Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Інформаційна система онлайн-сервісу пошуку знайомств»

08-54.БКР.015.00.000 ПЗ

Виконав: студент 4 курсу, групи 1КІ-21б
спеціальності
123 Комп'ютерна інженерія,
(шифр і назва напряму підготовки, спеціальності)
освітня програма
«Комп'ютерна інженерія»
Сухина Є. В.
(прізвище та ініціали)

Керівник: к.пед.н., доц. каф. ОТ
Добровольська Н. В.
(прізвище та ініціали)

Рецензент: д.т.н., проф. каф. ПЗ
Ліщинська Л. Б.
(прізвище та ініціали)

ДОПУЩЕНО
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д
"11" 06 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Галузь знань – 12 – Інформаційні технології

Спеціальність – 123 – Комп'ютерна інженерія

Освітньо-професійна програма – Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д
"___" _____2025 р.

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Сухині Євгену Вікторовичу

1 Тема роботи: «Інформаційна система онлайн-сервісу пошуку знайомств», керівник роботи к.пед.н., доц. каф. ОТ Добровольська Н. В. затверджені наказом ВНТУ від «20» березня 2025 року №97.

2 Термін подання студентом проєкту: 13.05.2025

3 Вихідні дані до роботи: функціональні вимоги до сервісу, структура бази даних PostgreSQL, монолітна архітектура на платформі ASP.NET, підтримка автентифікації через cookies, використання SignalR для обміну повідомленнями в реальному часі. Застосовне програмне забезпечення: .NET SDK, PostgreSQL, Docker, Postman, JetBrains Rider.

4 Зміст розрахунково-пояснювальної записки: вступ, аналіз предметної області, вибір та обґрунтування технологій розробки, програмна реалізація застосунка

5 Перелік графічного матеріалу: ER схема бази даних, uml діаграма класів системи.

6 Консультанти розділів роботи наведені у таблиці 1.

Таблиця 1— Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Спеціальна частина	к.пед.н., доц. каф. ОТ Добровольська Н. В.	21.03.25	13.05.25
Нормоконтроль	ас. каф. ОТ Швець С.І.	14.05.25	30.05.25

7 Дата видачі завдання 21.03.2025 р.

8 Календарний план наведений у таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання		Пр
		початок	закінчення	
1	Постановка задачі. Визначення мети, завдань, актуальності та структури системи цифрової соціалізації.	21.03.25	27.03.25	ви
2	Аналіз існуючих онлайн-сервісів знайомств. Обґрунтування вибору технологій ASP.NET та Nuxt.	28.03.25	04.04.25	ви
3	Проєктування бази даних, API та структури взаємодії клієнта з сервером.	05.04.25	12.04.25	ви
3	Реалізація серверної частини: обробка користувацьких даних та автентифікація	13.04.25	27.04.25	ви
5	Розробка клієнтського інтерфейсу	28.04.25	09.05.25	ви
6	Підключення модуля реального часу	10.05.25	11.05.25	ви
7	Оформлення пояснювальної записки та ілюстративного матеріалу	11.05.25	13.05.25	ви
8	Попередній захист БКР	13.05.25	13.05.25	ви

Студент _____ (підпис) Сухина Є. В.

Керівник роботи _____ (підпис) Добровольська Н. В.

АНОТАЦІЯ

УДК 004.382.4:621.382

Сухина Є. В. Інформаційна система онлайн-сервісу пошуку знайомств. Бакалаврська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна інженерія, освітня програма Комп'ютерна інженерія. Вінниця: ВНТУ, 2025. 84 с.

На укр. мові. Бібліогр.: 21 назв; рис.: 34; табл. 7.

В даній бакалаврській роботі розглянуто створення інформаційної системи онлайн-сервісу для пошуку знайомств. На основі аналізу актуальності теми та сучасних технологій було досліджено існуючі рішення у сфері онлайн-знайомств, їх функціональні можливості, переваги та недоліки. З урахуванням отриманих результатів було спроєктовано та реалізовано веб застосунок, який включає модулі реєстрації та авторизації користувачів, пошуку та фільтрації анкет, обміну повідомленнями у режимі реального часу, а також управління профілями. Особлива увага приділена створенню інтуїтивного та адаптивного інтерфейсу з використанням Nuxt.js на клієнтській стороні та надійного серверного рішення на базі ASP.NET, що забезпечують високу продуктивність, безпеку та масштабованість системи.

Ключові слова: онлайн-сервіс, пошук знайомств, веб застосунок, Nuxt.js, ASP.NET, авторизація, пошук анкет, чат, база даних.

