орієнтовану структуру;

– Amazon S3 – це об'єктне сховище, розраховане на зберігання і витяг будь-яких обсягів даних з будь-якого розташування в Інтернеті.

Для створення клієнтської частини використовуватимуться такі технології:

– Фреймворк Angular 10 – для розробки односторінкових веб-додатків;

– HTML та CSS/Sass – для верстки та стилізації додатку.

3.6 Спеціальні вимоги

Програма повинна мати зручний, мінімалістичний та сучасний дизайн інтерфейсу, зрозумілий для будь-якого користувача.

4 Вимоги до програмної документації

У момент здачі проекту замовнику надається наступний набір документів:

– текст програми з відповідними коментарями та поясненнями;

– опис програми – відомості про функціонування програми;

– короткий посібник з переносу системи на інший хостинг;

– технічне завдання;

– короткий посібник (довідкова інформація) користувачу;

– керівництво програмісту.

5 Стадії та етапи розробки

Стадії та етапи розробки програмної система для автоматизації публікації, поширення та пошуку кулінарних рецептів подані у таблиці Б.1.

Таблиця Б.1 – Стадії та етапи розробки проекту

Стадія розробки	Етапи робіт	Зміст робіт
1	2	3
Технічне завдання 02.01.21 – 31.01.21	Обґрунтування необхідності розробки програми	Коротка характеристика програмного забезпечення; підстава і призначення розробки; вимоги до програмної системи і документація; стадії і етапи розробки програми
Ескізний проект 01.02.21 – 14.02.21	Розробка ескізного проекту	Попередня розробка структури вхідних і вихідних даних; уточнення середовища програмування; розробка і опис загальної алгоритмічної структури системи, що буде розроблюватися
Технічний проект 15.02.21 – 28.02.21	Розробка технічного проекту	Уточнення структури вхідних і вихідних даних; розробка докладного алгоритму; розробка структури програми; остаточне визначення конфігурації технічних засобів
Робочий проект 01.03.21 – 10.04.21	Розробка програмного забезпечення	Реалізація програмного забезпечення; відлагодження; проведення попереднього тестування програмної системи

Кінець таблиці Б.1

1	2	3
Розробка програмної документації 11.04.21 – 20.04.21	Розробка документації до програмного забезпечення	Розробка необхідної документації, передбаченої технічним завданням
Тестування системи 21.04.21 – 30.04.21	Проведення тестування програмного забезпечення	Розробка методики тестування; проведення основних тестів; коректування ПЗ
Впровадження	Підготовка і передача програми	Підготовка і розгортання програмного забезпечення

6 Порядок контролю та приймання

Контроль здійснюється кінцевими користувачами системи, підключеними на етапі тестування системи. Прийом комплексу здійснюється після його повного вивантаження на хостинг і налаштування для нормального функціонування.

ДОДАТОК В

(обов’язковий)

ДІАГРАМА «СУТНІСТЬ-ЗВ’ЯЗОК»

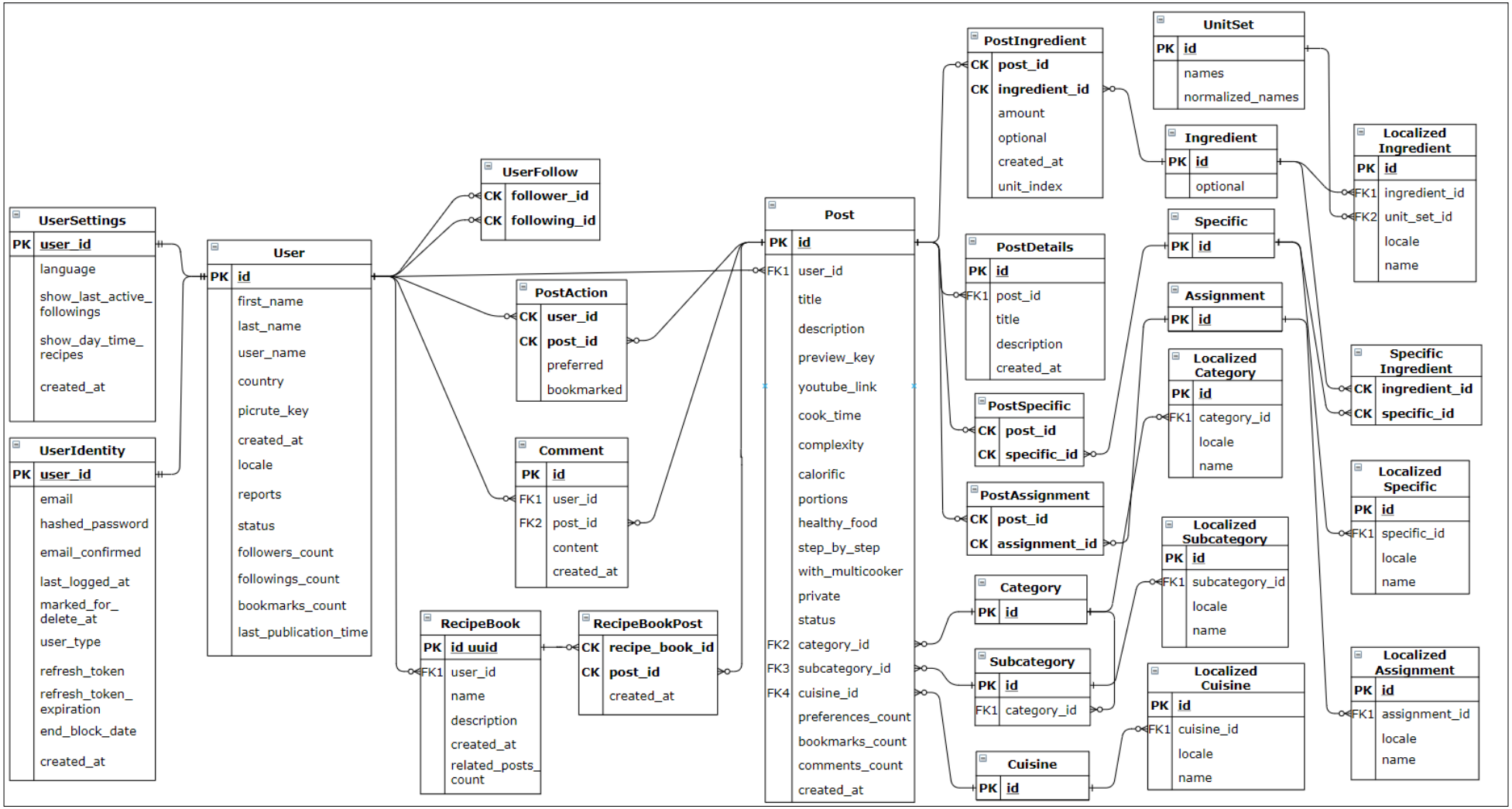


Рисунок В.1 – Діаграма «Сутність-зв’язок»

ДОДАТОК Г

(обов’язковий)

ДІАГРАМА ПАКЕТІВ

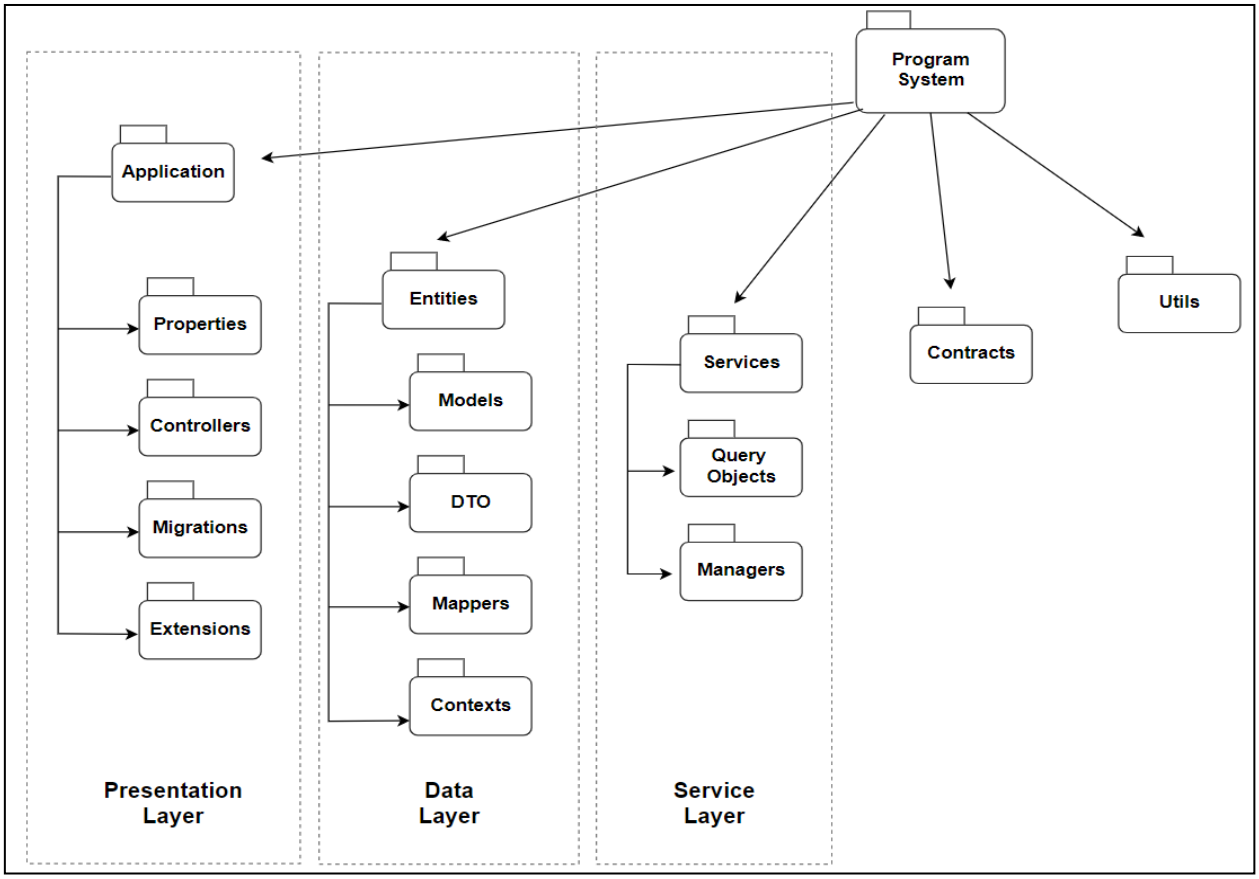


Рисунок Г.1 – Діаграма пакетів

ДОДАТОК Д

(обов’язковий)

ІЄРАРХІЯ КЛАСІВ ОБ’ЄКТНОЇ МОДЕЛІ

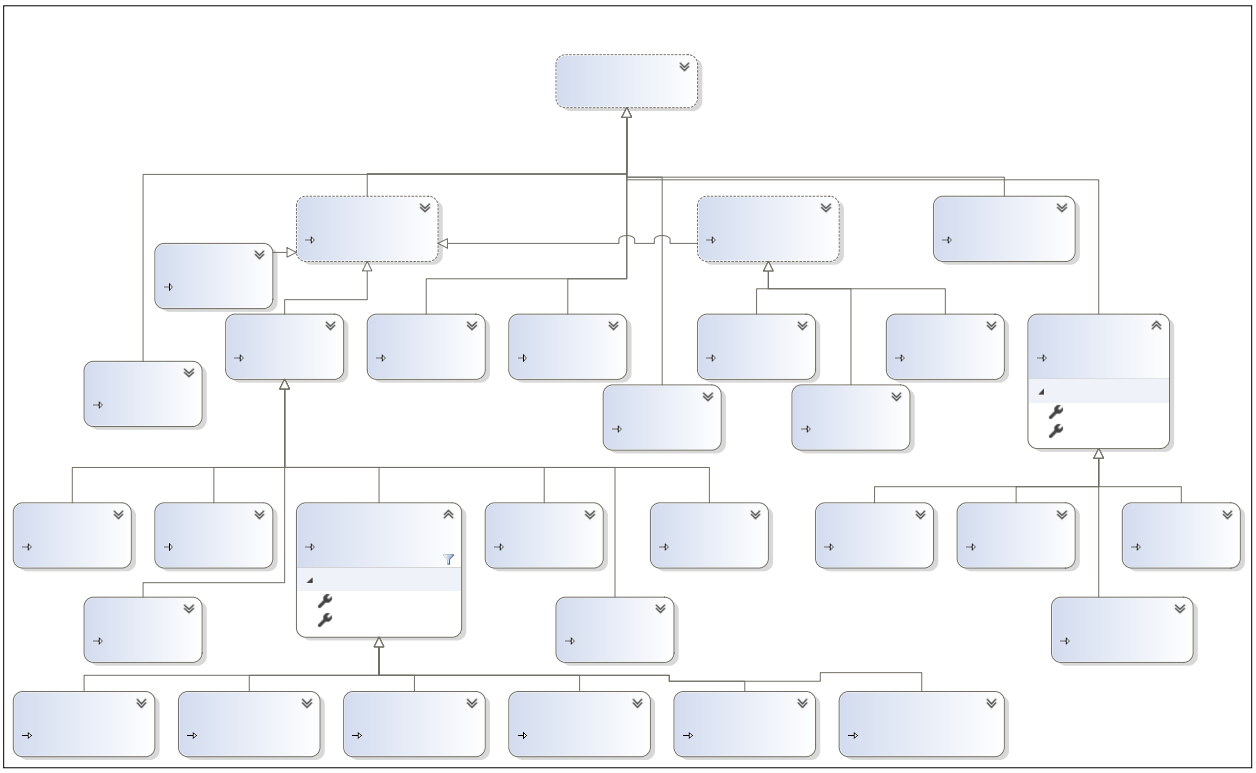


Рисунок Д.1 – Ієрархія класів об’єктної моделі

ДОДАТОК Е

(обов’язковий)

ДІАГРАМИ ПОСЛІДОВНОСТІ

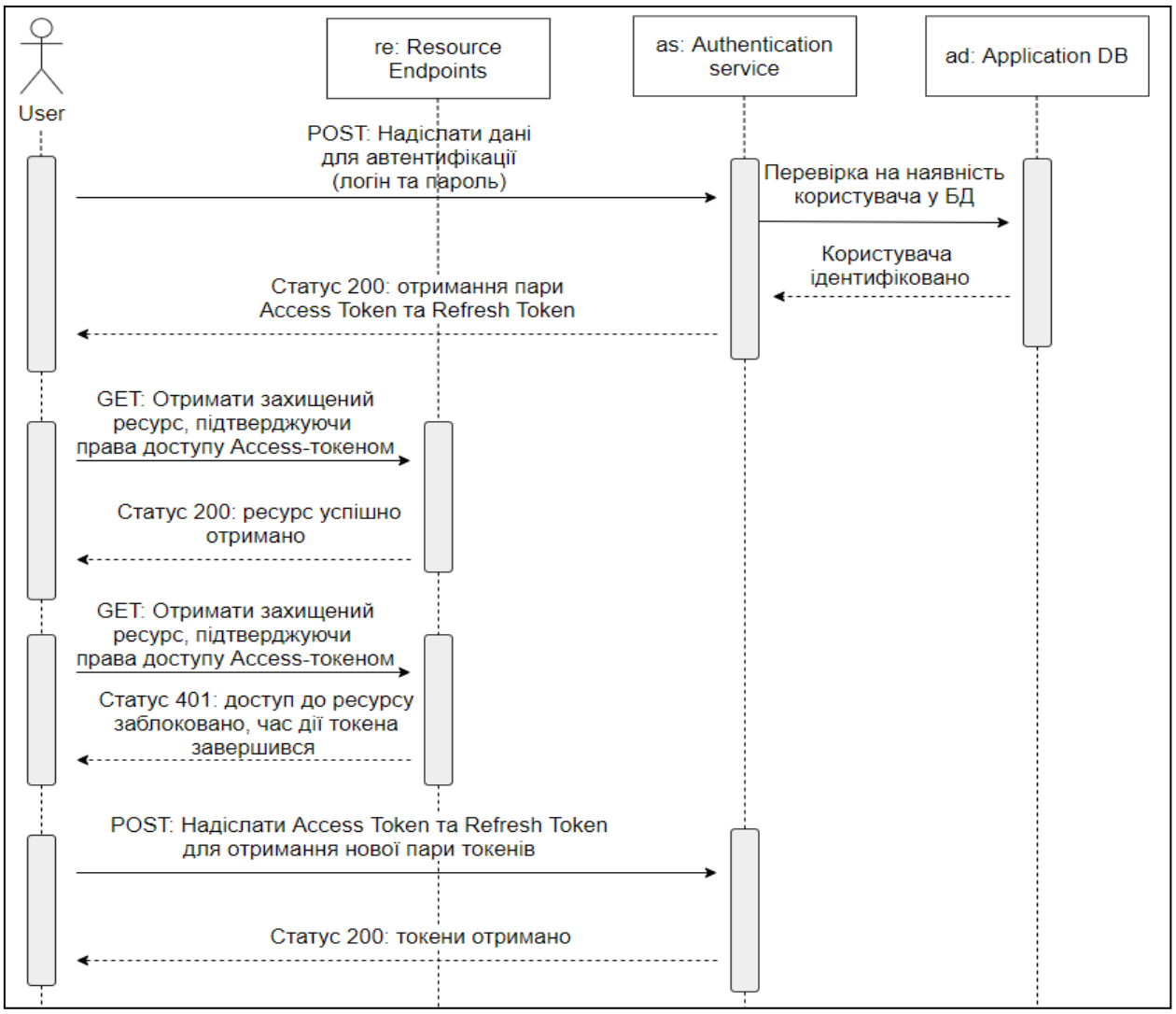


Рисунок Е.1 – Діаграма послідовності автентифікації користувача

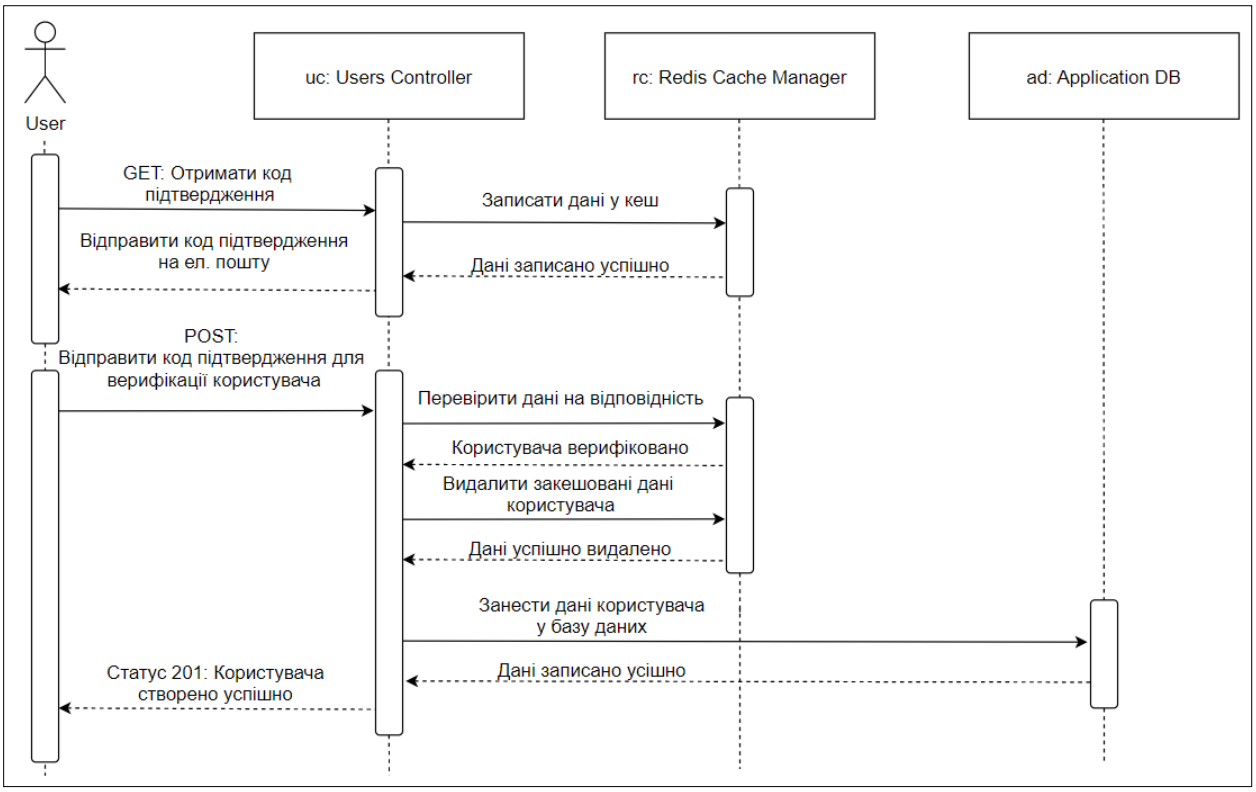


Рисунок Е.2 – Діаграма послідовності створення користувача

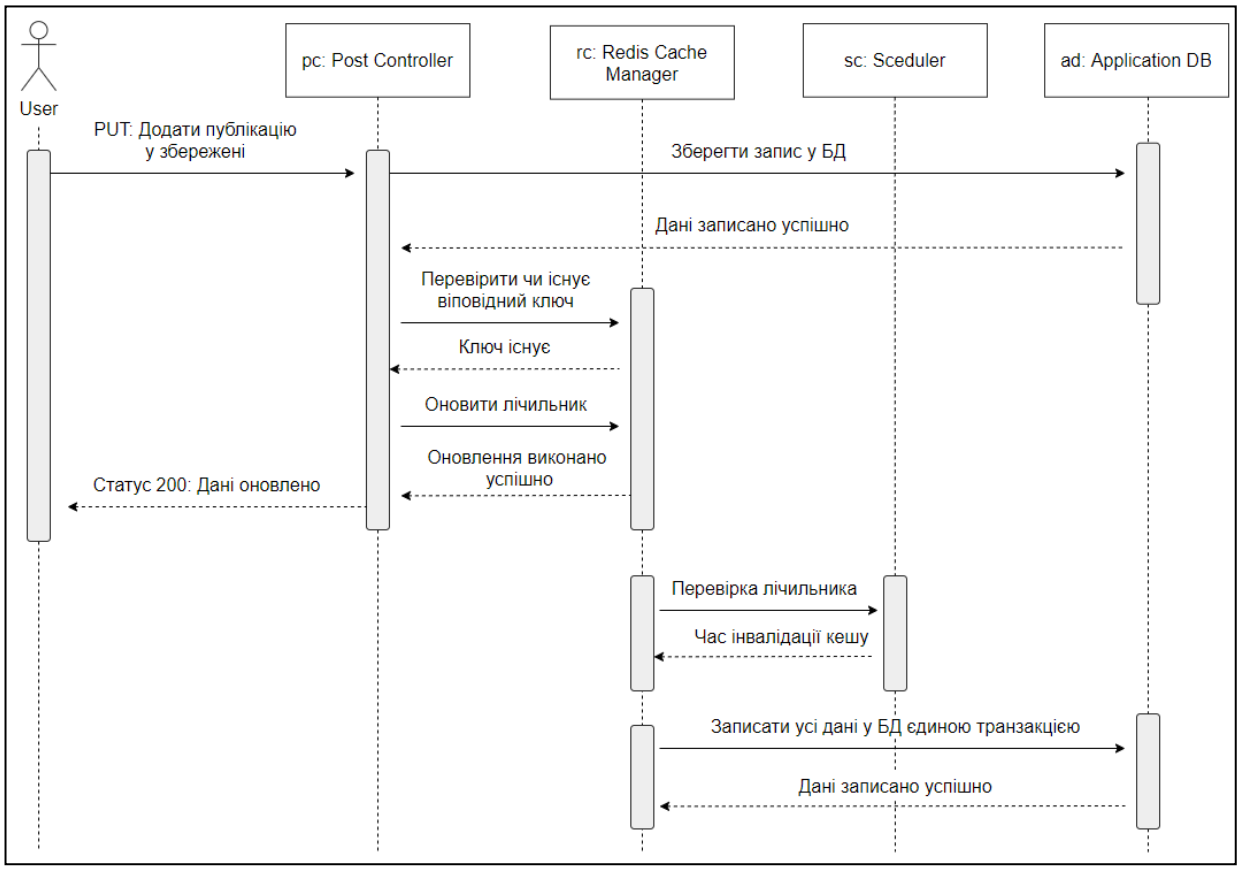


Рисунок Е.3 – Діаграма послідовності кешування оновлення лічильників

ДОДАТОК Ж

(обов’язковий)

ДІАГРАМИ ДІЯЛЬНОСТІ

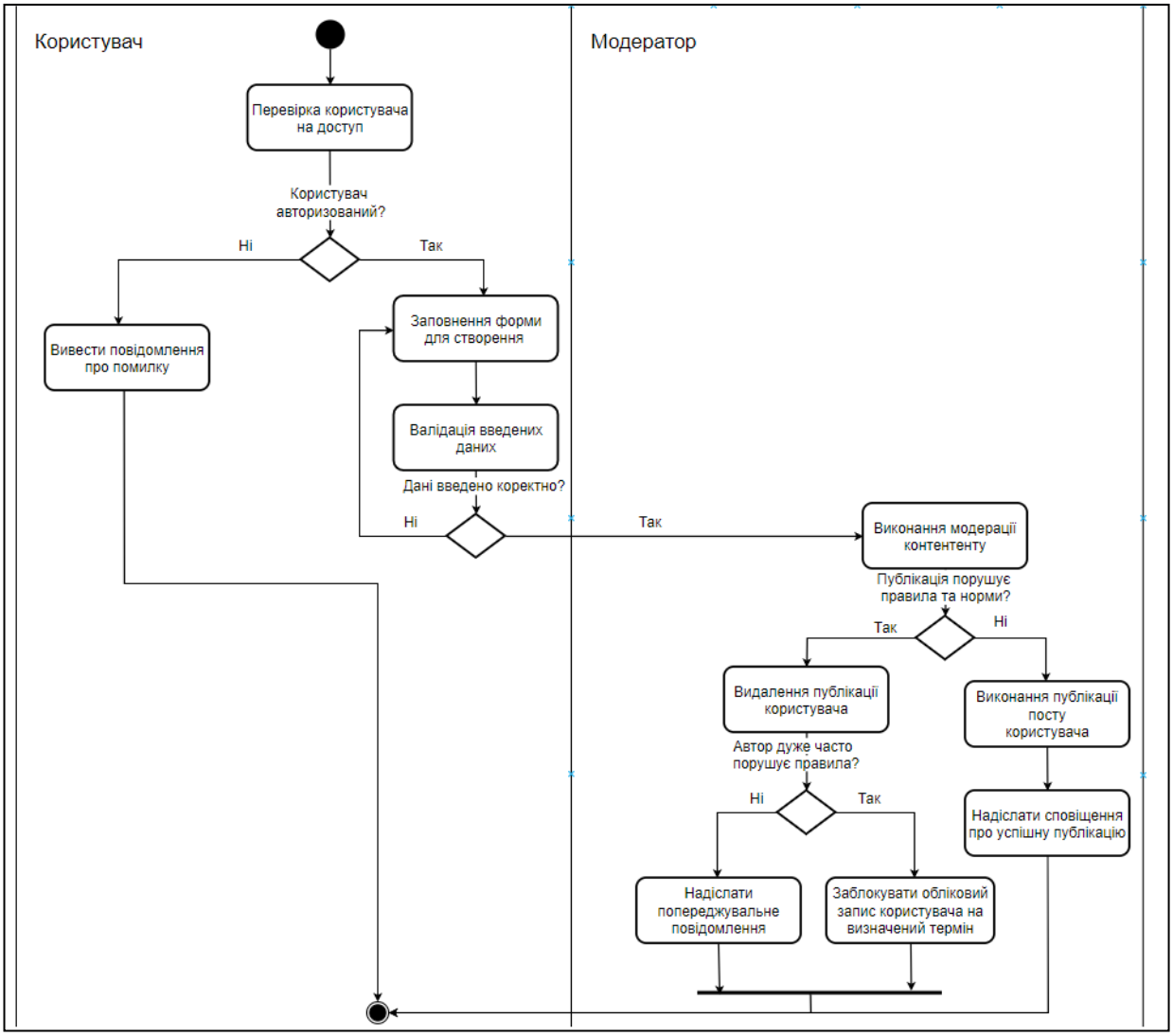


Рисунок Ж.1 – Діаграма діяльності створення публікації рецепту

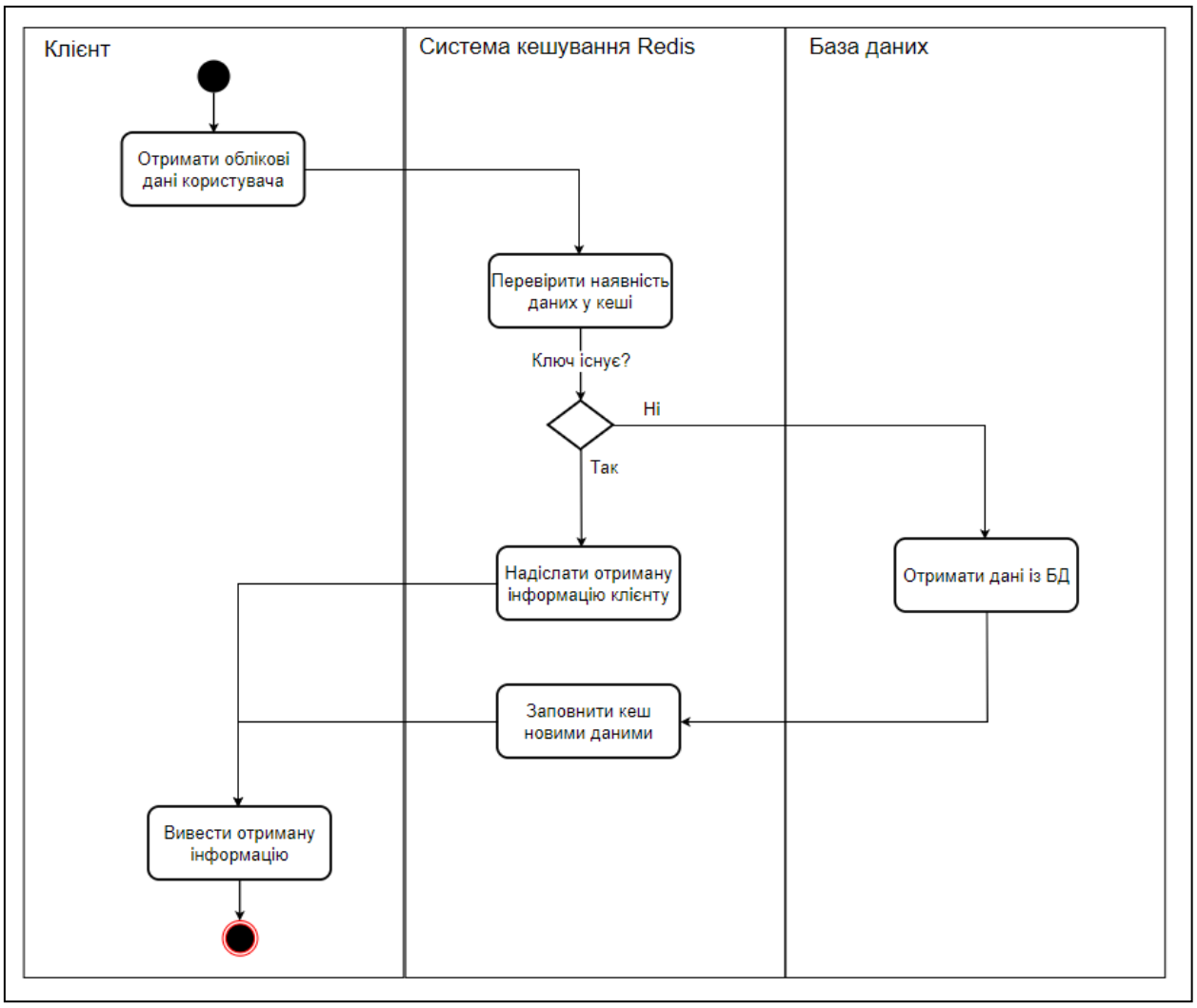


Рисунок Ж.2 – Діаграма діяльності стратегії кешування «Cache-Aside»

ДОДАТОК И

(обов’язковий)

ПРОЕКТ ІНТЕРФЕЙСУ КОРИСТУВАЧА

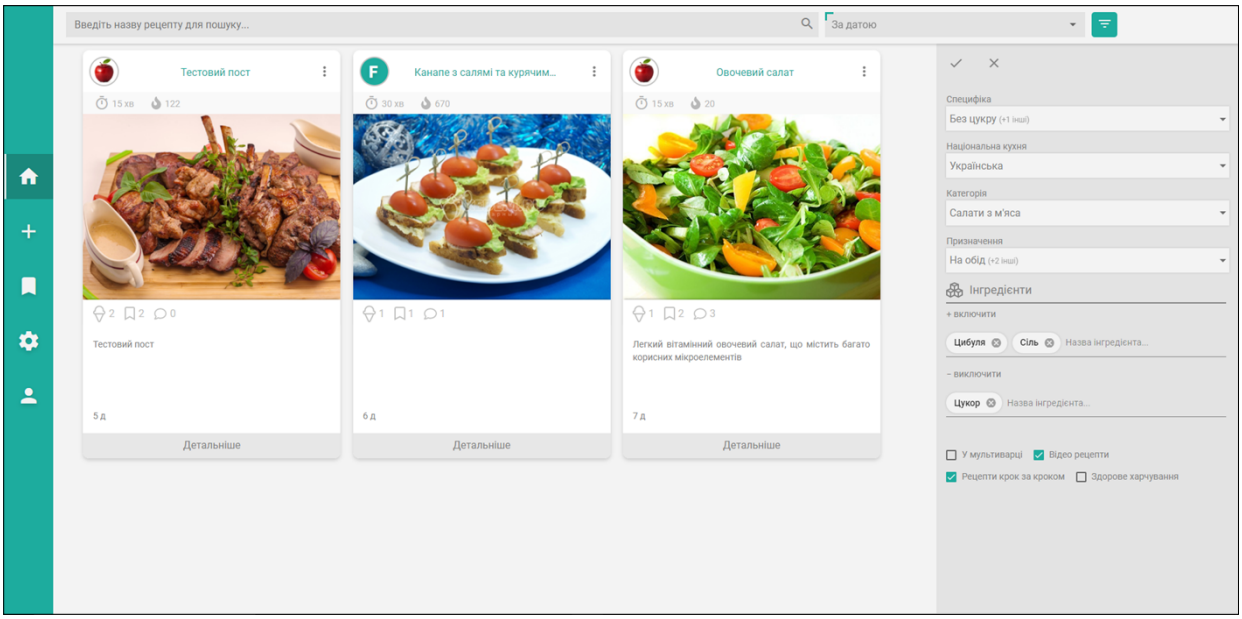


Рисунок И.1 – Проект інтерфейсу головної сторінки додатку

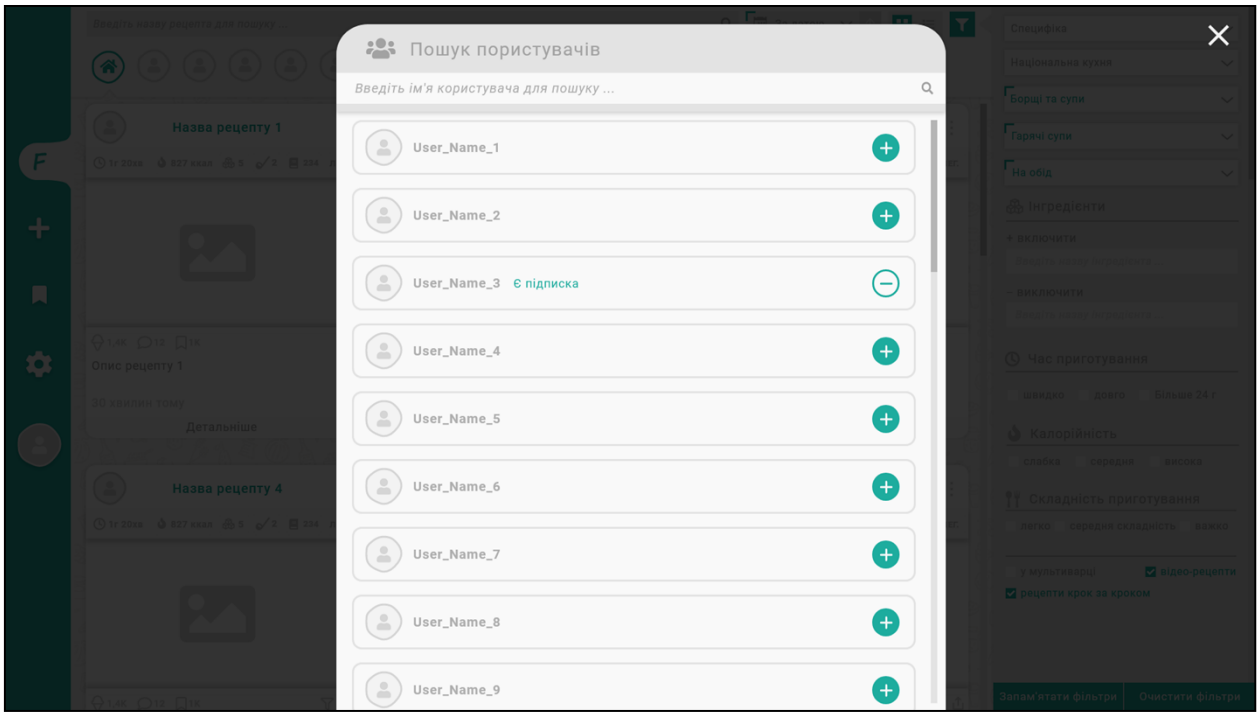


Рисунок И.2 – Модальне вікно пошуку користувачів

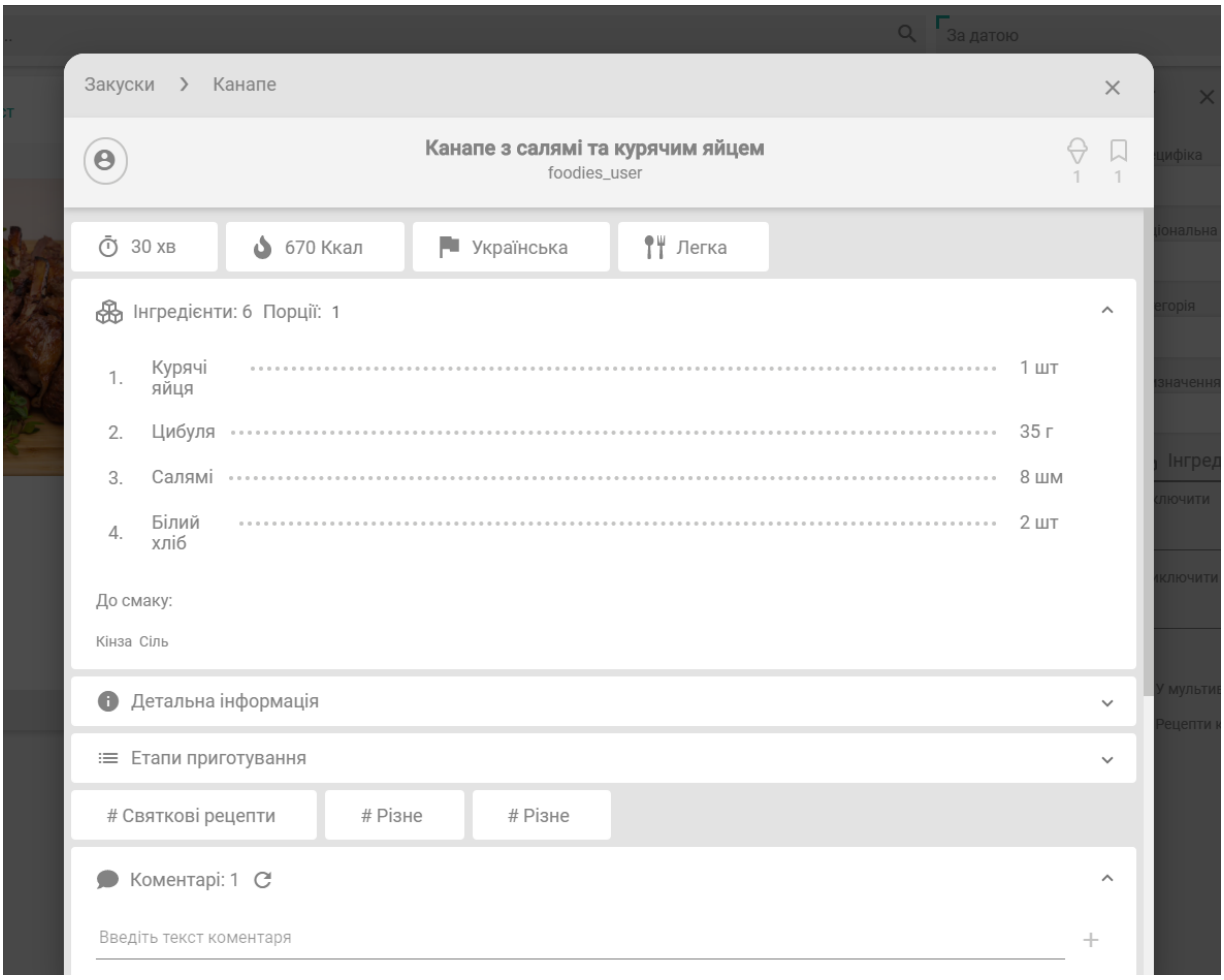


Рисунок И.3 – Модальне вікно «Детальна інформація»