ABSTRACT

UDC 004.382.4:621.382

Sukhyna Y. V. Information System of an Online Dating Service. Bachelor's Qualification Thesis in specialty 123 — Computer Engineering, educational program Computer Engineering. Vinnytsia: VNTU, 2025. 84 p.

In Ukrainian. Bibliography: 21 titles; illustrations: 34; tables: 7.

This bachelor's thesis presents the development of an information system for an online dating service. Based on an analysis of the relevance of the topic and modern technologies, existing solutions in the field of online dating were examined, along with their functionalities, advantages, and disadvantages. Taking these findings into account, a web application was designed and implemented. It includes modules for user registration and authentication, profile search and filtering, real-time messaging, and profile management. Particular attention was paid to creating an intuitive and responsive user interface using Nuxt.js on the client side and a reliable server-side solution based on ASP.NET, ensuring high performance, security, and scalability of the system.

Keywords: online service, dating search, web application, Nuxt.js, ASP.NET, authentication, profile search, chat, database

ЗМІСТ

ВСТУП

У сучасному цифровому світі онлайн-сервіси пошуку знайомств займають важливу роль серед інструментів соціальної взаємодії, надаючи користувачам ефективні механізми для встановлення нових знайомств, дружніх або романтичних відносин, незалежно від їхнього місцезнаходження. Збільшення доступності інтернету та розвиток мобільних технологій створили передумови для поширення таких сервісів, які стали невід'ємною частиною повсякденного життя мільйонів людей по всьому світу. Попит на якісні платформи зростає через необхідність забезпечення зручності, безпеки та персоналізації користувацького досвіду. Водночас, стрімке збільшення обсягів даних та зростаюча кількість активних користувачів висувають підвищені вимоги до продуктивності, масштабованості та надійності інформаційних систем, що реалізують функції пошуку знайомств [1].

Актуальність обраної теми зумовлена стрімким розвитком цифрових технологій та зростаючою роллю онлайн-комунікацій у соціальному житті сучасної людини. Інформаційні системи онлайн-сервісів знайомств забезпечують ефективний, зручний і безпечний спосіб встановлення нових соціальних зв'язків.

Метою дослідження є створення сучасної інформаційної системи онлайн-сервісу пошуку знайомств із використанням Nuxt.js для клієнтської частини та ASP.NET для серверної логіки.

Задачі роботи включають розробку архітектури системи, реалізацію ключових функціональних модулів, таких як реєстрація, авторизація, пошук і фільтрація анкет, обмін повідомленнями, а також інтеграцію механізмів безпеки та захисту персональних даних користувачів. Використання Nuxt.js забезпечує створення високопродуктивного, адаптивного та SEO-оптимізованого фронтенду, а ASP.NET дозволяє реалізувати надійний, масштабований бекенд із широкими можливостями для обробки бізнес-логіки та управління даними.

Об'єктом дослідження є інформаційні системи онлайн-сервісів пошуку знайомств, що представляють собою складні програмні комплекси, які інтегрують функції збору, зберігання, обробки та передачі інформації про користувачів з

метою створення умов для ефективної соціальної взаємодії. Такі системи включають у себе компоненти управління профілями, пошуку та фільтрації, організації комунікації між користувачами, а також заходів безпеки і контролю доступу.

Предметом дослідження є процеси проектування, розробки та впровадження клієнтської і серверної частин інформаційної системи онлайн-сервісу пошуку знайомств, що базуються на Nuxt.js та ASP.NET, із акцентом на інтеграцію цих технологій, оптимізацію продуктивності, забезпечення надійного захисту даних та створення зручного та інтуїтивного інтерфейсу для користувачів. Реаліючи архітектурні підходи, механізми аутентифікації та авторизації, а також особливості організації обміну повідомленнями і пошуку анкет у межах єдиної платформи.

Робота спрямована на формування комплексного технічного рішення, що задовольняє сучасні вимоги інформаційної безпеки, ефективності та користувацької взаємодії.

Апробація результатів бакалаврського кваліфікаційної роботи була проведена на конференції “Молодь в науці: дослідження, проблеми, перспективи” (МН-2025) [2]:

Сухина Є. В., Добровольська Н. В. Інформаційна система як засіб цифрової соціалізації // Матеріали всеукраїнської науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців — Режим доступу : https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/viewFile/25371/20954

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Вимоги до веб застосунку для пошуку знайомств

Веб застосунок для пошуку знайомств має бути всебічним рішенням, яке інтегрує великий набір функцій, високі стандарти безпеки, зручності та якості, адаптоване до сучасних умов цифрової взаємодії. Користувачі прагнуть не лише базового набору інструментів, а й інтуїтивно зрозумілого інтерфейсу, миттєвої реакції системи, надійного захисту персональних даних та гнучкості для різних сценаріїв використання. Тому вимоги до застосунку охоплюють як технічні аспекти, так і принципи UX-дизайну, які забезпечують позитивний досвід.