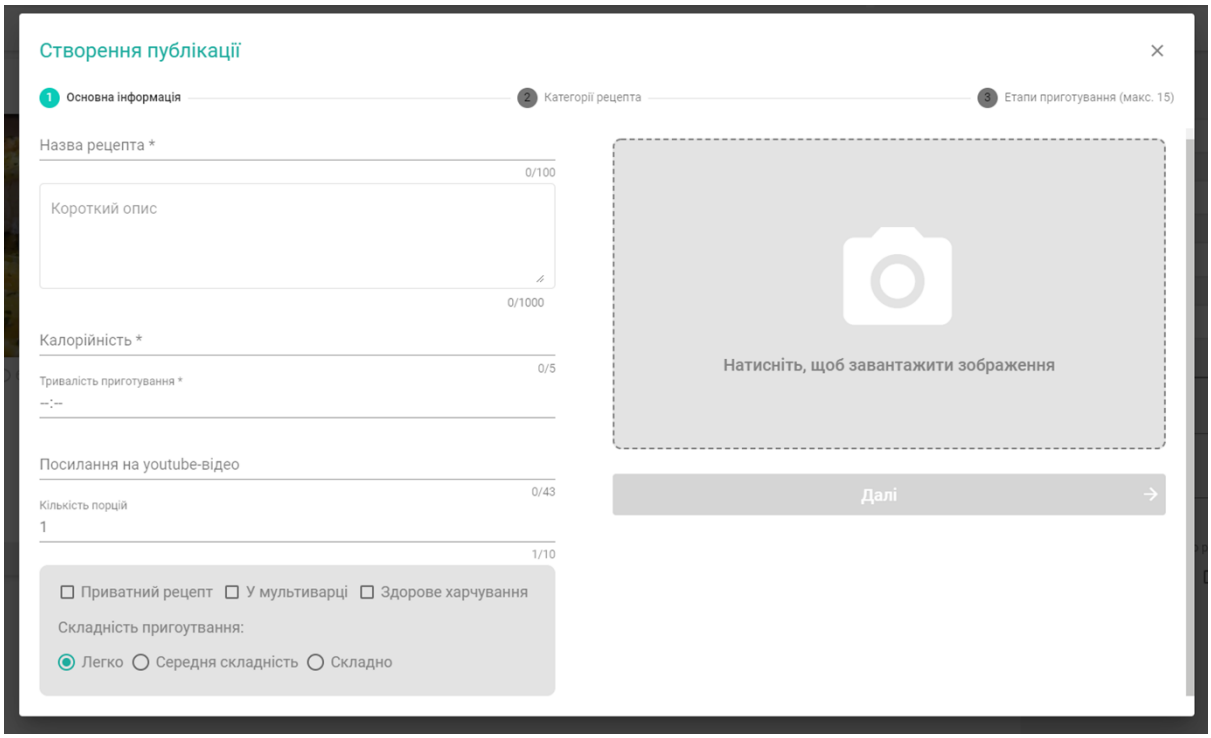
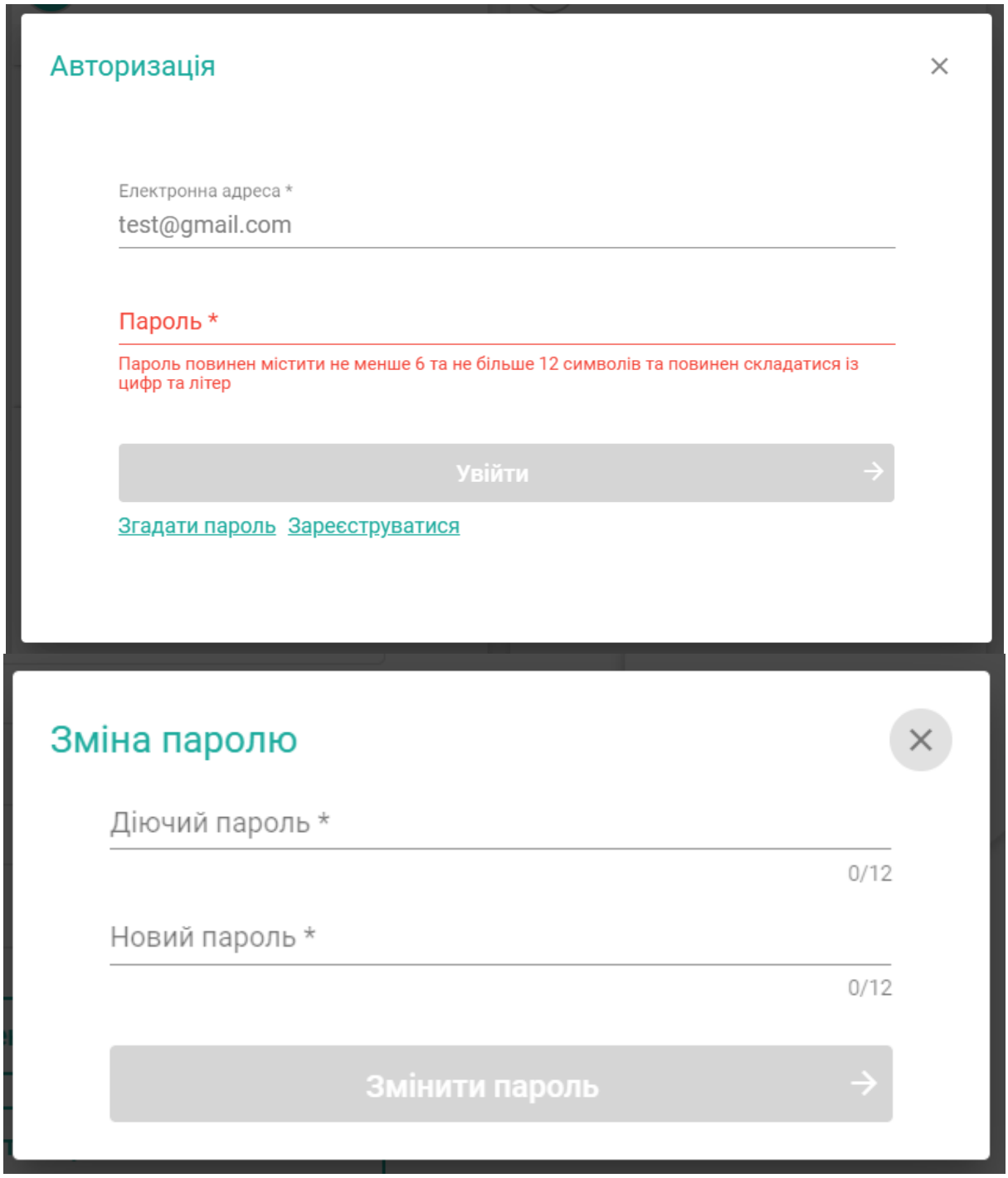


Рисунок И.4 – Форма створення публікації


Авторизація
Електронна адреса *
test@gmail.com
Пароль *
Пароль повинен містити не менше 6 та не більше 12 символів та повинен складатися із цифр та літер
Увійти
Згадати пароль
Зареєструватися
Зміна паролю
Діючий пароль *
0/12
Новий пароль *
0/12
Змінити пароль

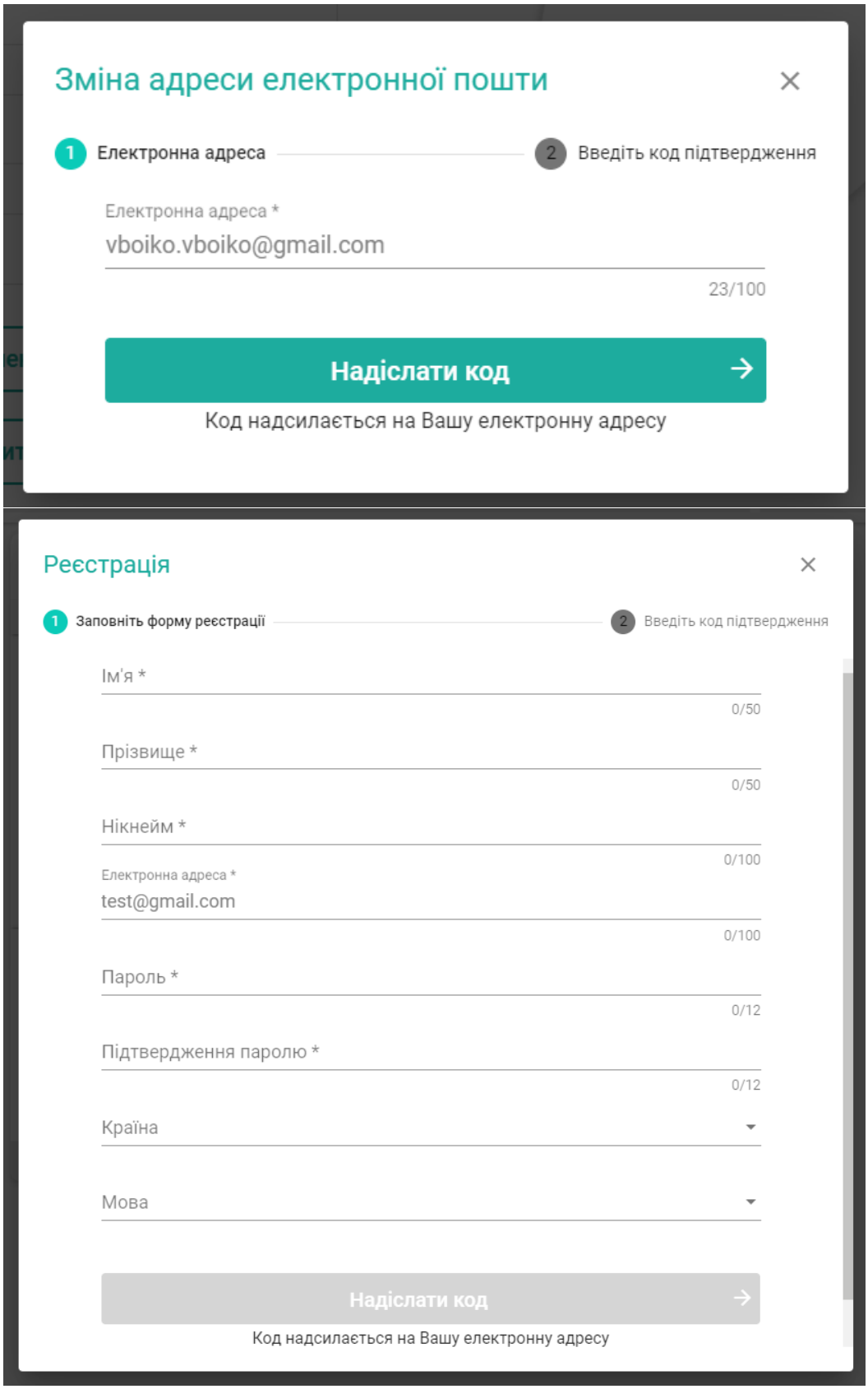


Рисунок И.5 – Форми авторизації, зміни паролю, зміни адреси електронної пошти та реєстрації

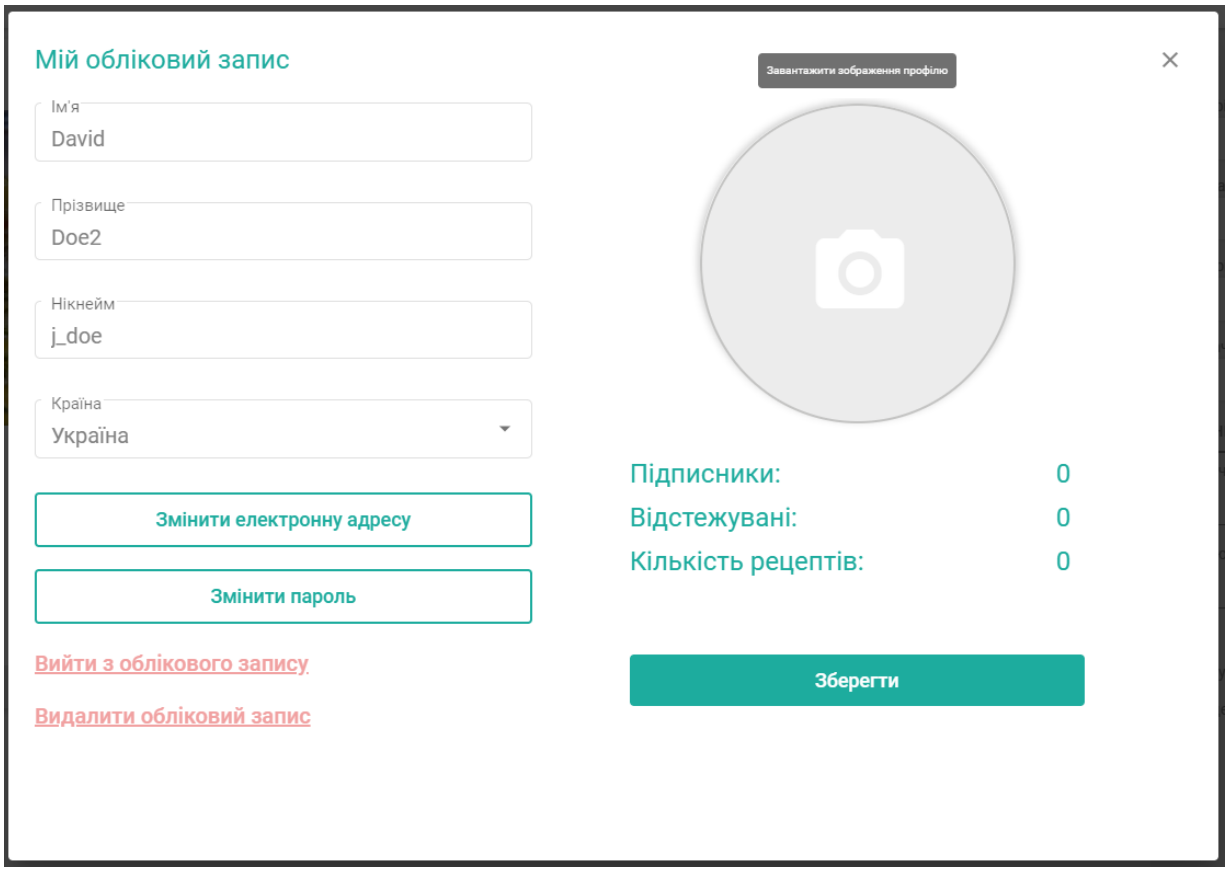


Рисунок И.6 – Розділ облікового запису користувача

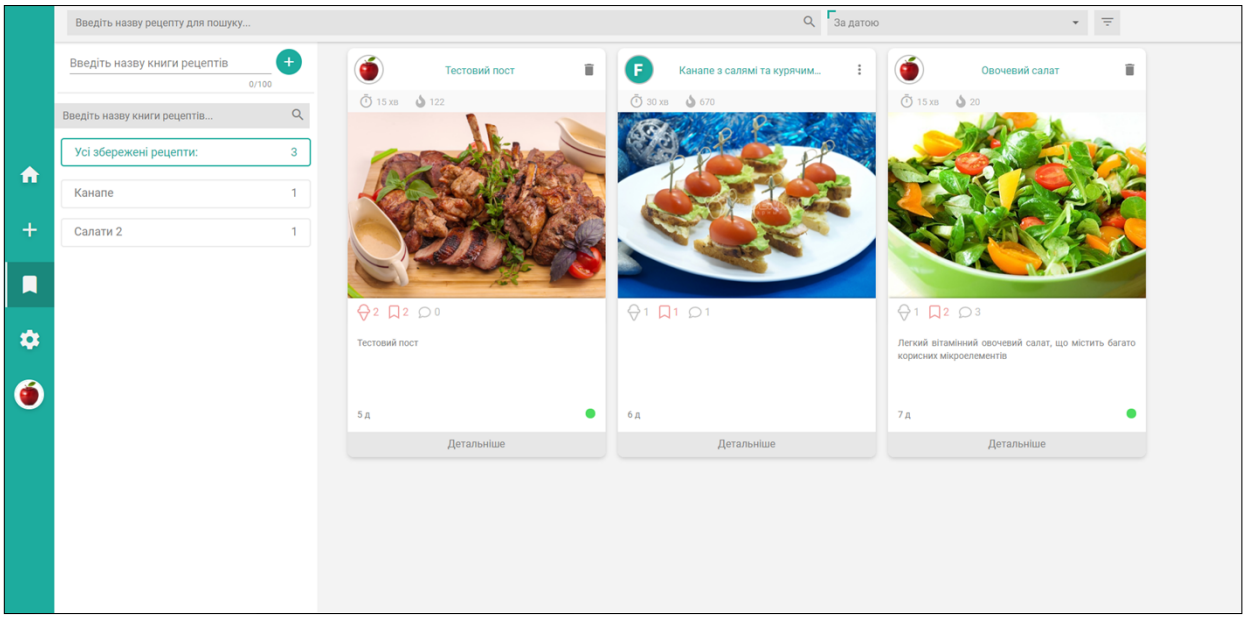


Рисунок И.8 – Розділ «Збережені рецепти»

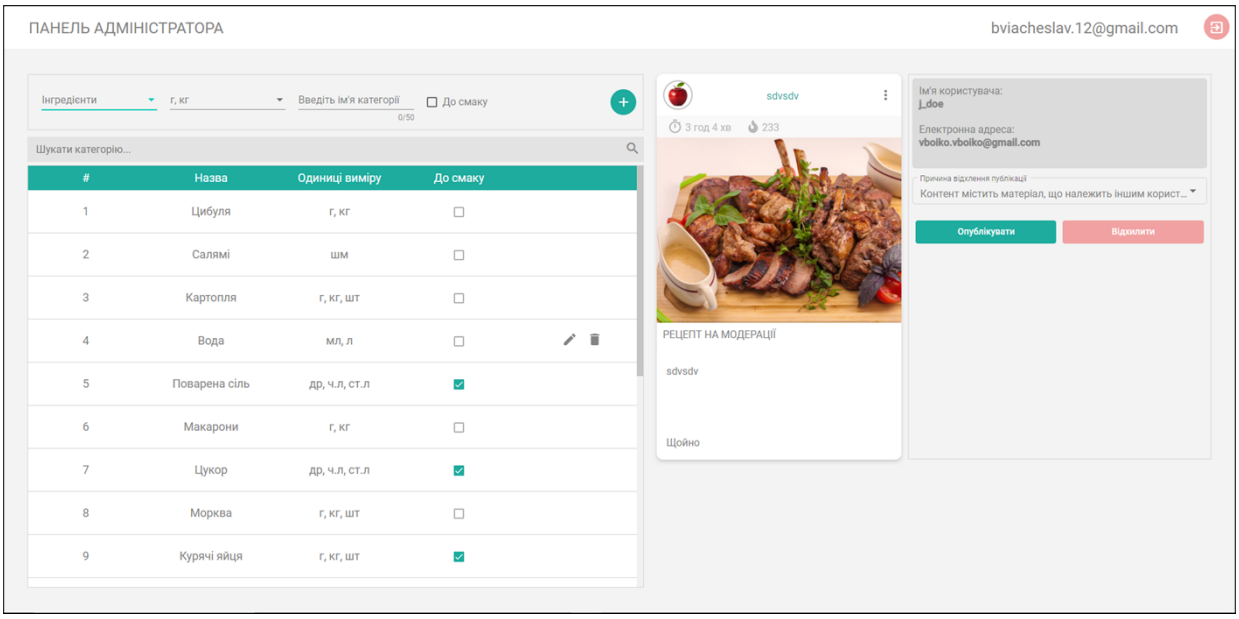


Рисунок И.9 – Розділ «Адміністративна панель»