Одним із фундаментальних елементів є функціонал реєстрації та авторизації, який має бути простим, доступним і водночас безпечним. Застосунок повинен підтримувати кілька способів реєстрації, зокрема через email, номер телефону або соціальні мережі, із обов’язковим підтвердженням для захисту від фейкових акаунтів. Паролі мають зберігатися з використанням сучасних алгоритмів хешування, таких як bcrypt, а авторизація передбачає захист сесійних даних через токени, протидію підробці запитів та захист від викрадення сесії, щоб гарантувати безпеку облікових записів [3].

Створення та редагування персонального профілю є основою для пошуку партнерів і має бути зручним і функціональним. Профіль повинен включати ключові дані: фото, вік, інтереси, місце проживання, наміри (наприклад, дружба чи стосунки) та інші характеристики, які використовуються в алгоритмах підбору. Інтерфейс має полегшувати внесення та оновлення інформації, передбачати автоматичну валідацію даних і надавати миттєвий зворотний зв’язок, щоб користувач одразу бачив результат змін.

Пошук анкет і фільтрація є центральною функцією, яка має забезпечувати гнучкість і релевантність результатів. Користувачі повинні мати можливість налаштувати параметри пошуку — вік, географічне розташування, інтереси чи цілі знайомства — і отримувати оновлені списки профілів у реальному часі без необхідності перезавантаження сторінки. Використання технологій кешування та

оптимізації запитів забезпечує високу продуктивність навіть при значному навантаженні, дозволяючи платформі ефективно працювати з великою кількістю користувачів.

Обмін повідомленнями є життєво важливим для підтримки комунікації між учасниками. Система має гарантувати миттєву доставку текстів із відображенням статусу прочитання та доступ до історії листування. Реалізація через технології реального часу, такі як WebSocket або SignalR, забезпечує безперервність і швидкість обміну. Інтерфейс чату має бути адаптивним, зручним для набору тексту та перегляду повідомлень, підтримуючи базові функції, як-от додавання емодзі чи зображень.

Нефункціональні вимоги включають ключові аспекти безпеки, такі як шифрування персональних даних, захист від зовнішніх атак і стійка система аутентифікації та авторизації. Масштабованість є необхідною, щоб платформа могла впоратися зі зростанням кількості користувачів і обсягу даних без втрати якості. Крім того, кросплатформенність і адаптивний дизайн є обов’язковими, забезпечуючи комфортне використання на комп’ютерах, смартфонах або планшетах, з адаптацією до різних розмірів екранів.

1.2 Аналіз потреб користувачів та вимог до функціоналу

Аналіз потреб користувачів і вимог до функціоналу онлайн-сервісу для пошуку знайомств є важливим кроком у розробці платформи, яка буде відповідати сучасним очікуванням і вимогам цільової аудиторії. Основна мета такого сервісу — створити комфортне, безпечне й ефективне середовище для пошуку партнерів, що відповідають індивідуальним запитам користувачів. Цільова аудиторія охоплює молодь у віці від 18 до 35 років, яка активно користується цифровими технологіями, а також старших користувачів, які шукають різні форми спілкування — від серйозних стосунків до легких знайомств або дружби. Потреби цієї аудиторії формуються під впливом сучасного ритму життя, обмеженого часу для спілкування та прагнення ефективно використовувати цифрові платформи для встановлення якісних соціальних зв’язків.

Однією з ключових вимог користувачів є простота і швидкість реєстрації. Вони очікують, що процес створення акаунту буде максимально зручним і не вимагатиме тривалого заповнення складних форм. Реєстрація через email або популярні соціальні мережі є стандартом, що дозволяє швидко почати користування сервісом. Важливим є також інтуїтивний та зрозумілий інтерфейс, який дає змогу легко створити і налаштувати особистий профіль — додати фотографії, вказати вік, інтереси, життєві цілі та інші характеристики, що допомагають сформувати більш точний портрет користувача [4].