ДОДАТОК К

(обов’язковий)

ФРАГМЕНТИ КОДУ ПРОГРАМНОЇ СИСТЕМИ

К.1 Код функцій та процедур бази даних

Функція getPosts

```
CREATE OR REPLACE FUNCTION public.getposts(
	firstpagination boolean,
	paginlimit integer,
	currentuserid uuid DEFAULT NULL::uuid,
	targetuserid uuid DEFAULT NULL::uuid,
	recipebookid uuid DEFAULT NULL::uuid,
	bookmarkedpostids uuid[] DEFAULT NULL::uuid[],
	bookmarkedpostidsforremove uuid[] DEFAULT NULL::uuid[],
	showcurrentpreferred boolean DEFAULT false,
	showcurrentbookmarked boolean DEFAULT false,
	includedingredients uuid[] DEFAULT NULL::uuid[],
	excludedingredients uuid[] DEFAULT NULL::uuid[],
	specifics uuid[] DEFAULT NULL::uuid[],
	assignments uuid[] DEFAULT NULL::uuid[],
	category uuid DEFAULT NULL::uuid,
	subcategory uuid DEFAULT NULL::uuid,
	cuisine uuid DEFAULT NULL::uuid,
	calorificfrom double precision DEFAULT NULL::double precision,
	calorificto double precision DEFAULT NULL::double precision,
	complexities integer[] DEFAULT NULL::integer[],
	stepbystep boolean DEFAULT NULL::boolean,
	withvideo boolean DEFAULT NULL::boolean,
	withmulticooker boolean DEFAULT NULL::boolean,
	healthfood boolean DEFAULT NULL::boolean,
	cooktimefrom integer DEFAULT NULL::integer,
	cooktimeto integer DEFAULT NULL::integer,
	targetlocale character varying DEFAULT NULL::character varying,
	searchterm character varying DEFAULT NULL::character varying,
	idmark uuid DEFAULT NULL::uuid,
	createdatmark timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
	initdate timestamp with time zone DEFAULT CURRENT_TIMESTAMP)
    RETURNS TABLE("Id" uuid, "Title" character varying, "Description"
character varying, "PreviewKey" character varying, "CookTime" integer,
"Complexity" integer, "Calorific" double precision, "StepByStep" boolean,
"PreferencesCount" integer, "BookmarksCount" integer, "CommentsCount"
integer, "Portions" integer, "CreatedAt" timestamp with time zone, "Private"
boolean, "YoutubeLink" character varying, "UserId" uuid, "UserPictureKey"
character varying, "UserName" character varying, "PostActions" boolean[],
"Status" integer, "RecipeBooks" uuid[])
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
begin
            return query
            -- Selection --
            select distinct
            "Post"."Id",
            "Post"."Title",
            "Post"."Description",
            "Post"."PreviewKey",
            "Post"."CookTime",
            "Post"."Complexity",
            "Post"."Calorific",
```

```
            "Post"."StepByStep",
            "Post"."PreferencesCount",
            "Post"."BookmarksCount",
            "Post"."CommentsCount",
            "Post"."Portions",
            "Post"."CreatedAt",
            "Post"."Private",
            "Post"."YoutubeLink",
            "User"."UserId",
            "User"."PictureKey" as "UserPictureKey",
            "User"."UserName",
            (select array["PostAction"."Preferred",
"PostAction"."Bookmarked"]
             from "PostAction" where currentUserId is not NULL and
"PostAction"."UserId" = currentUserId and "PostAction"."PostId" =
"Post"."Id") as "PostActions",
            "Post"."Status",
            null::uuid[] as "RecipeBooks"

            FROM "Post"

            -- Joins --
            join "User" on  "Post"."UserId" = "User"."UserId"
            join "PostSpecific" on "PostSpecific"."PostId" = "Post"."Id"
            join "PostAssignment" on "PostAssignment"."PostId" = "Post"."Id"

            -- Conditions --

            where ("Post"."Status" = 0 or ("Post"."Status" = 1 and
"Post"."UserId" = currentuserid))
            and (targetLocale is NULL or (targetLocale is not NULL and
"User"."Locale" = targetLocale))
            and ("Post"."Private" = false or ("Post"."Private" = true and
currentUserId is not NULL and "User"."UserId" = currentUserId))
            and (targetUserId is NULL or (targetUserId is not NULL and
("Post"."Private" = false and "User"."UserId" = targetUserId)))
            and  (includedIngredients is NULL or (includedIngredients is not
NULL and exists (select "IngredientId" from unnest(includedIngredients)
PIngredientId inner join "PostIngredient" on "PostIngredient"."IngredientId"
= PIngredientId where "PostIngredient"."PostId" = "Post"."Id")))
            and  (excludedIngredients is NULL or (excludedIngredients is not
NULL and not exists (select "IngredientId" from unnest(excludedIngredients)
PIngredientId inner join "PostIngredient" on  "PostIngredient"."IngredientId"
= PIngredientId where "PostIngredient"."PostId" = "Post"."Id")))
            and  (specifics is NULL or (specifics is not NULL and
"PostSpecific"."SpecificId" = any(specifics)))
            and  (assignments is NULL or (assignments is not NULL and
"PostAssignment"."AssignmentId" = any(assignments)))
            and  (category is NULL or (category is not NULL and
"Post"."CategoryId" = category))
            and  (subcategory is NULL or (subcategory is not NULL and
"Post"."SubcategoryId" = subcategory))
            and  (cuisine is NULL or (cuisine is not NULL and
"Post"."CuisineId" = cuisine))
            and  ((calorificFrom is NULL or calorificTo is NULL) or
((calorificFrom is not NULL and calorificTo is not NULL) and
"Post"."Calorific" >= calorificFrom and "Post"."Calorific" <= calorificTo))
            and  (complexities is NULL or (complexities is not NULL and
"Post"."Complexity" = any(complexities)))
            and  ((stepByStep is NULL or stepByStep = false) or
("Post"."StepByStep" = true))
            and  ((withVideo is NULL or withVideo = false) or (withVideo =
true and "Post"."YoutubeLink" is not NULL))
```

```
            and  ((withMulticooker is NULL or withMulticooker = false) or
("Post"."WithMulticooker" = true))
            and  ((healthFood is NULL or healthFood = false) or
("Post"."HealthyFood" = true))
            and  ((cooktimeFrom is NULL or cooktimeTo is NULL) or
((cooktimeFrom is not NULL and cooktimeTo is not NULL) and "Post"."CookTime"
>= cooktimeFrom and "Post"."CookTime" <= cooktimeTo))
            and (searchterm is NULL or (searchterm is not NULL and
lower("Post"."Title") like lower(searchterm)))

            -- Pagination block --
            and (
                    (not firstPagination
                            and ("Post"."CreatedAt", "Post"."Id") <
(createdatmark, idMark)
                            and "Post"."CreatedAt" < initDate)
                      or (firstPagination = true))

            -- Orderings --

            order by "Post"."CreatedAt" desc, "Post"."Id" desc

            limit paginLimit;

            end if;

            end; $BODY$;
```

Функція getRecipeBooks

```
CREATE OR REPLACE FUNCTION public.getrecipebooks(
      firstpagination boolean,
      userid uuid,
      paginlimit integer DEFAULT 15,
      searchterm character varying DEFAULT NULL::character varying,
      idmark uuid DEFAULT NULL::uuid,
      createdatmark timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
      initdate timestamp with time zone DEFAULT CURRENT_TIMESTAMP)
    RETURNS  TABLE("Id"  uuid,  "UserId"  uuid,  "Name"  character  varying,
"Description"  character  varying,  "RelatedPostsCount"  integer,  "CreatedAt"
timestamp with time zone)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
begin
            return query
            select          "RecipeBook"."Id",          "RecipeBook"."UserId",
"RecipeBook"."Name",                                "RecipeBook"."Description",
"RecipeBook"."RelatedPostsCount", "RecipeBook"."CreatedAt"
            FROM "RecipeBook"
            where "RecipeBook"."UserId" = userId and ((
                  not firstPagination
                  and    ("RecipeBook"."CreatedAt",    "RecipeBook"."Id")    <
(createdAtMark, idMark)
                  and "RecipeBook"."CreatedAt" < initDate
                  and ((searchTerm is not NULL and lower("RecipeBook"."Name")
like lower(searchTerm)) or (searchTerm is NULL))
```

```
            )
            or (firstPagination = true and ((searchTerm is not NULL and
lower("RecipeBook"."Name") like lower(searchTerm)) or (searchTerm is NULL))))
            order by "RecipeBook"."CreatedAt" desc, "RecipeBook"."Id" desc
            limit paginLimit;
end
$BODY$;
```

Функція getPostsForModerator

```
CREATE OR REPLACE FUNCTION public.getpostsformoderator(
      firstpagination boolean,
      paginlimit integer,
      showpostsonmoderation boolean,
      targetuserid uuid DEFAULT NULL::uuid,
      targetlocale character varying DEFAULT NULL::character varying,
      searchterm character varying DEFAULT NULL::character varying,
      idmark uuid DEFAULT NULL::uuid,
      createdatmark timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
      initdate timestamp with time zone DEFAULT CURRENT_TIMESTAMP)
    RETURNS TABLE("Id" uuid, "Title" character varying, "Description" character
varying, "PreviewKey" character varying, "CookTime" integer, "Complexity"
integer, "Calorific" double precision, "StepByStep" boolean, "Portions"
integer, "CreatedAt" timestamp with time zone, "Private" boolean, "YoutubeLink"
character varying, "Status" integer, "UserId" uuid, "UserPictureKey" character
varying, "UserName" character varying, "Email" character varying)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
begin
            return query

            select distinct
            "Post"."Id",
            "Post"."Title",
            "Post"."Description",
            "Post"."PreviewKey",
            "Post"."CookTime",
            "Post"."Complexity",
            "Post"."Calorific",
            "Post"."StepByStep",
            "Post"."Portions",
            "Post"."CreatedAt",
            "Post"."Private",
            "Post"."YoutubeLink",
            "Post"."Status",
            "User"."UserId",
            "User"."PictureKey" as "UserPictureKey",
            "User"."UserName",
            "UserIdentity"."Email"

            FROM "Post"

            join "User" on  "Post"."UserId" = "User"."UserId"
            join "UserIdentity" on  "UserIdentity"."Id" = "User"."UserId"
```

```
            where (((showPostsOnModeration = true and "Post"."Status" = 1) or
(not showPostsOnModeration and "Post"."ReportsCount" >= 10)))
            and (targetLocale is NULL or (targetLocale is not NULL and
"User"."Locale" = targetLocale))
            and (targetUserId is NULL or (targetUserId is not NULL and
"User"."UserId" = targetUserId))
            and (searchterm is NULL or (searchterm is not NULL and
lower("Post"."Title") like lower(searchterm)))
            and (
                    (not firstPagination
                          and     ("Post"."CreatedAt",     "Post"."Id")     >
(createdatmark, idMark)
                          and "Post"."CreatedAt" < initDate)
                     or (firstPagination = true))
            order by "Post"."CreatedAt" asc, "Post"."Id" asc

            limit paginLimit;
end
$BODY$;
```

Функція getComments

```
CREATE OR REPLACE FUNCTION public.getcomments(
      firstpagination boolean,
      postid uuid,
      paginlimit integer DEFAULT 20,
      userid uuid DEFAULT NULL::uuid,
      idmark uuid DEFAULT '00000000-0001-0000-0000-000000000000'::uuid,
      createdatmark timestamp with time zone DEFAULT CURRENT_TIMESTAMP,
      initdate timestamp with time zone DEFAULT CURRENT_TIMESTAMP)
    RETURNS TABLE("Id" uuid, "UserId" uuid, "UserName" character varying,
"UserPictureKey" character varying, "PostId" uuid, "CreatedAt" timestamp with
time zone, "Content" character varying, "Modified" boolean)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
begin
      return query
      select

      "Comment"."Id",
      "Comment"."UserId",
      "User"."UserName",
      "User"."PictureKey" as "UserPictureKey",
      "Comment"."PostId",
      "Comment"."CreatedAt",
      "Comment"."Content",
      "Comment"."Modified"

      FROM "Comment"
      join "User" on "User"."UserId" = "Comment"."UserId"
      where "User"."Status" = 0 and (("Comment"."PostId" = postId
      and (not firstPagination
            and  ("Comment"."CreatedAt",  "Comment"."Id")  <  (createdatmark,
idMark)
            and "Comment"."CreatedAt" < initDate)
      and ((userId is not NULL and "Comment"."UserId" = userId)
```

```
            or userId is NULL))
      or (firstPagination = true and "Comment"."PostId" = postId and ((userId
is not NULL and "Comment"."UserId" = userId) or userId is NULL)))
      order by "Comment"."CreatedAt" desc, "Comment"."Id" desc
      limit paginLimit;
end;
$BODY$;
```

Функція getLastActiveUsers

```
CREATE OR REPLACE FUNCTION public.getlastactiveusers(
      firstpagination boolean,
      searchterm character varying DEFAULT NULL::character varying,
      currentuserid uuid DEFAULT NULL::uuid,
      retrievefollowers boolean DEFAULT NULL::boolean,
      paginlimit integer DEFAULT 20,
      targetlocale character varying DEFAULT 'en-EN'::character varying,
      idmark uuid DEFAULT '00000000-0001-0000-0000-000000000000'::uuid,
      lastpublicationmark timestamp with time zone DEFAULT CURRENT_TIMESTAMP)
    RETURNS TABLE("UserId" uuid, "FirstName" character varying, "LastName"
character varying, "UserName" character varying, "PictureKey" character
varying, "FollowersCount" integer, "FollowingsCount" integer, "RecipesCount"
integer, "BookmarksCount" integer, "LastPublicationTime" timestamp with time
zone)
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
    ROWS 1000

AS $BODY$
begin
    if currentUserId is NULL then
            return query
            select

            "User"."UserId",
            "User"."FirstName",
            "User"."LastName",
            "User"."UserName",
            "User"."PictureKey",
            "User"."FollowersCount",
            "User"."FollowingsCount",
            "User"."RecipesCount",
            "User"."BookmarksCount",
            "User"."LastPublicationTime"

            FROM "User"
            where "User"."Status" = 0 and ((searchTerm is not NULL and
(lower("User"."UserName") like lower(searchTerm) or lower("User"."FirstName")
like lower(searchTerm) or lower("User"."LastName") like lower(searchTerm))) or
(searchTerm is NULL)) and ((
                  not firstPagination
                  and   ("User"."LastPublicationTime",   "User"."UserId")   <
(lastPublicationMark, idMark)
                  and "User"."Locale" = targetLocale)
            or (firstPagination = true and "User"."Locale" = targetLocale))
            order by "User"."LastPublicationTime" desc, "User"."UserId" desc
            limit paginLimit;

      end if;
```

```
	if retrieveFollowers = true then

		return query
		select

		"User"."UserId",
		"User"."FirstName",
		"User"."LastName",
		"User"."UserName",
		"User"."PictureKey",
		"User"."FollowersCount",
		"User"."FollowingsCount",
		"User"."RecipesCount",
		"User"."BookmarksCount",
		"User"."LastPublicationTime"

		FROM "UserFollow"
		join "User" on "UserFollow"."FollowerId" = "User"."UserId"
		where "User"."Status" = 0 and "UserFollow"."FollowingId" =
currentuserid and ((searchTerm is not NULL and (lower("User"."UserName") like
lower(searchTerm) or lower("User"."FirstName") like lower(searchTerm) or
lower("User"."LastName") like lower(searchTerm))) or (searchTerm is NULL)) and
((
			not firstPagination
			and ("User"."LastPublicationTime", "User"."UserId") <
(lastPublicationMark, idMark)
			and "User"."Locale" = targetLocale)
		or (firstPagination = true and "User"."Locale" = targetLocale))
		order by "User"."LastPublicationTime" desc, "User"."UserId" desc
		limit paginLimit;
	else
		return query
		select

		"User"."UserId",
		"User"."FirstName",
		"User"."LastName",
		"User"."UserName",
		"User"."PictureKey",
		"User"."FollowersCount",
		"User"."FollowingsCount",
		"User"."RecipesCount",
		"User"."BookmarksCount",
		"User"."LastPublicationTime"

		FROM "UserFollow"
		join "User" on "UserFollow"."FollowingId" = "User"."UserId"
		where "User"."Status" = 0 and "UserFollow"."FollowerId" =
currentuserid and ((searchTerm is not NULL and (lower("User"."UserName") like
lower(searchTerm) or lower("User"."FirstName") like lower(searchTerm) or
lower("User"."LastName") like lower(searchTerm))) or (searchTerm is NULL)) and
((
			not firstPagination
			and ("User"."LastPublicationTime", "User"."UserId") <
(lastPublicationMark, idMark)
			and "User"."Locale" = targetLocale)
		or (firstPagination = true and "User"."Locale" = targetLocale))
		order by "User"."LastPublicationTime" desc, "User"."UserId" desc
		limit paginLimit;
	end if;

end;
$BODY$;
```

Процедура markUsersForRemove

```
CREATE OR REPLACE PROCEDURE public.mark_users_for_remove(
      currenttime timestamp with time zone)
LANGUAGE 'sql'
AS $BODY$
WITH src AS (
        UPDATE "UserIdentity"
        SET "LastLoggedAt" = NULL
        WHERE  DATE_PART('days',  currenttime  -  "LastLoggedAt")  >=  30  and
"UserType" = 2 and "EndBlock" is NULL
        RETURNING *
        )
UPDATE "User" dst
SET "Status" = 3
FROM src
WHERE dst."UserId" = src."Id" and dst."Status" != 3
$BODY$;
```

Процедура updatePostRelatedData

```
CREATE OR REPLACE PROCEDURE public.update_post_related_data(
      post_id uuid,
      preferred_count integer,
      bookmarked_count integer,
      user_ids uuid[],
      preferrences boolean[],
      bookmarks boolean[],
      post_ids uuid[])
LANGUAGE 'sql'
AS $BODY$
update "Post"
set "PreferencesCount" = "PreferencesCount" + preferred_count,
      "BookmarksCount" = "BookmarksCount" + bookmarked_count
where "Id" = post_id;

INSERT INTO "PostAction"("UserId", "PostId", "Preferred", "Bookmarked")
   SELECT * FROM unnest(user_ids, post_ids, preferrences, bookmarks)
   ON CONFLICT ("UserId", "PostId")
   DO UPDATE SET
      "Preferred" = excluded."Preferred",
      "Bookmarked" = excluded."Bookmarked";
$BODY$;
```

К.2 Фрагменти коду серверної частини програмної системи

Клас PostgresContext

```
public class PostgresContext : DbContext
    {
        public PostgresContext(DbContextOptions<PostgresContext> options)
            : base(options)
        {
            ChangeTracker.QueryTrackingBehavior =
QueryTrackingBehavior.NoTracking;
        }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
             => optionsBuilder.LogTo(Console.WriteLine);

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<UserFollow>()
                .HasKey(u => new { u.FollowerId, u.FollowingId });

            modelBuilder.Entity<PostAction>()
                .HasKey(p => new { p.PostId, p.UserId });

            modelBuilder.Entity<PostIngredient>()
                .HasKey(p => new { p.PostId, p.IngredientId });

            modelBuilder.Entity<PostSpecific>()
                .HasKey(p => new { p.PostId, p.SpecificId });

            modelBuilder.Entity<PostAssignment>()
                .HasKey(p => new { p.PostId, p.AssignmentId });

            modelBuilder.Entity<RecipeBookPost>()
                .HasKey(r => new { r.RecipeBookId, r.PostId });

            modelBuilder.Entity<SpecificIngredient>()
                .HasKey(s => new { s.SpecificId, s.IngredientId });

            modelBuilder.Entity<UserIdForEmailCheckDto>().HasNoKey();

            modelBuilder.Entity<ReportedUserDto>().HasNoKey();

            modelBuilder.Entity<ResponseUserDto>().HasNoKey();

            modelBuilder.Entity<MiscResponseDto>().HasNoKey();

            modelBuilder.Entity<SubcategoryDto>().HasNoKey();

            modelBuilder.Entity<IngredientsResponseDto>().HasNoKey();

            modelBuilder.Entity<PostDetailsDto>().HasNoKey();

            modelBuilder.Entity<PostIngredientDto>().HasNoKey();

            modelBuilder.Entity<PostDto>().HasNoKey();

            modelBuilder.Entity<PostForModerationDto>().HasNoKey();
```

```
    modelBuilder.Entity<CommentDto>().HasNoKey();
}

/** Dto Entities **/

public DbSet<CommentDto> OutputComments { get; set; }

public DbSet<UserIdForEmailCheckDto> UsersForEmailCheck { get; set; }

public DbSet<ResponseUserDto> ResponseUsers { get; set; }

public DbSet<ReportedUserDto> ReportedUsers { get; set; }

public DbSet<MiscResponseDto> OutputMiscs { get; set; }

public DbSet<SubcategoryDto> OutputCategoriesWithSubs{ get; set; }

public DbSet<IngredientsResponseDto> OutputIngredients { get; set; }

public DbSet<PostIngredientDto> RelatedIngredients { get; set; }

public DbSet<PostDetailsDto> PostSteps { get; set; }

public DbSet<PostDto> OutputPosts { get; set; }

public DbSet<PostForModerationDto> PostsForModeration { get; set; }

/** Plain Entities **/

public DbSet<UserIdentity> UserIdentities { get; set; }

public DbSet<User> Users { get; set; }

public DbSet<UserSettings> UserSettings { get; set; }

public DbSet<Post> Posts { get; set; }

public DbSet<RecipeBook> RecipeBooks { get; set; }

public DbSet<UserFollow> UserFollows { get; set; }

public DbSet<PostAction> PostActions { get; set; }

public DbSet<PostDetails> PostDetails { get; set; }

public DbSet<RecipeBookPost> RecipeBookPosts { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<Ingredient> Ingredients { get; set; }

public DbSet<Specific> Specifics { get; set; }

public DbSet<SpecificIngredient> SpecificIngredients { get; set; }

public DbSet<Assignment> Assignments { get; set; }

public DbSet<Category> Categories { get; set; }

public DbSet<Subcategory> Subcategories { get; set; }

public DbSet<Cuisine> Cuisines { get; set; }
```

```
        public DbSet<PostIngredient> PostIngredients { get; set; }

        public DbSet<PostSpecific> PostSpecifics { get; set; }

        public DbSet<PostAssignment> PostAssignments { get; set; }

        public DbSet<LocalizedAssignment> LocalizedAssignments { get; set; }

        public DbSet<LocalizedCategory> LocalizedCategories { get; set; }

        public DbSet<LocalizedCuisine> LocalizedCuisines { get; set; }

        public DbSet<LocalizedIngredient> LocalizedIngredients { get; set; }

        public DbSet<LocalizedSpecific> LocalizedSpecifics { get; set; }

        public DbSet<LocalizedSubcategory> LocalizedSubcategories { get; set;
}

        public DbSet<UnitSet> UnitSets { get; set; }

        public DbSet<Country> Countries { get; set; }

        public DbSet<Language> Languages { get; set; }
    }
```

Інтерфейс IQueryObject

```
public interface IQueryObject<T>
 {
        Task<T> Execute(PostgresContext context);
 }
```

Клас QueryExecutor

```
public class QueryExecutor : IQueryExecutor
    {
        private readonly PostgresContext context;
        private readonly IMemoryCacheManager cache;

        public QueryExecutor(PostgresContext context, IMemoryCacheManager
cache)
        {
            this.context = context;
            this.cache = cache;
        }

        public async Task<T> Execute<T>(IQueryObject<T> query, bool autoSave
= true)
        {
            T data = await query.Execute(context);
            if (autoSave) await Save();

            return data;
        }
```

```
        public async Task<T> CachedRead<T>(IQueryObject<T> query, string
cacheKey) where T : class
        {
            T data = await cache.Get<T>(cacheKey, true);

            if (data == null)
            {
                data = await query.Execute(context);
                await cache.Set(cacheKey, data);
            }

            return data;
        }

        public IQueryable<T> GetBy<T>(Expression<Func<T, bool>> condition)
where T : PostgresEntityBase
            => condition == null ? context.Set<T>() :
context.Set<T>().Where(condition);

        public async Task<T> GetSingleBy<T>(Expression<Func<T, bool>>
condition) where T : PostgresEntityBase
            => await context.Set<T>().Where(condition).FirstOrDefaultAsync();

        public async Task<bool> ExistsBy<T>(Expression<Func<T, bool>>
condition) where T : PostgresEntityBase
            => await context.Set<T>().Where(condition).AnyAsync();

        public async Task<T> Create<T>(T entity, bool autoSave = true) where
T : PostgresEntityBase
        {
            var added = await context.Set<T>().AddAsync(entity);
            if (autoSave) await Save();
            return added.Entity;
        }

        public async Task CreateFew<T>(IEnumerable<T> entities, bool autoSave
= true) where T : PostgresEntityBase
        {
            await context.Set<T>().AddRangeAsync(entities);
            if (autoSave) await Save();
        }

        public async Task Update<T>(T entity, bool autoSave = true) where T :
PostgresEntityBase
        {
            context.Set<T>().Update(entity);
            if (autoSave) await Save();
        }

        public async Task Delete<T>(T entity, bool autoSave = true) where T :
PostgresEntityBase
        {
            context.Set<T>().Remove(entity);
            if (autoSave) await Save();
        }

        public async Task Save() => await context.SaveChangesAsync();
    }
```

Структура QueryObjects/AddComment

```
public readonly struct AddComment : IQueryObject<int>
    {
        private readonly Guid postId;

        public AddComment(Guid postId)
        {
            this.postId = postId;
        }

        public async Task<int> Execute(PostgresContext context)
        {
            FormattableString sql = $@"update ""Post"" set ""CommentsCount""
= ""CommentsCount"" + 1 where ""Id"" = {postId}";

            return await context.Database.ExecuteSqlInterpolatedAsync(sql);
        }
    }
```

Структура QueryObjects/DeleteComment

```
public readonly struct DeleteComment : IQueryObject<int>
    {
        private readonly Guid? userId;
        private readonly Guid commentId;
        private readonly Guid postId;

        public DeleteComment(Guid? userId, Guid commentId, Guid postId)
        {
            this.userId = userId;
            this.commentId = commentId;
            this.postId = postId;
        }

        public async Task<int> Execute(PostgresContext context)
        {
            FormattableString sql;
            if (userId != null) sql = $@"delete from ""Comment"" where
""UserId"" = {userId.Value} and ""Id"" = {commentId}";
            else sql = $@"delete from ""Comment"" where ""Id"" =
{commentId}";

            await context.Database.ExecuteSqlInterpolatedAsync(sql);

            sql = $@"update ""Post"" set ""CommentsCount"" =
""CommentsCount"" - 1 where ""Id"" = {postId}";

            return await context.Database.ExecuteSqlInterpolatedAsync(sql);
        }
    }
```

Структура QueryObjects/GetComments

```
public readonly struct GetComments : IQueryObject<List<CommentDto>>
    {
        private readonly GetCommentsRequest request;

        public GetComments(GetCommentsRequest request)
        {
            this.request = request;
        }

        public async Task<List<CommentDto>> Execute(PostgresContext context)
        {
            FormattableString sql = $@"select * from getComments(

                                        {request.FirstPagination()},
                                    {request.PostId},
                                    {request.PagingLimit},
                                    {request.UserId},
                                    {request.IdMark},
                                              {request.DateTimeMark},
                                              {request.InitialDateTime})";

            return await
context.OutputComments.FromSqlInterpolated(sql).ToListAsync();
        }
    }
```

Структура QueryObjects/CreatePost

```
public readonly struct CreatePost : IQueryObject<int>
    {
        private readonly IMapper mapper;
        private readonly Post post;
        private readonly PostRelatedForCreationDto relatedData;
        private readonly Guid userId;

        public CreatePost(IMapper mapper, Guid userId, Post post,
PostRelatedForCreationDto relatedData)
        {
            this.mapper = mapper;
            this.post = post;
            this.relatedData = relatedData;
            this.userId = userId;
        }

        public async Task<int> Execute(PostgresContext context)
        {
            var createdPost = (await
context.Set<Post>().AddAsync(post)).Entity;

            var postAssignments = relatedData.PostAssignments.Select(p => new
PostAssignment { AssignmentId = p.Id, PostId = createdPost.Id });

            await
context.Set<PostAssignment>().AddRangeAsync(postAssignments);

            var postSpecifics = relatedData.PostSpecifics.Select(p => new
PostSpecific { SpecificId = p.Id, PostId = createdPost.Id });
```

```
            await context.Set<PostSpecific>().AddRangeAsync(postSpecifics);

            IMapper mapper = this.mapper;

            var postIngredients = relatedData.PostIngredients.Select(p =>
            {
                var postIngredient = mapper.Map<PostIngredient>(p);
                postIngredient.PostId = createdPost.Id;
                postIngredient.IngredientId = p.Id;
                return postIngredient;
            });

            await
context.Set<PostIngredient>().AddRangeAsync(postIngredients);

            if(relatedData.PostDetails != null)
            {
                var postDetails = relatedData.PostDetails.Select(p =>
                new PostDetails
                {
                    Number = p.Number,
                    PostId = createdPost.Id,
                    Title = p.Title,
                    Description = p.Description
                });

                await context.Set<PostDetails>().AddRangeAsync(postDetails);
            }

            FormattableString sql = $@"update ""User""
                                       set ""LastPublicationTime"" =
{DateTime.Now},
                                           ""RecipesCount"" =
""RecipesCount"" + 1
                                       where ""UserId"" = {userId}";

            return await context.Database.ExecuteSqlInterpolatedAsync(sql);
        }
    }
```

Структура QueryObjects/GetPosts

```
public readonly struct GetPosts : IQueryObject<List<PostDto>>
    {
        private readonly GetPostsRequest postsRequest;
        private readonly IMemoryCacheManager cache;
        private readonly Guid? currentUserId;
        private readonly IConfiguration configuration;

        public GetPosts(GetPostsRequest postsRequest, Guid? currentUserId,
IMemoryCacheManager memoryCacheManager, IConfiguration configuration)
        {
            this.postsRequest = postsRequest;
            this.currentUserId = currentUserId;
            this.cache = memoryCacheManager;
            this.configuration = configuration;
        }

        public async Task<List<PostDto>> Execute(PostgresContext context)
        {
```

```
            var bookmarkedPostIds = await GetBookMarkedPostIds();

            var posts =  await
context.OutputPosts.FromSqlInterpolated(GetPostsSql(bookmarkedPostIds.Item1,
bookmarkedPostIds.Item2)).ToListAsync();

            foreach (var post in posts)
            {
                var postCached = await
cache.Get<PostCachedDto>(Constants.RelatedCacheKey(Constants.PostRelatedKeyPa
rt, post.Id), true);

                if (postCached != null)
                {
                    post.PreferencesCount += postCached.PreferencesCount;
                    post.BookmarksCount += postCached.BookmarksCount;

                    var cachedPostUser =
postCached.CurrentUser(currentUserId);

                    if (cachedPostUser != null)
                    {
                        if (post.PostActions == null)
                        {
                            post.PostActions = new List<bool?> {
cachedPostUser.Preferred, cachedPostUser.Bookmarked };
                        }
                        else
                        {
                            post.PostActions[0] = cachedPostUser.Preferred;
                            post.PostActions[1] = cachedPostUser.Bookmarked;
                        }
                    }
                }

                //post.PreviewKey = Constants.GetAwsPath(configuration,
post.PreviewKey);
                //post.UserPictureKey = Constants.GetAwsPath(configuration,
post.UserPictureKey);
            }

            return posts;
        }

        private async Task<(List<Guid>, List<Guid>)> GetBookMarkedPostIds()
        {
            if (!postsRequest.ShowCurrentBookmarked || currentUserId == null)
return (null, null);

            var keys = await cache.GetKeys(Constants.PostRelatedKeyPart +
"*");

            if (!keys.Any()) return (null, null);

            var postIdsForAdd = new List<Guid>();
            var postIdsForRemove = new List<Guid>();

            foreach(string key in keys)
            {
                var postCached = await cache.Get<PostCachedDto>(key);

                var cachedPostUser = postCached.CurrentUser(currentUserId);

                if(cachedPostUser != null)
```

```
                {
                    Guid postId = Constants.GetIdFromKey(key);

                    if (cachedPostUser.Bookmarked)
                    {
                        postIdsForAdd.Add(postId);
                    }
                    else
                    {
                        postIdsForRemove.Add(postId);
                    }
                }
            }

            return (
                postIdsForAdd.Count == 0 ? null : postIdsForAdd,
                postIdsForRemove.Count == 0 ? null : postIdsForRemove);
        }


        private FormattableString GetPostsSql(List<Guid>
bookmarkedPostIdsForAdd, List<Guid> bookmarkedPostIdsForRemove)
        {
            FormattableString sql = $@"select * from getPosts
                    (
                      {postsRequest.FirstPagination()},
                      {postsRequest.PagingLimit},
                      {currentUserId},
                      {postsRequest.Filter?.TargetUserId},
                      {postsRequest.RecipeBookId},
                      {bookmarkedPostIdsForAdd}::uuid[],
                      {bookmarkedPostIdsForRemove}::uuid[],
                      {postsRequest.ShowCurrentPreferred},
                      {postsRequest.ShowCurrentBookmarked},
                      {postsRequest.Filter?.IncludedIngredients}::uuid[],
                      {postsRequest.Filter?.ExcludedIngredients}::uuid[],
                      {postsRequest.Filter?.Specifics}::uuid[],
                      {postsRequest.Filter?.Assignments}::uuid[],
                      {postsRequest.Filter?.Category},
                      {postsRequest.Filter?.Subcategory},
                      {postsRequest.Filter?.Cuisine},
                      {postsRequest.Filter?.CalorificFrom},
                      {postsRequest.Filter?.CalorificTo},
                      {postsRequest.Filter?.Complexities}::int[],
                      {postsRequest.Filter?.StepByStep},
                      {postsRequest.Filter?.WithVideo},
                      {postsRequest.Filter?.WithMulticooker},
                      {postsRequest.Filter?.HealthyFood},
                      {postsRequest.Filter?.CookTimeFrom},
                      {postsRequest.Filter?.CookTimeTo},
                      {postsRequest.Locale},
                      {postsRequest.SearchTerm.ToSearchTerm()},
                      {postsRequest.IdMark},
                      {postsRequest.DateTimeMark},
                      {postsRequest.InitialDateTime}
                    )";

            return sql;
        }
    }
```

Структура QueryObjects/GetPostsForModerator

```
public readonly struct GetPostsForModerator :
IQueryObject<List<PostForModerationDto>>
    {
        private readonly GetPostsForModeratorRequest postsRequest;
        private readonly bool showPostsOnModeration;

        public GetPostsForModerator(GetPostsForModeratorRequest postsRequest,
bool showPostsOnModeration)
        {
            this.postsRequest = postsRequest;
            this.showPostsOnModeration = showPostsOnModeration;
        }

        public async Task<List<PostForModerationDto>> Execute(PostgresContext
context)
        {
            FormattableString sql = $@"select * from getPostsForModerator
                    (
                      {postsRequest.FirstPagination()},
                      {postsRequest.PagingLimit},
                      {showPostsOnModeration},
                      {postsRequest.TargetUserId},
                      {postsRequest.Locale},
                      {postsRequest.SearchTerm.ToSearchTerm()},
                      {postsRequest.IdMark},
                      {postsRequest.DateTimeMark},
                      {postsRequest.InitialDateTime}
                    )";

            return await
context.PostsForModeration.FromSqlInterpolated(sql).ToListAsync();
        }
    }
```

Структура QueryObjects/GetUserInfo

```
public readonly struct GetUserInfo : IQueryObject<UserIdentityDto>
    {
        private readonly Guid userId;
        private readonly IMapper mapper;
        private readonly IConfiguration configuration;
        private readonly IMemoryCacheManager cache;

        public GetUserInfo(Guid userId, IMapper mapper, IConfiguration
configuration, IMemoryCacheManager cache)
        {
            this.userId = userId;
            this.mapper = mapper;
            this.configuration = configuration;
            this.cache = cache;
        }

        public async Task<UserIdentityDto> Execute(PostgresContext context)
        {
            var userId = this.userId;
```

```
            var userIdenity = await context.UserIdentities.Where(u => u.Id ==
userId)
                .Include(u => u.User)
                .FirstOrDefaultAsync();

            var user = mapper.Map<UserIdentityDto>(userIdenity);

            var userSettings = await context.UserSettings.Where(u => u.Id ==
userIdenity.Id).FirstOrDefaultAsync();

            if (userSettings != null)
            {
                user.UserSettings = userSettings;
            }

            if(user.User != null)
            {
                //user.User.PictureKey = Constants.GetAwsPath(configuration,
user.User.PictureKey);
            }


            var keys = await cache.GetKeys(Constants.UserRelatedKeyPart +
"*");


            foreach(string key in keys)
            {
                if(Constants.GetIdFromKey(key) == userId)
                {
                    var stat = await cache.Get<UserStatisitcsDto>(key);

                    user.User.BookmarksCount += stat.BookmarksCount;

                    break;
                }
            }

            return user;
        }
    }
```

Клас QuartzJobRunner

```
public class QuartzJobRunner : IJob
    {
        private readonly IServiceProvider serviceProvider;
        public QuartzJobRunner(IServiceProvider serviceProvider)
        {
            this.serviceProvider = serviceProvider;
        }

        public async Task Execute(IJobExecutionContext context)
        {
            using var scope = serviceProvider.CreateScope();
            var jobType = context.JobDetail.JobType;
            var job = scope.ServiceProvider.GetRequiredService(jobType) as
IJob;

            await job.Execute(context);
        }
```

```
    }
```

Клас QuartzHostedService

```
public class QuartzHostedService : IHostedService
    {
        private readonly ISchedulerFactory schedulerFactory;
        private readonly IJobFactory jobFactory;
        private readonly IEnumerable<JobSchedule> jobSchedules;

        public QuartzHostedService(
            ISchedulerFactory schedulerFactory,
            IJobFactory jobFactory,
            IEnumerable<JobSchedule> jobSchedules)
        {
            this.schedulerFactory = schedulerFactory;
            this.jobSchedules = jobSchedules;
            this.jobFactory = jobFactory;
        }
        public IScheduler Scheduler { get; set; }

        public async Task StartAsync(CancellationToken cancellationToken)
        {
            Scheduler = await
schedulerFactory.GetScheduler(cancellationToken);
            Scheduler.JobFactory = jobFactory;

            foreach (var jobSchedule in jobSchedules)
            {
                var job = CreateJob(jobSchedule);
                var trigger = CreateTrigger(jobSchedule);

                await Scheduler.ScheduleJob(job, trigger, cancellationToken);
            }

            await Scheduler.Start(cancellationToken);
        }

        public async Task StopAsync(CancellationToken cancellationToken)
        {
            await Scheduler?.Shutdown(cancellationToken);
        }

        private static IJobDetail CreateJob(JobSchedule schedule)
        {
            var jobType = schedule.JobType;
            return JobBuilder
                .Create(jobType)
                .WithIdentity(jobType.FullName)
                .WithDescription(jobType.Name)
                .Build();
        }

        private static ITrigger CreateTrigger(JobSchedule schedule)
        {
            return TriggerBuilder
                .Create()
                .WithIdentity($"{schedule.JobType.FullName}.trigger")
                .WithCronSchedule(schedule.CronExpression)
                .WithDescription(schedule.CronExpression)
                .Build();
```

```
        }
    }
```