Ще одним важливим аспектом є розширені можливості пошуку і фільтрації партнерів. Користувачі хочуть мати інструменти, що дозволяють відбирати потенційних співрозмовників за різними критеріями — віком, статтю, місцем проживання, хобі, намірами. Важливою складовою будь-якого сервісу знайомств є безпека та конфіденційність. Користувачі висувають високі вимоги щодо захисту особистих даних та приватності спілкування. Вони очікують, що їх інформація буде надійно зашифрована, а платформа надасть можливість контролювати, хто може переглядати їхні профілі. Функції блокування небажаних контактів і модерація профілів для запобігання появі фейкових акаунтів є обов’язковими. Додатково користувачі цінують можливість використовувати механізми підвищення приватності, наприклад, самознищення повідомлень після певного часу, що захищає від зловживань і покращує довіру до сервісу. Особливі функції, які дозволяють жінкам першими починати спілкування або інші заходи захисту від нав’язливості, також підвищують рівень комфорту.

Для ефективності пошуку і підвищення якості збігів застосовуються алгоритми, які аналізують інтереси, уподобання та поведінку користувачів. Вони пропонують найбільш релевантних партнерів, що значно підвищує ймовірність успішного знайомства.

Також враховується зростаюча мобільність користувачів: платформа має бути адаптивною і зручною для роботи на смартфонах і планшетах із різними операційними системами. Швидкість завантаження сторінок, миттєвий відгук інтерфейсу, простота навігації та підтримка жестів — усе це підвищує зручність і

привабливість сервісу. Динамічні елементи, такі як обмеження часу на відповідь у чаті, додають активності, хоча й повинні бути гнучкими, щоб не створювати додаткового дискомфорту для користувачів із різними стилями спілкування.

Враховуючи описані потреби, сформовано основні вимоги до функціоналу сервісу. Він повинен забезпечувати просту і швидку реєстрацію, гнучке налаштування профілю, багатокритеріальний пошук і фільтрацію, застосування інтелектуальних алгоритмів для рекомендацій, безпечний чат, контроль конфіденційності і модерацію акаунтів.

1.3 Огляд та аналіз аналогів

Огляд аналогів веб застосунків для пошуку знайомств, які орієнтовані на серйозні стосунки, дружбу чи нетворкінг, а не на сексуальний підтекст, дозволяє оцінити поточний ринок, визначити сильні та слабкі сторони популярних платформ і знайти напрямки для створення унікального рішення.

Tinder є одним із лідерів із понад 50 мільйонами активних користувачів у світі, із приблизно 1,5 мільйона в Україні (рис. 1.1). Доступний на iOS, Android і веб, сервіс працює за принципом свайпів, де користувачі обирають профілі на основі фото, короткого опису та геолокації [5].

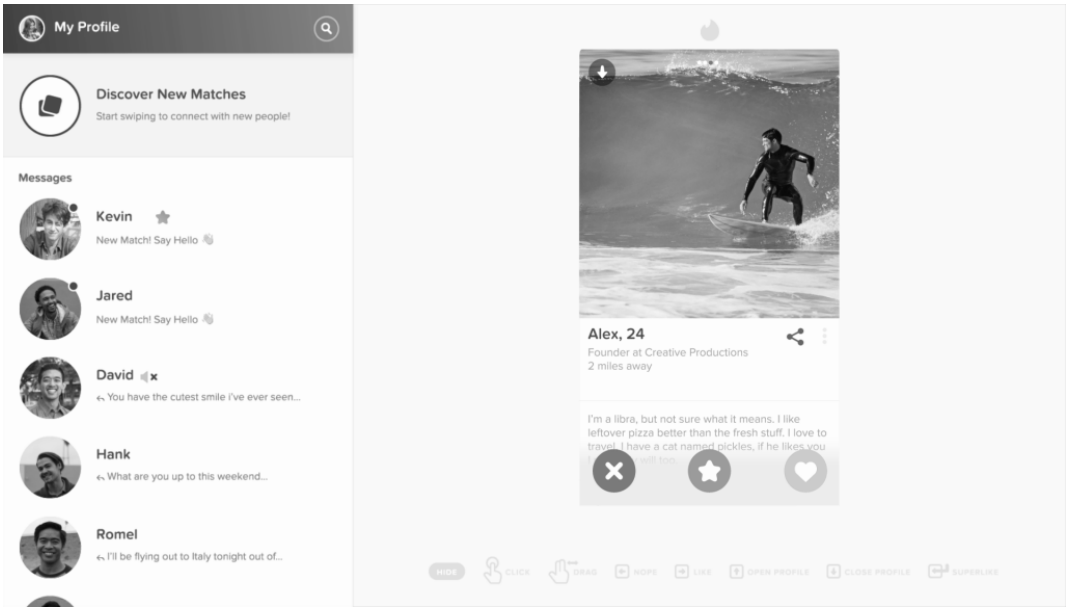

Рисунок 1.1 — Інтерфейс додатку Tinder

Базовий функціонал безкоштовний має обмеження свайпів, а для доспуту до всіх функцій доведеться придбати один із платних тарифів.