Клас FoodiesJobFactory

```
public class FoodiesJobFactory : IJobFactory
    {
        private readonly IServiceProvider serviceProvider;

        public FoodiesJobFactory(IServiceProvider serviceProvider) =>
this.serviceProvider = serviceProvider;

        public IJob NewJob(TriggerFiredBundle bundle, IScheduler scheduler)
=> serviceProvider.GetRequiredService<QuartzJobRunner>();

        public void ReturnJob(IJob job) { }
    }
```

Клас MarkUserAsRemovedJob

```
[DisallowConcurrentExecution]
    public class MarkUserAsRemovedJob : JobBase
    {
        private readonly IQueryExecutor executor;
        private readonly ILoggerManager logger;

        public MarkUserAsRemovedJob(IQueryExecutor executor, ILoggerManager
logger)
        {
            this.executor = executor;
            this.logger = logger;
        }

        public override async Task Execute(IJobExecutionContext context)
        {
            try
            {
                await executor.Execute(new UpdateMarkUserAsRemoved());
            }
            catch (Exception ex)
            {
                logger.LogError($"{nameof(MarkUserAsRemovedJob)} |
{DateTime.Now} \n {ex.Message}");
            }
        }
    }
```

Клас UpdatePostRelatedDataJob

```
[DisallowConcurrentExecution]
    public class UpdatePostRelatedDataJob : JobBase
    {
        private readonly IMemoryCacheManager cache;
        private readonly IQueryExecutor executor;
        private readonly ILoggerManager logger;
```

```
        public UpdatePostRelatedDataJob(IQueryExecutor executor,
ILoggerManager logger, IMemoryCacheManager cache)
        {
            this.executor = executor;
            this.logger = logger;
            this.cache = cache;
        }

        public override async Task Execute(IJobExecutionContext context)
        {
            try
            {
                var postKeys = await
cache.GetKeys(Constants.PostExpirationKeyPart + "*");
                var userKeys = await
cache.GetKeys(Constants.UserExpirationKeyPart + "*");

                var filteredPostKeys = FilterKeys(postKeys);
                var filteredUserKeys = FilterKeys(userKeys);


                if(!filteredPostKeys.Any() && !filteredUserKeys.Any())
                {
                    return;
                }

                foreach(string key in filteredUserKeys)
                {
                    var (id, entity) = await ActCache<UserStatisitcsDto>(key,
Constants.UserRelatedKeyPart);

                    if (entity != null)
                    {
                        await executor.Execute(new UpdateUserStatistics(id,
entity.BookmarksCount));
                    }
                }

                foreach(string key in filteredPostKeys)
                {
                    var (id, entity) = await ActCache<PostCachedDto>(key,
Constants.PostRelatedKeyPart);

                    if(entity != null)
                    {
                        await executor.Execute(new UpdatePostRelatedData(id,
entity));
                    }
                }
            }
            catch (Exception ex)
            {
                logger.LogError($"{nameof(UpdatePostRelatedDataJob)} |
{DateTime.Now} \n {ex.Message}");
            }
        }

        private async Task<(Guid id, T entity)> ActCache<T>(string key,
string keyPart) where T: class
        {
            Guid id = Constants.GetIdFromKey(await cache.Get<string>(key));

            await cache.Remove(key);
```

```
            var _key = Constants.RelatedCacheKey(keyPart, id);

            var cached = await cache.Get<T>(_key);

            await cache.Remove(_key);

            return (id, cached);
        }

        private IEnumerable<string> FilterKeys(IEnumerable<string> keys)
        {
            var filteredKeys = new List<string>();

            foreach (string k in keys)
            {
                if (DateTime.Now >= Constants.GetExpirationFromKey(k))
                {
                    filteredKeys.Add(k);
                }
            }

            return filteredKeys;
        }
    }
```

Клас AuthManager

```
public class AuthManager : IAuthManager
    {
        private readonly IConfiguration configuration;

        public AuthManager(IConfiguration configuration)
        {
            this.configuration = configuration;
        }

        public string CreateAccessToken(UserIdentity user)
        {
            var key =
Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariable(Constants.secret));
            var secret = new SymmetricSecurityKey(key);
            var signingCredentials = new SigningCredentials(secret,
SecurityAlgorithms.HmacSha256);

            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                new Claim(ClaimTypes.Role, user.UserType.ToString()),
            };

            var jwtSettings =
configuration.GetSection(Constants.jwtSettings);

            var tokenOptions = new JwtSecurityToken
            (
                issuer: jwtSettings.GetSection(Constants.validIssuer).Value,
                audience:
jwtSettings.GetSection(Constants.validAudience).Value,
                claims: claims,
```

```
                expires:
DateTime.Now.AddMinutes(Convert.ToDouble(jwtSettings.GetSection(Constants.acc
essTokenExpiresMinutes).Value)),
                signingCredentials: signingCredentials
            );

            return new JwtSecurityTokenHandler().WriteToken(tokenOptions);
        }

        public string CreateRefreshToken()
        {
            var randomNumber = new byte[32];
            using var rng = RandomNumberGenerator.Create();
            rng.GetBytes(randomNumber);
            return Convert.ToBase64String(randomNumber);
        }

        public ClaimsPrincipal GetPrincipalFromExpiredToken(string token)
        {
            var tokenValidationParameters = new TokenValidationParameters
            {
                ValidateAudience = false,
                ValidateIssuer = false,
                ValidateIssuerSigningKey = true,
                IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(Environment.GetEnvironmentVariabl
e(Constants.secret))),
                ValidateLifetime = false
            };

            var tokenHandler = new JwtSecurityTokenHandler();

            var principal = tokenHandler.ValidateToken(token,
tokenValidationParameters, out SecurityToken securityToken);

            if (securityToken is not JwtSecurityToken jwtSecurityToken ||
!jwtSecurityToken.Header.Alg.Equals(SecurityAlgorithms.HmacSha256,
StringComparison.InvariantCultureIgnoreCase))
                throw new SecurityTokenException(Constants.InvalidTokenMsg);

            return principal;
        }
    }
```

Клас AWSS3BucketManager

```
public class AWSS3BucketManager : IAWSS3BucketManager
    {
        private readonly IAmazonS3 amazonS3;
        private readonly string bucketName;
        private readonly IConfiguration configuration;

        public AWSS3BucketManager(IAmazonS3 amazonS3, IConfiguration
configuration)
        {
            this.amazonS3 = amazonS3;
            bucketName =
configuration.GetSection(Constants.AWSS3).GetSection(Constants.bucketName).Va
lue;
            this.configuration = configuration;
        }
```

```
        public async Task<bool> UploadFile(IFormFile file, string fileName)
        {
            using var stream = new MemoryStream();

            await file.CopyToAsync(stream);

            var request = new PutObjectRequest()
            {
                InputStream = stream,
                BucketName = bucketName,
                Key = fileName,
                CannedACL = S3CannedACL.PublicRead
            };

            PutObjectResponse response = await
amazonS3.PutObjectAsync(request);
            return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
        }

        public async Task<bool> DeleteFile(string key)
        {
            DeleteObjectResponse response = await
amazonS3.DeleteObjectAsync(bucketName, key);
            return response.HttpStatusCode ==
System.Net.HttpStatusCode.NoContent;
        }

        public async Task<Stream> GetFile(string key)
        {
            GetObjectResponse response = await
amazonS3.GetObjectAsync(bucketName, key);

            return response.HttpStatusCode == System.Net.HttpStatusCode.OK
                ? response.ResponseStream : null;
        }
    }
```

Клас MemoryCacheManager

```
public class MemoryCacheManager : IMemoryCacheManager
    {
        private readonly IRedisCacheClient cache;

        public MemoryCacheManager(IRedisCacheClient cache) => this.cache =
cache;

        public async Task<bool> KeyExists(string key)
            => await cache.Db0.ExistsAsync(key);

        public async Task<T> Get<T>(string key, bool checkIfExists = false)
where T : class
           => checkIfExists
                ? (await KeyExists(key) ? await cache.Db0.GetAsync<T>(key) :
null)
                : await cache.Db0.GetAsync<T>(key);


        public async Task<IEnumerable<string>> GetKeys(string pattern)
            => await cache.Db0.SearchKeysAsync(pattern);
```

```
        public async Task Set<T>(string key, T value, TimeSpan?
absoluteExpiration = null) where T : class
        {
            await cache.Db0.AddAsync(key, value, absoluteExpiration == null ?
Constants.AbsoluteExpiration : absoluteExpiration.Value,
StackExchange.Redis.When.NotExists);
        }

        public async Task Remove(params string[] keys)
        {
            foreach(string key in keys)
            {
                await cache.Db0.RemoveAsync(key);
            }
        }
    }
```

Клас FoodiesControllerBase

```
public class FoodiesControllerBase : ControllerBase
    {
        protected Guid CurrentUserId
            => User == null || !User.Claims.Any()
            ? default
            : new(User.Claims.Where(c => c.Type ==
ClaimTypes.NameIdentifier).FirstOrDefault().Value);

        protected UserType CurrentUserType =>
(UserType)Enum.Parse(typeof(UserType), User.Claims.Where(c => c.Type ==
ClaimTypes.Role).FirstOrDefault().Value);

        protected IActionResult GetModelStateWithErrors(IdentityResult
result)
        {
            result.Errors.ToList().ForEach(error =>
                ModelState.TryAddModelError(error.Code, error.Description));

            return BadRequest(ModelState);
        }

        protected bool IsSuperuser() => CurrentUserType ==
UserType.SUPERUSER;

        protected bool IsAdministrator() => CurrentUserType ==
UserType.ADMINISTRATOR;

        protected bool IsModerator() => IsSuperuser() || IsAdministrator();

    }
```

Клас UsersControllerBase

```
public class UsersControllerBase : FoodiesControllerBase
    {
        private readonly IMemoryCacheManager cache;
        private readonly IQueryExecutor executor;
```

```
        public UsersControllerBase(IMemoryCacheManager cache, IQueryExecutor
executor)
        {
            this.cache = cache;
            this.executor = executor;
        }

        protected async Task<bool> VerifyCacheData(string accessCode, string
email)
        {
            if (!await cache.KeyExists(accessCode))
            {
                ModelState.AddModelError(Constants.UserEmailNotFoundKey,
Constants.UserEmailNotFoundMsg);
                return false;
            }

            string hashedEmail = await cache.Get<string>(accessCode);

            if (string.IsNullOrEmpty(hashedEmail) ||
!BCrypt.Net.BCrypt.Verify(email, hashedEmail))
            {
                ModelState.AddModelError(Constants.AccessCodeIncorrectKey,
Constants.AccessCodeIncorrectMsg);
                return false;
            }

            await cache.Remove(accessCode, email);

            return true;
        }

        protected async Task<bool> UserExists(string email, string userName)
        {
            bool exists = await executor.Execute(new CheckIfUserExists(email,
userName), false);

            if (exists)
            {
                ModelState.AddModelError(Constants.UserWithEmailExistsKey,
Constants.UserWithEmailExistsMsg);
                return true;
            }

            return false;
        }
    }
```

Клас UsersController

```
[Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController : UsersControllerBase
    {
        private readonly IMemoryCacheManager cache;
        private readonly IMapper mapper;
        private readonly IQueryExecutor executor;
        private readonly IAWSS3BucketManager s3Manager;
        private readonly IConfiguration configuration;
```

```
        public UsersController(IMemoryCacheManager cache, IMapper mapper,
IQueryExecutor executor, IAWSS3BucketManager s3Manager, IConfiguration
configuration)
            : base(cache, executor)
        {
            this.cache = cache;
            this.mapper = mapper;
            this.executor = executor;
            this.s3Manager = s3Manager;
            this.configuration = configuration;
        }

        [HttpGet("reportedUsers")]
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        public async Task<ActionResult<ReportedUserDto>>
GetReportedUsers([FromBody] GetReportedUserRequest request)
        {
            var reportedUsers = await executor.Execute(new
GetReportedUsers(request), false);

            //reportedUsers.ForEach(u => u.PictureKey =
Constants.GetAwsPath(configuration, u.PictureKey));

            return Ok(reportedUsers);
        }

        [AllowAnonymous]
        [HttpGet("searchUsers")]
        public async Task<ActionResult<ResponseUserDto>>
SearchUsers([FromBody] GetLastActiveUsersRequest request)
        {
            var lastActiveUsers = await executor.Execute(new
GetLastActiveUsers(request), false);

            //lastActiveUsers.ForEach(u => u.PictureKey =
Constants.GetAwsPath(configuration, u.PictureKey));

            return Ok(lastActiveUsers);
        }

        [HttpGet("userInfo")]
        public async Task<IActionResult> GetUserInfo()
        {
            var user = await executor.Execute(new GetUserInfo(CurrentUserId,
mapper, configuration, cache), false);

            return Ok(user);
        }

        [AllowAnonymous]
        [HttpPost("sendAccessCode")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> SendAccessCode([FromBody]
UserForSendingAccessCodeDto userData)
        {
            string accessCode = Hashing.GenerateCharSet();

            string locale = userData.Locale ?? userData.Language;

            string hashedEmail =
BCrypt.Net.BCrypt.HashPassword(userData.Email);

            if (await cache.KeyExists(userData.Email))
            {
```

```
                ModelState.AddModelError(Constants.AccessCodeAlreadySentKey,
Constants.AccessCodeAlreadySentMsg);
                return BadRequest(ModelState);
            }

            var emailBody = Mailing.GetWelcomeEmailBody(locale, accessCode);

            await Mailing.SendEmail(configuration, userData.UserName,
userData.Email, emailBody);

            if (!await cache.KeyExists(accessCode))
            {
                await cache.Set(userData.Email, string.Empty);
                await cache.Set(accessCode, hashedEmail);
            }

            return Ok();
        }

        [AllowAnonymous]
        [HttpPost]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> CreateUser([FromBody]
BaseUserForRegistrationDto data)
        {
            if (!await VerifyCacheData(data.AccessCode, data.Email) || await
UserExists(data.Email, data.UserName))
            {
                return BadRequest(ModelState);
            }

            var userIdentity = mapper.Map<UserIdentity>(data);

            userIdentity.UserType = UserType.USER;

            userIdentity.PasswordHash =
BCrypt.Net.BCrypt.HashPassword(data.Password);

            var user = mapper.Map<User>(data);

            var userSettings = new UserSettings
            {
                Language = data.Language ?? data.Locale,
                ShowLastActiveUsers = true,
                ShowRecipesDependOnDayTime = true
            };

            await executor.Execute(new CreateUser(userIdentity, user,
userSettings));

            return StatusCode(201);
        }


        [HttpPost("createPrivilagedUser")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> CreatePrivilagedUser([FromBody]
PrivilagedUserForCreationDto userData)
        {
            if (!IsSuperuser() || userData.UserType == UserType.SUPERUSER ||
await UserExists(userData.Email, userData.UserName))
            {
                return BadRequest(ModelState);
            }
```

```
            var newUser = mapper.Map<UserIdentity>(userData);

            string password = Hashing.GenerateCharSet(12).ToLower();

            var emailBody = Mailing.GetWelcomeEmailBody(userData.Locale ??
userData.Language, password, true);

            await Mailing.SendEmail(configuration, userData.UserName,
userData.Email, emailBody);

            newUser.PasswordHash = BCrypt.Net.BCrypt.HashPassword(password);

            var user = mapper.Map<User>(userData);

            await executor.Execute(new CreateUser(newUser,
userData.CreateFoodiesUser ? user : null, userData.CreateFoodiesUser ? new
UserSettings() : null));

            return StatusCode(201);
        }

        [HttpPut("updatePicture")]
        public async Task<IActionResult> UpdatePicture([FromForm(Name =
"file")] [UploadedImageFile(Constants.MaxUserPictureSize)] IFormFile picture)
        {
            var user = await executor.GetSingleBy<User>(x => x.UserId ==
CurrentUserId);

            if (user == null)
                return NotFound();

            string fileName = user.PictureKey ??
FileUtils.GenerateFileName(picture.FileName, Constants.userPicturesFolder);

            string fileUrl = Constants.GetAwsPath(configuration, fileName);

            await s3Manager.UploadFile(picture, fileName);

            if(user.PictureKey == null)
            {
                await executor.Execute(new UpdateUserPicture(CurrentUserId,
fileName));
            }

            return Ok(new { FileUrl = fileUrl });
        }

        [HttpPut("deletePicture")]
        public async Task<IActionResult> DeletePicture()
        {
            var user = await executor.GetSingleBy<User>(x => x.UserId ==
CurrentUserId);

            if (user == null || user.PictureKey == null)
                return NotFound();

            await s3Manager.DeleteFile(user.PictureKey);

            await executor.Execute(new UpdateUserPicture(CurrentUserId));

            return Ok();
        }
```

```
        [HttpPut("updateCredentials")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdateCredentials([FromBody]
UserForUpdateProfileDataDto data)
        {
            var currentUserIdentity = await executor.GetBy<UserIdentity>(ui
=> ui.Id == CurrentUserId).Include(u => u.User).FirstOrDefaultAsync();

            if (currentUserIdentity == null ||
(currentUserIdentity.UserIdentityName != data.UserName && await
UserExists(string.Empty, data.UserName)))
            {
                return BadRequest(ModelState);
            }

            var updatedUserIdentity = mapper.Map(data, currentUserIdentity);

            updatedUserIdentity.UserIdentityName = data.UserName;

            var updatedUser = mapper.Map(data, currentUserIdentity.User);

            await executor.Update(updatedUserIdentity, false);
            await executor.Update(updatedUser);

            return Ok();
        }

        [HttpPut("updatePassword")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdateIdentity([FromBody]
UserForUpdatePasswordDto data)
        {
            var user = await executor.GetSingleBy<UserIdentity>(u => u.Id ==
CurrentUserId);

            if(user == null || IsSuperuser())
            {
                return BadRequest(Constants.InvalidRequestMsg);
            }

            if (!BCrypt.Net.BCrypt.Verify(data.OldPassword,
user.PasswordHash))
            {
                ModelState.AddModelError("oldPasswordsDifferent", "");
                return BadRequest(ModelState);
            }

            user.PasswordHash =
BCrypt.Net.BCrypt.HashPassword(data.NewPassword);

            await executor.Update(user);

            return Ok();
        }


        [HttpPut("updateEmail")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdateIdentity([FromBody]
UserForUpdateEmailDto data)
        {
            var user = await executor.GetSingleBy<UserIdentity>(u => u.Id ==
CurrentUserId);
```

```
            if(await UserExists(data.Email, string.Empty))
            {
                return BadRequest(ModelState);
            }

            bool verified = await VerifyCacheData(data.AccessCode,
data.Email);

            if (user == null || IsSuperuser() || !verified)
            {
                return BadRequest(Constants.InvalidRequestMsg);
            }

            user.Email = data.Email;

            await executor.Update(user);

            return Ok();
        }

        [HttpPut("markForDelete")]
        public async Task<IActionResult> MarkForDelete(UserForUpdateEmailDto
data)
        {
            var verified = await VerifyCacheData(data.AccessCode,
data.Email);

            if (IsModerator() || !verified)
            {
                return BadRequest();
            }

            await executor.Execute(new UpdateMarkForDeleteUser(CurrentUserId,
true));


            return Ok();
        }

        [AllowAnonymous]
        [HttpPut("restore")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> Restore([FromBody] UserForRestoreDto
data)
        {
            if (!await VerifyCacheData(data.AccessCode, data.Email))
            {
                return BadRequest(ModelState);
            }

            var currentUserIdentity = await
executor.GetSingleBy<UserIdentity>(x => x.Email == data.Email);

            if (currentUserIdentity == null)
            {
                ModelState.AddModelError(Constants.UserEmailNotFoundKey,
Constants.UserEmailNotFoundMsg);
                return NotFound(ModelState);
            }

            if(currentUserIdentity.EndBlock != null)
            {
                ModelState.AddModelError("userBlocked", "User has been
blocked.");
```

```
                return BadRequest(ModelState);
            }

            if(!BCrypt.Net.BCrypt.Verify(data.Password,
currentUserIdentity.PasswordHash))
            {
                ModelState.AddModelError("emailPasswordIncorrect", "");
                return BadRequest(ModelState);
            }

            await executor.Execute(new
UpdateMarkForDeleteUser(currentUserIdentity.Id, false));

            return Ok();
        }

        [AllowAnonymous]
        [HttpPut("forgetPassword")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> ForgetPassword([FromBody]
UserForRestoreDto data)
        {
            if (!await VerifyCacheData(data.AccessCode, data.Email))
            {
                return BadRequest(ModelState);
            }

            var currentUserIdentity = await
executor.GetSingleBy<UserIdentity>(x => x.Email == data.Email);

            if (currentUserIdentity == null)
            {
                ModelState.AddModelError(Constants.UserEmailNotFoundKey,
Constants.UserEmailNotFoundMsg);
                return NotFound(ModelState);
            }

            if (currentUserIdentity.EndBlock != null)
            {
                ModelState.AddModelError("userBlocked", "User has been
blocked.");
                return BadRequest(ModelState);
            }

            if (currentUserIdentity.MarkedForDeleteAt != null)
            {
                ModelState.AddModelError("userRemoved", "User has been
removed. You can restore it.");
                return BadRequest(ModelState);
            }

            string passwordHash =
BCrypt.Net.BCrypt.HashPassword(data.Password);

            currentUserIdentity.PasswordHash = passwordHash;
            currentUserIdentity.RefreshToken = null;
            currentUserIdentity.RefreshTokenExpiration = null;

            await executor.Update(currentUserIdentity);

            return Ok();
        }

        [HttpPut("block")]
```

```
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> BlockUser([FromBody] BlockUserDto
data)
        {
            using var transaction = new
TransactionScope(TransactionScopeAsyncFlowOption.Enabled);

            var userIdentity = await executor.GetSingleBy<UserIdentity>(x =>
x.Email.Equals(data.Email));

            if (userIdentity == null || userIdentity.UserType ==
UserType.SUPERUSER)
            {
                return BadRequest(Constants.InvalidRequestMsg);
            }

            await executor.Execute(new UpdateBlockUser(userIdentity.Id,
data.Duration));

            transaction.Complete();

            return Ok();
        }

        [HttpPost("changeSubscription")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> AddSubscribtion([FromQuery] Guid
userId, [FromQuery] bool subscribe)
        {
            var result = await executor.Execute(new
ChangeSubscription(CurrentUserId, userId, subscribe));

            if(result == 0)
            {
                return BadRequest();
            }

            return StatusCode(201);
        }

        [HttpDelete("removeSubscription")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> RemoveSubscribtion([FromQuery] Guid
userId)
        {
            var userFollow = await executor.GetSingleBy<UserFollow>(u =>
u.FollowerId == CurrentUserId && u.FollowingId == userId);
            await executor.Delete(userFollow);
            return Ok();
        }

        [HttpPut("report")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> Report([FromQuery] Guid userId)
        {
            await executor.Execute(new UpdateUserReports(userId,
CurrentUserId));
            return Ok();
        }

        [HttpPut("unblock")]
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
```

```
        public async Task<IActionResult> Unblock([FromQuery] Guid userId)
        {
            await executor.Execute(new UpdateUserReports(userId,
CurrentUserId, true));
            return Ok();
        }

        [HttpPut("updateSettings")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdateSettings([FromBody]
UserSettingsDto settings)
        {
            var currentSettings = await executor.GetSingleBy<UserSettings>(x
=> x.Id == CurrentUserId);

            if(currentSettings == null)
            {
                return BadRequest();
            }

            var newSettings = mapper.Map(settings, currentSettings);

            await executor.Update(newSettings);

            return Ok();
        }

        [AllowAnonymous]
        [HttpPost("EBD37019-8F21-4A74-B8A8-A7EFE7A43BAE")]
        public async Task<IActionResult> CreateSuperuser([FromQuery] string
superuserSecret)
        {
            var (email, password, secret) = Constants.GetCreds(configuration,
Constants.SuperuserCredentials);

            if (superuserSecret.Equals(secret))
            {
                var superUser = new UserIdentity
                {
                    Id = Constants.superuserId,
                    UserIdentityName = Constants.superuser,
                    Email = email,
                    LastLoggedAt = DateTime.Now,
                    EmailConfirmed = true,
                    UserType = UserType.SUPERUSER,
                    PasswordHash = BCrypt.Net.BCrypt.HashPassword(password),
                    EndBlock = null
                };

                await executor.Execute(new CreateUser(superUser));
            }

            return StatusCode(201);
        }
    }
```