До переваг можна віднести: обширну аудиторію, від 18 до 30 років, швидкий і простий механізм підбору, а також підтримку понад чим 35 мов.

Недоліки є: значна кількість фейків і ботів, сильний акцент на неформальних знайомствах, а також часткова локалізація українською мовою, яка все ще потребує доопрацювання для повноцінного досвіду.

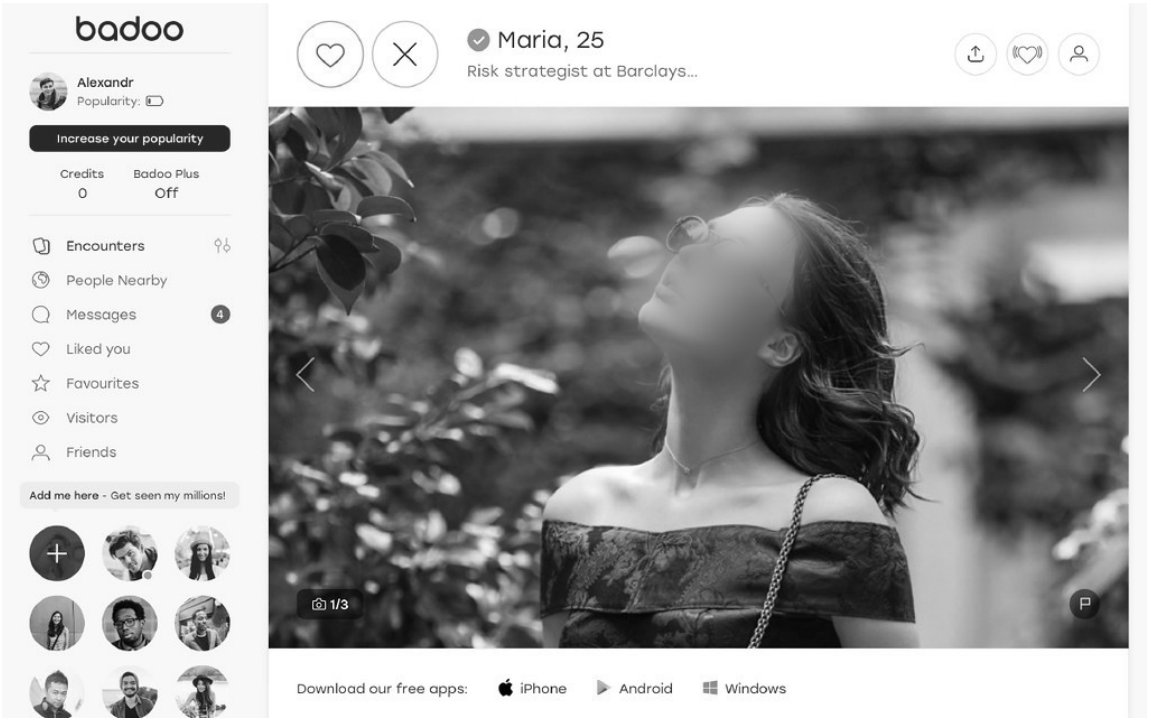

Рисунок 1.2 — Інтерфейс застосунку Badoo

Bumble має 16 мільйонів користувачів, із 450 тисячами в Україні, і вирізняється тим, що після взаємного лайка перше повідомлення можуть надсилати лише жінки [6]. Доступний на iOS, Android і веб, із безкоштовною версією та платними планами Bumble Boost і Premium, які додають "Rematch" і "Spotlight" і розширені фільтри. Має режими BFF для дружби та Bizz для нетворкінгу. Інтерфейс додатку зображено на рисунку 1.3.

Переваги: акцент на безпеці та контролі, адаптивний дизайн із підтримкою темного режиму, підтримка різних типів зв’язків, регулярні оновлення з новими функціями.