Клас PostsController

```
[Authorize]
    [Route("api/[controller]")]
    [ApiController]
    public class PostsController : FoodiesControllerBase
    {
        private readonly IMemoryCacheManager cache;
        private readonly IMapper mapper;
        private readonly IQueryExecutor executor;
        private readonly IAWSS3BucketManager s3Manager;
        private readonly IConfiguration configuration;

        public PostsController(IMemoryCacheManager cache, IMapper mapper,
IQueryExecutor executor, IAWSS3BucketManager s3Manager, IConfiguration
configuration)
        {
            this.cache = cache;
            this.mapper = mapper;
            this.executor = executor;
            this.s3Manager = s3Manager;
            this.configuration = configuration;
        }

        [AllowAnonymous]
        [HttpGet("details")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> GetPostDetails([FromQuery] Guid
postId, [FromQuery] string locale)
        {
            if (CurrentUserId != default)
            {
                return StatusCode(405);
            }

            var details = await executor.CachedRead(new
GetPostDetails(postId, default, locale),
Constants.RelatedCacheKey(Constants.PostDetailsPart, postId));

            return Ok(details);
        }

        [HttpGet("details/auth")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> GetPostDetailsAuth([FromQuery] Guid
postId, [FromQuery] string locale)
        {
            var details = await executor.CachedRead(new
GetPostDetails(postId, CurrentUserId, locale),
Constants.RelatedCacheKey(Constants.PostDetailsPart, postId));
            return Ok(details);
        }

        [AllowAnonymous]
        [HttpPost("get")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> GetPosts([FromBody] GetPostsRequest
request)
        {
            var posts = await executor.Execute(
                new GetPosts(request, CurrentUserId == default ? null :
CurrentUserId, cache, configuration), false);
```

```
            return Ok(posts);
        }

        [HttpPost("postsOnModeration")]
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> GetPostsOnModeration([FromBody]
GetPostsForModeratorRequest request)
        {
            var posts = await executor.Execute(new
GetPostsForModerator(request, true), false);

            posts.ForEach(post =>
            {
                post.PreviewKey = Constants.GetAwsPath(configuration,
post.PreviewKey);
                post.UserPictureKey = Constants.GetAwsPath(configuration,
post.UserPictureKey);
            });

            return Ok(posts);
        }

        [HttpPost]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> CreatePost
            ([ModelBinder(BinderType = typeof(JsonModelBinder))]
PostForCreationDto data,
            [ModelBinder(BinderType = typeof(JsonModelBinder))]
PostRelatedForCreationDto relatedData,
            [UploadedImageFile(Constants.MaxPostPictureSize)] IFormFile file)
        {
            var post = mapper.Map<Post>(data);

            post.UserId = CurrentUserId;
            post.CommentsCount = 0;
            post.CreatedAt = DateTime.Now;

            post.Status = data.Private ? PostStatus.ACTIVE :
PostStatus.ONMODERATION;

            string previewKey = FileUtils.GenerateFileName(file.FileName,
Constants.postMediaFolder);

            await s3Manager.UploadFile(file, previewKey);

            post.PreviewKey = previewKey;

            await executor.Execute(new CreatePost(mapper, CurrentUserId,
post, relatedData));

            string fileUrl = Constants.GetAwsPath(configuration, previewKey);

            return StatusCode(201, new { FileUrl = fileUrl });
        }

        [HttpPut]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdatePost
            ([FromQuery] Guid? postId, [ModelBinder(BinderType =
typeof(JsonModelBinder))] PostForCreationDto data,
            [ModelBinder(BinderType = typeof(JsonModelBinder))]
PostRelatedForUpdateDto includeRelatedData,
```

```
            [ModelBinder(BinderType = typeof(JsonModelBinder))]
PostRelatedForUpdateDto excludedRelatedData)
        {
            if(postId == null)
            {
                return BadRequest();
            }

            var existedPost = await executor.GetSingleBy<Post>(p => p.Id ==
postId && p.UserId == CurrentUserId);

            if (existedPost == null)
            {
                return BadRequest();
            }

            var post = mapper.Map(data, existedPost);

            post.Status = PostStatus.ONMODERATION;

            await executor.Update(post, includeRelatedData == null &&
excludedRelatedData == null);

            await executor.Execute(new UpdatePost(mapper, postId.Value,
includeRelatedData, excludedRelatedData));

            return Ok();
        }

        [HttpPut("updatePreview")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> UpdatePreview([FromQuery] Guid
postId, [UploadedImageFile(Constants.MaxPostPictureSize)] IFormFile file)
        {
            var existedPost = await executor.GetSingleBy<Post>(p => p.Id ==
postId && p.UserId == CurrentUserId);

            if (existedPost == null)
            {
                return BadRequest();
            }

            await s3Manager.UploadFile(file, existedPost.PreviewKey);

            return Ok();
        }

        [HttpPut("approve")]
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> ApprovePost([FromBody]
ApprovePostDto data)
        {
            await executor.Execute(new UpdatePostStatus(data.PostId,
data.UserId, data.CreatedAt));

            var emailBody = Mailing.GetApprovedPostEmailBody(data.Locale,
data.PostTitle);

            await Mailing.SendEmail(configuration, data.UserName, data.Email,
emailBody);

            return Ok();
        }
```

```
        [HttpPut("changePostAction")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> ChangePostAction([FromQuery] Guid
postId, [FromQuery] bool preferred, [FromQuery] bool bookmarked)
        {
            string postKey =
Constants.RelatedCacheKey(Constants.PostRelatedKeyPart, postId);

            string userKey =
Constants.RelatedCacheKey(Constants.UserRelatedKeyPart, CurrentUserId);

            var fromCache = await cache.Get<PostCachedDto>(postKey, true);

            var fromCacheStatistics = await
cache.Get<UserStatisitcsDto>(userKey, true);

            var postCached = fromCache ?? new();

            var statisitcCached = fromCacheStatistics ?? new();

            await cache.Remove(postKey);

            await cache.Remove(userKey);

            PublicRelatedDto user = postCached.CurrentUser(CurrentUserId);

            if (user == null)
            {
                var postAction = await executor.GetSingleBy<PostAction>(x =>
x.UserId == CurrentUserId && x.PostId == postId);

                if(postAction == null && bookmarked)
                {
                    await executor.Create(new PostAction
                    {
                        UserId = CurrentUserId,
                        PostId = postId,
                        Bookmarked = bookmarked,
                        Preferred = preferred
                    });
                }

                var newUser = new PublicRelatedDto { UserId = CurrentUserId
};

                if(postCached.IsBookmarkedChanged(bookmarked, postAction !=
null && postAction.Bookmarked))
                {
                    statisitcCached.UpdateBookmarksCount(bookmarked);
                }

                postCached.UpdateBookmarks(newUser, bookmarked, postAction !=
null && postAction.Bookmarked);
                postCached.UpdatePreferences(newUser, preferred, postAction
!= null && postAction.Preferred);

                postCached.UserData.Add(newUser);
            }
            else
            {
                if (postCached.IsBookmarkedChanged(bookmarked,
user.Bookmarked))
                {
```

```
                    statisitcCached.UpdateBookmarksCount(bookmarked);
                }

                postCached.UpdateBookmarks(user, bookmarked,
user.Bookmarked);
                postCached.UpdatePreferences(user, preferred,
user.Preferred);
            }

            if(fromCache == null)
                await
cache.Set(Constants.ExpirationCacheKey(Constants.PostExpirationKeyPart),
postKey, Constants.LargeExpiration);

            if (fromCacheStatistics == null)
                await
cache.Set(Constants.ExpirationCacheKey(Constants.UserExpirationKeyPart),
userKey, Constants.LargeExpiration);

            await cache.Set(postKey, postCached, Constants.LargeExpiration);

            await cache.Set(userKey, statisitcCached,
Constants.LargeExpiration);

            return Ok(new { Preferences = postCached.PreferencesCount,
Bookmarks = postCached.BookmarksCount });
        }


        [HttpDelete]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> DeletePost([FromQuery] Guid postId)
        {
            var post = await executor.GetSingleBy<Post>(p => p.Id == postId);

            if(post.UserId != CurrentUserId)
            {
                return BadRequest();
            }

            //await s3Manager.DeleteFile(post.PreviewKey);

            await executor.Execute(new DeletePost(postId, CurrentUserId,
post.Private || post.Status == PostStatus.ONMODERATION));

            return Ok();
        }

        [HttpPut("moderationDelete")]
        [ServiceFilter(typeof(ModeratorFilterAttribute))]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> ModerationDeletePost([FromBody]
RejectPostDto data)
        {
            await s3Manager.DeleteFile(data.PreviewKey);

            await executor.Execute(new DeletePost(data.PostId, data.UserId,
true));

            var emailBody = Mailing.GetRejectedPostEmailBody(data.Locale,
data.PostTitle, data.RejectReason);

            await Mailing.SendEmail(configuration, data.UserName, data.Email,
emailBody);
```

```
            return Ok();
        }
    }
```

Клас AuthController

```
[Route("api/[controller]")]
    [ApiController]
    public class AuthController : FoodiesControllerBase
    {
        private readonly IAuthManager authManager;
        private readonly IConfiguration configuration;
        private readonly IQueryExecutor executor;

        public AuthController(IAuthManager authManager, IConfiguration
configuration, IQueryExecutor executor)
        {
            this.authManager = authManager;
            this.configuration = configuration;
            this.executor = executor;
        }

        [HttpPost("login")]
        [ServiceFilter(typeof(ValidationFilterAttribute))]
        public async Task<IActionResult> Authenticate([FromBody]
UserForAuthenticationDto userForAuthentication)
        {
            var user = await executor.GetBy<UserIdentity>(x =>
x.Email.Equals(userForAuthentication.Email))
                .Include(x => x.User).Where(x => x.User.Status !=
Entities.Models.Postgres.UserStatus.REMOVED).FirstOrDefaultAsync();

            if (user == null)
            {
                return NotFound();
            }

            if(user.EndBlock != null || user.MarkedForDeleteAt != null)
            {
                ModelState.AddModelError("userBlockedOrRemoved", "User has
been blocked or removed.");
                return BadRequest(ModelState);
            }

            if (!BCrypt.Net.BCrypt.Verify(userForAuthentication.Password,
user.PasswordHash))
            {
                ModelState.AddModelError("wrongEmailOrPassword",
Constants.AuthUserNotFoundMsg);
                return Unauthorized(ModelState);
            }

            if (!user.EmailConfirmed)
            {
                ModelState.AddModelError(Constants.UserNotVerifiedKey,
Constants.UserNotVerifiedMsg);
                return BadRequest(ModelState);
            }

            var accessToken = authManager.CreateAccessToken(user);
```

```
            var refreshToken = authManager.CreateRefreshToken();

            var refreshTokenExpiration = DateTime.Now.AddDays(

Convert.ToDouble(configuration.GetSection(Constants.jwtSettings)
                .GetSection(Constants.refreshTokenExpiresDays).Value));

            await executor.Execute(new UpdateUserAuthData(user.Id,
refreshToken, refreshTokenExpiration, DateTime.Now));

            return Ok(new { accessToken, refreshToken });
        }
    }
```

Клас TokenController

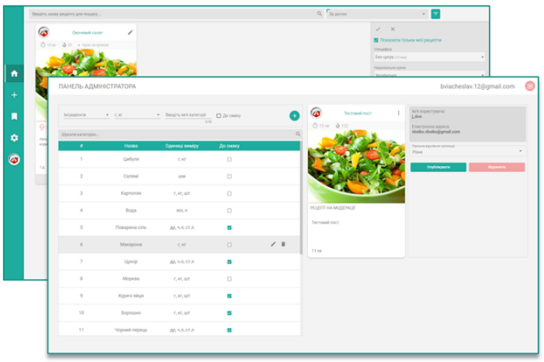
```
[Route("api/[controller]")]
    [ApiController]
    public class TokenController : FoodiesControllerBase
    {
        private readonly IAuthManager authManager;
        private readonly IQueryExecutor executor;

        public TokenController(IAuthManager authManager, IQueryExecutor
executor)
        {
            this.authManager = authManager;
            this.executor = executor;
        }

        [HttpPost("refresh")]
        public async Task<IActionResult> Refresh([FromBody] TokenApiModel
tokenApiModel)
        {
            if (tokenApiModel == null)
            {
                return BadRequest(Constants.InvalidRequestMsg);
            }

            string accessToken = tokenApiModel.AccessToken;

            string refreshToken = tokenApiModel.RefreshToken;

            var principal =
authManager.GetPrincipalFromExpiredToken(accessToken);

            var id = new Guid(principal.Claims.Where(c => c.Type ==
ClaimTypes.NameIdentifier).First().Value);

            var user = await executor.GetSingleBy<UserIdentity>(x => x.Id ==
id);

            if (user == null || user.RefreshToken != refreshToken ||
user.RefreshTokenExpiration <= DateTime.Now)
            {
                return BadRequest(Constants.InvalidRequestMsg);
            }
```

```
            var newAccessToken = authManager.CreateAccessToken(user);

            var newRefreshToken = authManager.CreateRefreshToken();

            var result = await executor.Execute(new
UpdateUserAuthData(user.Id, newRefreshToken, user.RefreshTokenExpiration,
user.LastLoggedAt), true);

            return Ok(new { AccessToken = newAccessToken, RefreshToken =
newRefreshToken });
        }

        [HttpPost("revoke"), Authorize]
        public async Task<IActionResult> Revoke()
        {
            var user = await executor.GetSingleBy<UserIdentity>(x => x.Id ==
CurrentUserId);

            if (user == null)
            {
                return BadRequest();
            }

            await executor.Execute(new UpdateUserAuthData(user.Id), true);
            return NoContent() }}}
```

ДОДАТОК Л

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ



Кафедра інженерії програмного забезпечення

Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів

Виконав:
студент IV курсу, група ІПЗ-17-1
Бойко В'ячеслав

Керівник:
д-р фіз.-мат. наук, професор
Бедратюк Л.П.

Мета та завдання проекту

Метою проекту є розробка програмної системи, яка дозволяє автоматизувати публікацію та поширення кулінарних рецептів серед користувачів, а також забезпечує швидкий та зручний їх пошук та має сучасний інтуїтивно зрозумілий користувацький інтерфейс.

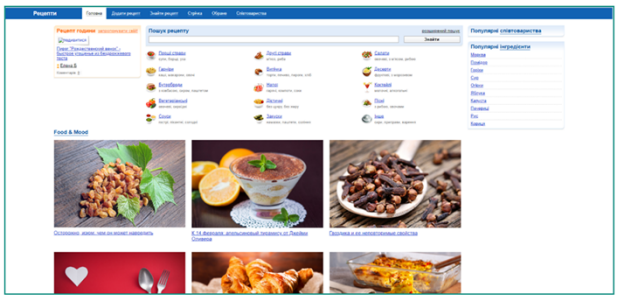
Для реалізації проекту необхідно виконати такі завдання:

1. встановити особливості та визначити специфіку предметної області соціальної мережі;
2. виконати порівняння монолітної та мікросервісної архітектури;
3. провести аналіз методів та інструментів по управлінню високонавантаженими системами та шляхи їх оптимізації;
4. проаналізувати використання реляційних, документо-орієнтованих та графових баз даних при реалізації; програмної системи та реалізувати серверну частину програмної системи;
5. встановити особливості фронт-енд фреймворків та реалізувати клієнтську частину веб-додатку;
6. провести апробацію програмної системи, реалізувати тестування програмного забезпечення.