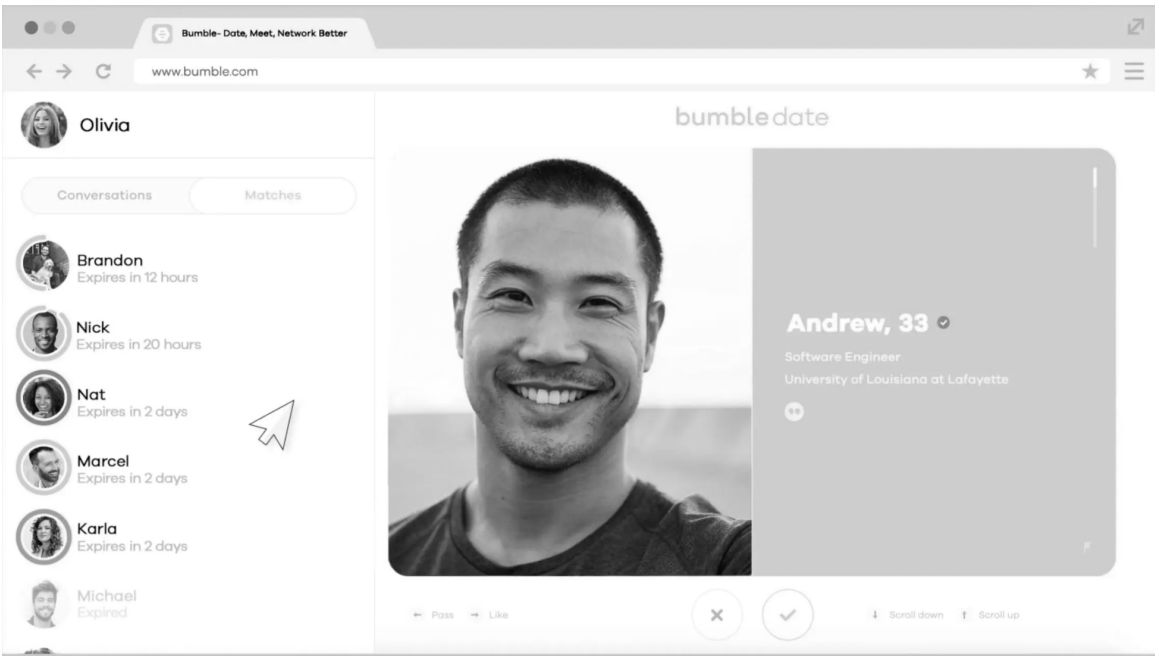

Рисунок 1.3 — Інтерфейс застосунку Bumble

OkCupid — міжнародна платформа для знайомств із 10 мільйонами активних користувачів, із приблизно 300 тисячами в Україні, яка зосереджується на глибокому підборі партнерів на основі тестів сумісності (рис. 1.4).

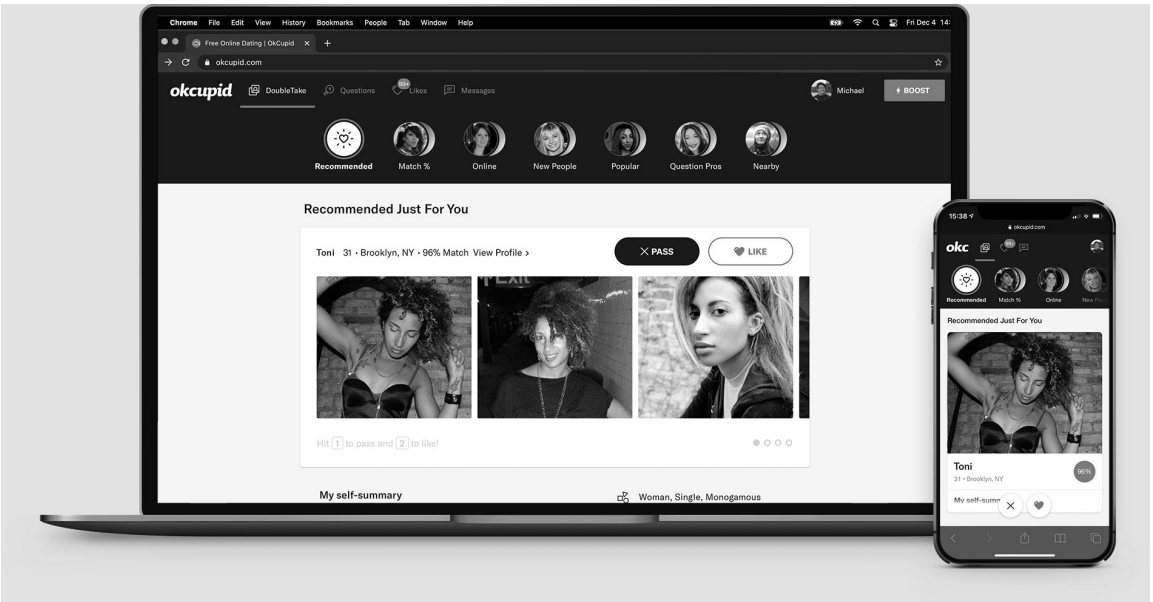

Рисунок 1.4 — Інтерфейс застосунку OkCupid

Доступний на iOS, Android і веб, із безкоштовною версією, що дозволяє переглядати профілі та надсилати обмежену кількість повідомлень, і платними

планами Basic та Premium, які розблоковують необмежене спілкування, розширені фільтри і детальну статистику сумісності.

Особливості: детальні анкети для оцінки сумісності, підтримка дружніх і романтичних зв’язків, алгоритми, які враховують відповіді на тести.

Переваги: акцент на глибокому підборі, підтримка понад 20 мов, включаючи часткову локалізацію українською [7].

Недоліки: складний інтерфейс для новачків, менша популярність в Україні порівняно з Tinder чи Mamba, обмежений безкоштовний функціонал.

Особливості платформ відображають їхню адаптацію до різних потреб: одні орієнтовані на швидкі контакти з широкою аудиторією, інші додають безпеку через ініціативу жінок і режими для дружби, деякі фокусуються на локалізації для України, а ще інші пропонують глибокий підбір через тести сумісності. Технологічні інновації, як-от геолокація, алгоритми машинного навчання та інтеграція з соцмережами, підвищують зручність, але культурна адаптація залишається слабким місцем для глобальних сервісів, на відміну від локальних платформ, які краще враховують регіональні особливості.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ІНСТРУМЕНТІВ ТА ІНСТРУМЕНТІВ ДЛЯ РОЗРОБКИ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Технологія контейнеризації

Контейнеризація — це сучасна технологія, яка забезпечує ізоляцію програмного забезпечення разом із усім необхідним для його роботи: бібліотеками, залежностями, конфігураціями. Завдяки цьому розробник може бути впевнений, що застосунок функціонуватиме однаково на будь-якому сервері чи комп’ютері, незалежно від операційної системи або налаштувань середовища.

Контейнери мають спільне ядро операційної системи з хостом, що робить їх значно легшими та швидшими у запуску, ніж традиційні віртуальні машини. Ця легкість дозволяє легко масштабувати сервіси, запускати їх паралельно в різних середовищах, а також швидко тестувати нові версії без ризику порушити стабільність системи загалом. Усе це робить контейнеризацію невід’ємною частиною сучасного підходу до розробки програмного забезпечення, особливо у мікросервісній архітектурі. Схему роботи контейнерів зображено на рисунку 2.1.