Актуальність теми

Актуальність теми полягає у тому, що на сьогодні у мережі Інтернет є велика кількість кулінарних веб-сайтів, проте повноцінних соціальних мереж із такою тематикою практично не існує.

Подібні програмні рішення

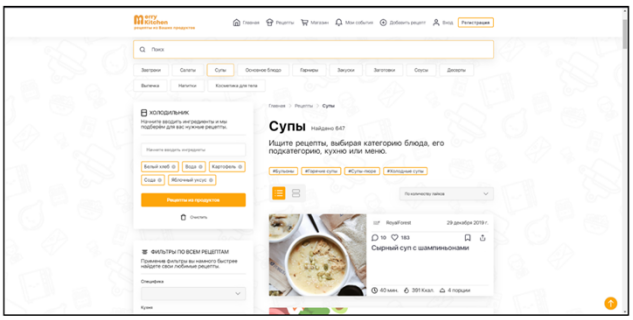


- ✓ Можливість публікації рецептів
- ✓ Авторизація через логін та пароль
- ✓ Редагування даних особистого кабінету
- × Застарілий дизайн інтерфейсу
- × Відсутність сучасної функціональності соціальних мереж

Актуальність теми: подібні програмні рішення

- ✓ Зручний користувацький інтерфейс
- × Відсутність можливості публікації своїх рецептів
- × Авторизація тільки через соцмережі
- × Відсутність редагування особистої інформації
- × Відсутність сучасної функціональності соціальних мереж

Актуальність теми: подібні програмні рішення



- ✓ Зручний користувацький інтерфейс
- ✓ Є механізм публікації рецептів, проте тільки використовуючи мобільний додаток
- ✓ Присутня як авторизація через логін та пароль, так і через популярні соцмережі
- ✓ Є можливість редагування особистих даних
- ✓ Присутня сучасна функціональність соцмереж

Порівняння монолітної та мікросервісної архітектури

Моноліт:

- ✓ Легше реалізувати
- ✓ Єдина кодова база
- ✓ Простий у розгортання
- ✓ Простий у масштабуванні
- ✓ Вдалий вибір для MVP-додатків
- × Кодова база з часом розростається
- × Складно тестувати, якщо кодова база велика
- × Складніше інтегрувати нові технології
- × Витрачається багато часу на тестування зі зростанням кодової бази

Мікросервіси:

- ✓ Незалежні модулі легше реалізовувати, особливо у команді
- ✓ Легше тестувати за рахунок тестування кожного окремого модуля
- ✓ Відмовостійкі
- ✓ Прості у підтримці (залежить від кількості мікросервісів та розміром команди розробки)
- × Складно домогтися цілісності даних між сервісами
- × Складність тестування системи у цілому за рахунок виникнення нових рівнів тестування
- × Не рекомендовано використовувати для розробки MVP-додатків

Аналіз та вибір типу бази даних

На етапі проектування було проаналізовано основні типи баз даних

Реляційні бази даних:

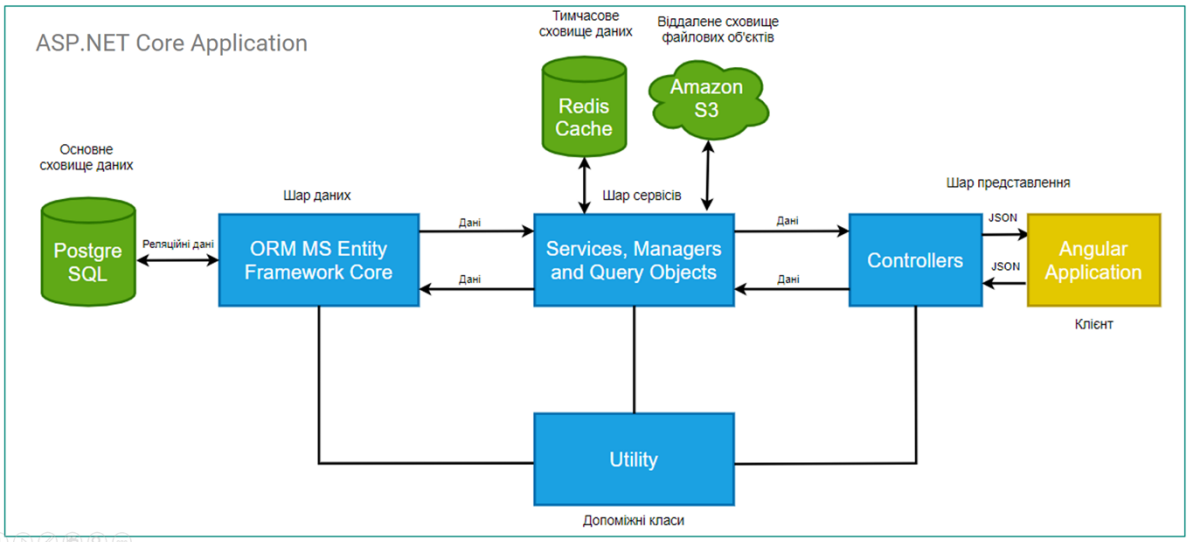
- ✓ Легко класифікувати та зберігати дані
- ✓ Присутня узгодженість даних при правильному проектування схеми
- ✓ Мають чітко визначену структуру, що запобігає невизначеності у прийнятті архітектурних рішень
- × Зі збільшенням обсягу даних зменшується продуктивність додатку
- × Велика ймовірність побудови не ефективних запитів

Бази даних «NoSQL»:

- ✓ Простота розробки
- ✓ Гнучкість схеми даних
- ✓ Велика різноманітність засобів для керування даними
- ✓ Легко масштабувати та виконувати реплікацію даних
- ✓ Деякі підтипи ефективно використовуються як прошарки кешування даних між основним сховищем та програмою
- × Втрачається стовідсоткова цілісність даних
- × Деякі підтипи не підтримують базових функцій, такі як фільтрація та сортування

Модульна архітектура

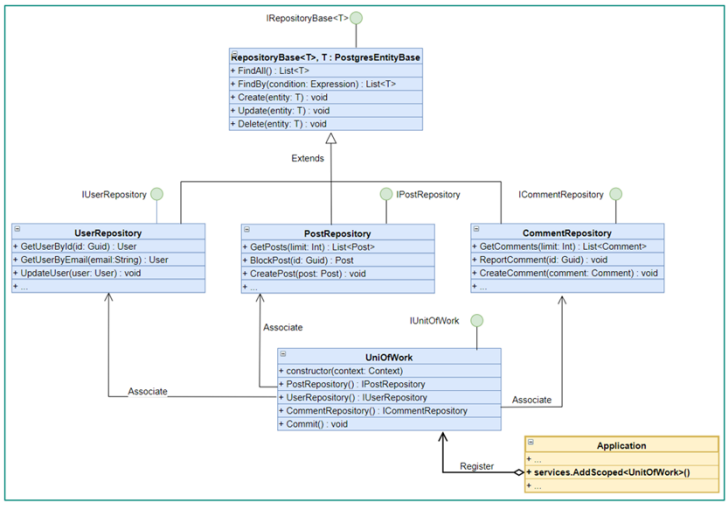
Модульна архітектура програмної системи у загальному виглядає таким чином:



Модульна архітектура: порівняння патернів проектування сервісного шару додатку

Існує два найбільш поширених патернів проектування: «Репозиторій» та «Об’єкт запиту».

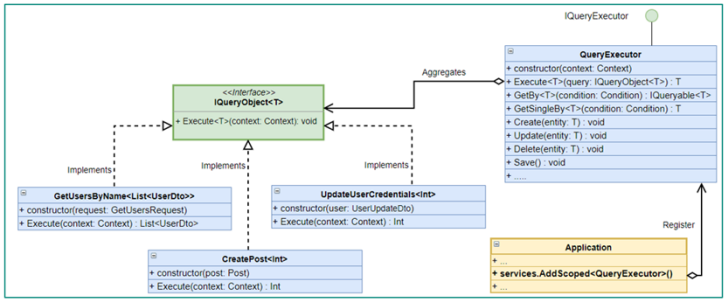
Нижче представлено шаблон «Репозиторій» та подано його переваги та недоліки.



- ✓ Інкапсулює логіку доступу до даних
- ✓ Легко тестувати
- ✓ Відповідає парадигмі DDD
- × Кількість репозиторіїв збільшується зі збільшенням моделей бізнес-логіки
- × Не можна описати усі нетривіальні запити до бази даних, використовуючи моделі бізнес логіки, що застосовуються у репозиторіях
- × CRUD методи, описані у базовому репозиторії можуть не реалізовуватися дочірніми за відсутності необхідності

Модульна архітектура: порівняння патернів проектування сервісного шару додатку

Характеристики шаблону «Об’єкт запиту»:

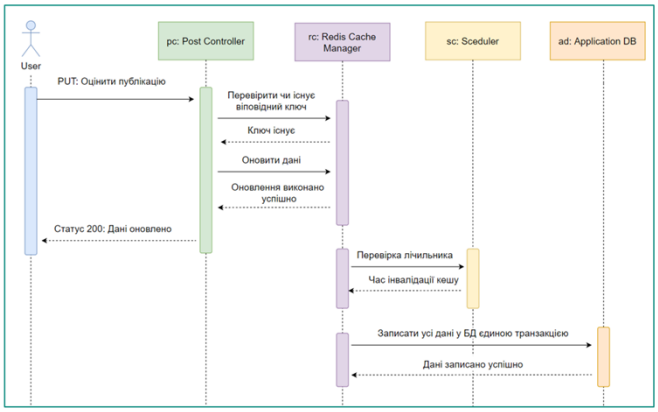


Отже, для реалізації сервісного шару було обрано патерн «Об’єкт запиту», що є заміною громіздкого репозиторію

- ✓ Інкапсулює логіку доступу до даних
- ✓ Тестувати легше, ніж репозиторії
- ✓ Кожен об’єкт запиту виконує лише специфічну функцію
- ✓ Можна виконувати будь-які нетривіальні запити
- ✓ Інтерфейсна реалізація дозволяє замінити класи-об’єкти запиту на структури, що позбавить їх від утилізації GC
- ✓ Крім нетривіальних може містити і стандартні CRUD методи для роботи з даними

Функціональна архітектура: кешування дій користувача

Якщо багато користувачів, яким сподобалась публікація іншого користувача будуть одночасно ставити оцінку або додавати у збережені, то у результаті високого навантаження на БД може погіршитись продуктивність додатку. Для уникнення даної ситуації було реалізовано стратегію кешування, що дозволить «розтягнути» навантаження у часі і записувати акумульовані дані у БД лише через певний проміжок часу.



Функціональна архітектура: кешування дій користувача

При реєстрації користувача його дані не заносяться у базу даних до тих пір, поки він не підтвердить свою електронну пошту за допомогою коду доступу, що був на неї надісланий. Дані зберігаються у кеші протягом деякого терміну, а потім автоматично видаляються.

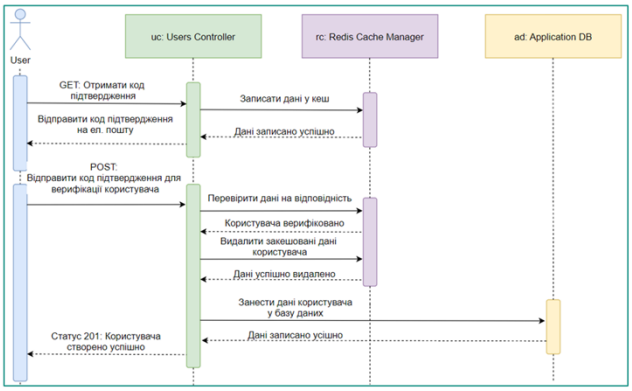
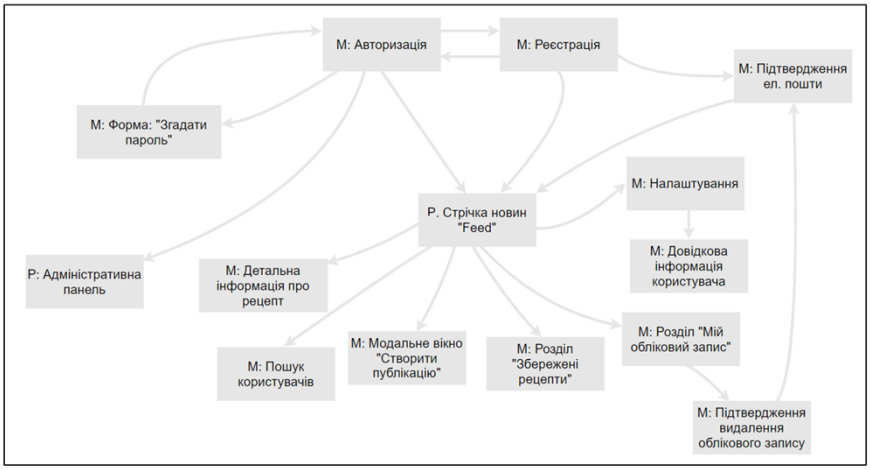


Схема екранів додатку



ВИСНОВКИ

У процесі виконання дипломного проекту було проведено комплексний аналіз у галузі соціальних мереж та програмних системи із публікації, поширення та пошуку кулінарних рецептів. Визначено основні переваги та недоліки існуючого програмного забезпечення та встановлено, що їх кількість невелика, тому тема є актуальною. Також було виконано порівняння монолітного та мікросервісного стилю серверної архітектури програмної системи та проаналізовано типи сховища даних, визначено їх переваги та недоліки і вибрано реляційну систему керування базами даних. У ході аналізу та проектування програмного забезпечення було визначено потенційні ділянки падіння продуктивності програми та за допомогою інструментів по управлінню високонавантаженими системами їх усунено. Проаналізовано характеристики сучасних фронтенд-фреймворків та реалізовано клієнтську частину програмної системи із використанням платформи Angular. Проведено тестування програмного забезпечення, а результати практичної апробації системи підтверджують її працездатність.

У результаті роботи було реалізовано програмне забезпечення, що дозволяє автоматизувати пошук та обмін кулінарних рецептів серед користувачів і має сучасний та зрозумілий користувацький інтерфейс.

ДЯКУЮ ЗА УВАГУ

Завідувачу кафедри
Бедратюку Леоніду Петровичу

_____

здобувача вищої освіти
ФПКТС, 4 курсу, групи ІПЗ-17-1,
Бойка В'ячеслава Олександровича

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

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

01.06.2021
дата

підпис

Tue Jun 01 07:24:22 EEST 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 12.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 17%**

	Документ		Сумарний збіг по Базі Даних	
ID: 91748 Назва: Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів Додано в БД: 2021-06-01 Автора: В.О. Бойко Керівники: Л.П. Бедратюк Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	174478	1674	23888 (14%)	237 (14%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
90216	Назва: Звіт з переддипломної практики Бойко В.О. Додано в БД: 2021-05-11 Автора: Бойко В.О. Керівники: Бедратюк Л.П. Консультанти: Опоненти:	20108 (12.0%)	188 (11.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1008134724

Дата перевірки:
02.06.2021 10:21:34 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
02.06.2021 10:27:17 EEST

ID користувача:
100005589

Назва документа: **Записка Бойко В.О**

Кількість сторінок: **177** Кількість слів: **29074** Кількість символів: **244414** Розмір файлу: **9.56 MB** ID файлу: **1008215685**

4.52%
Схожість

Найбільша схожість: **1.74%** з джерелом з Бібліотеки (ID файлу: **1008215667**)

3.06% Джерела з Інтернету	379	Сторінка 179
2.02% Джерела з Бібліотеки	65	Сторінка 182

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	1

РЕЦЕНЗІЯ НА ДИПЛОМНИЙ ПРОЕКТ
освітнього ступеня «Бакалавр»

Дипломник Бойко В'ячеслав Олександрович

Тема Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломного проекту:

Кількість листів креслень _____; кількість сторінок записки 86

1. Короткий зміст пояснювальної записки та прийнятих рішень У дипломному проекті проведено аналіз, встановлено особливості та визначено специфіку тематичних соціальних мереж. Виконано порівняння монолітного та мікросервісного підходу до проектування серверної архітектури, визначено особливості реляційних та нереляційних баз даних, проаналізовано інструменти по управлінню високонавантаженими системами, визначено шляхи їх оптимізації та реалізовано серверну частину програмної системи. Встановлено особливості фронтенд-фреймворків та реалізовано клієнтську частину веб-додатку. Проведено тестування та практичну апробацію програмного забезпечення

2. Висновок про відповідність проекту поставленому завданню Дипломний проект освітнього ступеня «бакалавр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині

3. Характеристика виконання кожного розділу проекту, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі обґрунтовується актуальність теми роботи, аналізується досліджувана проблема та обґрунтовується застосовуваний підхід до її вирішення, формуються цілі та завдання. У першому розділі глибоко проаналізовано предметну область, виконано порівняння наявного програмного забезпечення та встановлено значущість розроблюваної програмної системи, детально описані варіанти використання системи. У другому розділі якісно подано характеристику сучасних підходів до проектування та реалізації серверної частини додатку. Якісно спроектована концептуальна модель, глибоко описано деталі проектування модульної та функціональної архітектури, а також користувацького інтерфейсу додатку. У третьому розділі наведені ключові етапи реалізації програмної системи, що у повній мірі відповідають розробленим архітектурним рішенням. У четвертому розділі проведено аналіз основних методів тестування та наведено опис системного тестування програмного забезпечення.

4. Позитивні сторони проекту Реалізований програмний продукт має інтуїтивно зрозумілий користувацький інтерфейс та містить як базовий функціонал сучасних соціальних мереж, так і зручну систему збереження та упорядкування збережених рецептів. Окрім того базова реалізація системи передбачає забезпечення стабільної

продуктивності за рахунок використання сучасних технік кешування та оптимізації запитів до сховища даних.

5.Негативні сторони проекту Розроблюваний додаток є MVP-продуктом, а отже не можна виконати у повній мірі навантажувальне тестування, що означає, що забезпечення стабільної продуктивності при високих навантаженнях залишається лише на теоретичному рівні. Негативна сторона не зменшує позитивне враження від проекту

6. Оцінка графічного оформлення та пояснювальної записки проекту Графічне оформлення виконане відповідно до теми дипломного проекту із дотриманням вимог стандартів. У загальному графічне оформлення виконане на достатньому рівні. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про дипломний проект в цілому Дипломний проект заслуговує позитивної оцінки. Весь матеріал добре структурований, поданий чітко та послідовно. Розділи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломного проекту. Графічний матеріал наочно відображає доцільність та ефективність рішень, що були прийняті за основу для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломного проекту Визначивши позитивні та негативні сторони представленого дипломного проекту, можна зробити висновок, що дипломний проект заслуговує оцінки «відмінно» (4,75/A).

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)

Кисіль Тетяна Миколаївна, кандидат фізико-математичних наук, доцент кафедри комп'ютерної інженерії та системного програмування

«25» травня 2021 р. _____
(підпис)

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

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Програмна система для автоматизації публікації, поширення та пошуку кулінарних рецептів»

Автор: Бойко В'ячеслав Олександрович

Спеціальність: 121 – Інженерія програмного забезпечення

Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»

Науковий керівник: Бедратюк Леонід Петрович, доктор фіз.-мат. наук, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломного проекту системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень в бланках (титулка, бланк завдання, в структурі підрозділів ВСТУПУ) та в назвах публікацій джерел;
2) В якості запозичень системою було зафіксовано послідовність вихідного коду, які є спільними для великої кількості задач і не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;
3) усі запозичення фрагментарні, або мають належним чином оформленні посилання.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності / схожості, складає 4,52% і адресується до 444 першоджерел, що з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь дипломного проекту.

Керівник _____ Л.П. Бедратюк

Гарант ОП _____ Л.П. Бедратюк

Завідувач кафедри _____ Л.П. Бедратюк