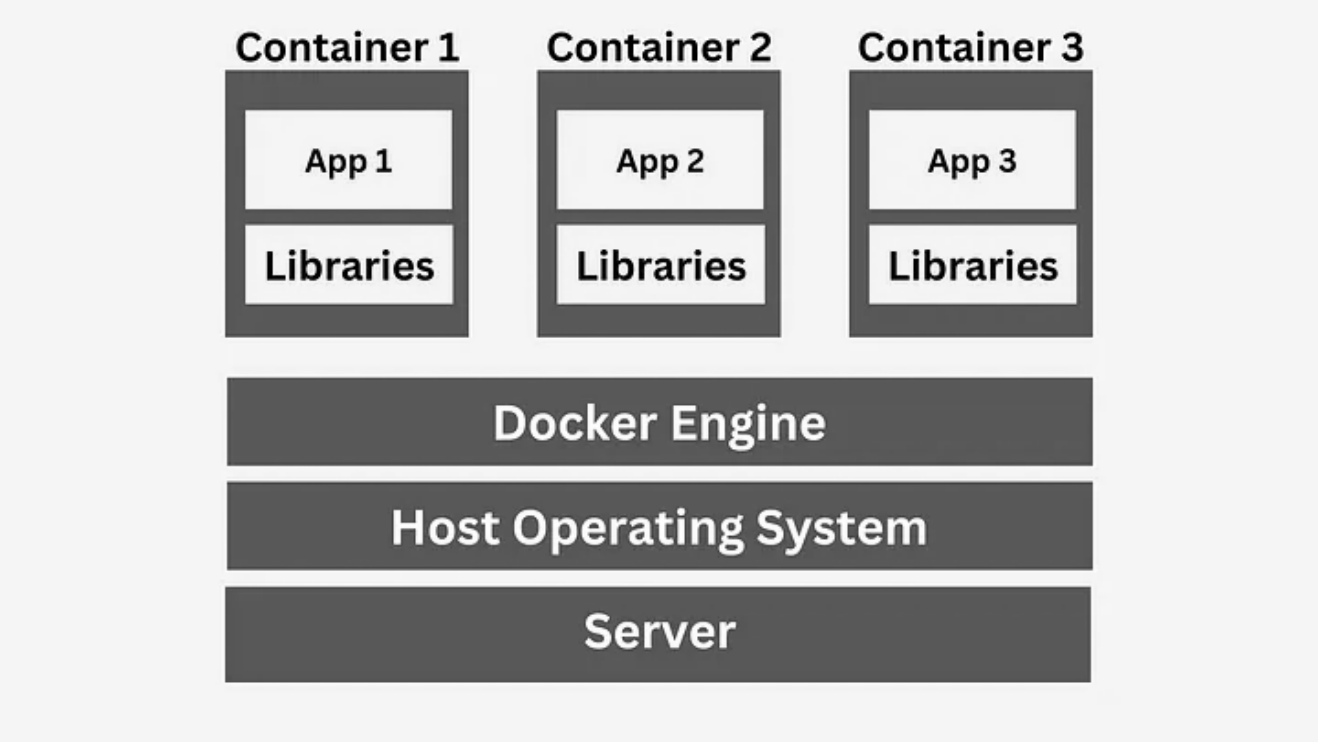

Рисунок 2.1 — Схема роботи контейнерів

Контейнери інтегруються з інструментами автоматизації розгортання, такими як CI/CD-системи, а також з хмарними платформами, що дозволяє організовувати масштабовану і гнучку інфраструктуру. Завдяки контейнерам команди розробників, тестувальників і DevOps-інженерів працюють у єдиному середовищі, без ризику втрати сумісності між етапами розробки та розгортання.

Контейнеризація знайшла своє найширше втілення завдяки інструменту Docker, який став фактично стандартом у цій сфері. Docker забезпечує простий спосіб створення, поширення та запуску контейнерів. Його популярність пояснюється тим, що він дозволяє описати програму та все її середовище у вигляді одного конфігураційного файлу — Dockerfile, що гарантує повну відтворюваність результату. Цей файл визначає, які саме залежності, змінні середовища та інструкції потрібні для того, щоб система працювала так, як задумано.

Використовуючи Docker, розробник може локально запустити ту саму версію програми, яка пізніше буде запущена на сервері, в хмарі або на будь-якому іншому середовищі, що підтримує Docker Engine (рис. 2.2). Це усуває типові проблеми, пов’язані з несумісністю середовищ [8,9].

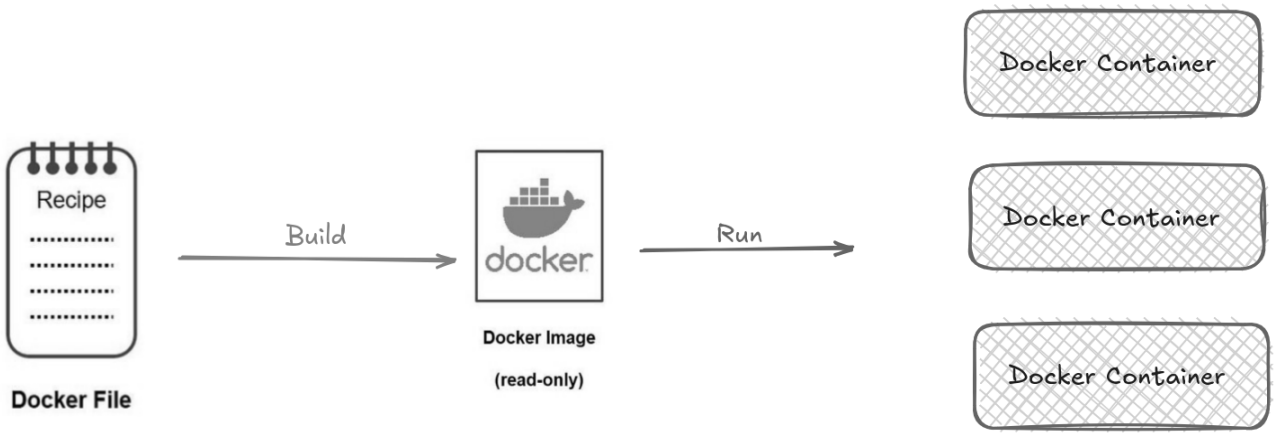

Рисунок 2.2 — Схема роботи Docker

Окрім локального запуску, Docker підтримує механізми зберігання та поширення образів — готових «знімків» контейнерів. Ці образи можуть зберігатися на спеціальних сервісах, таких як Docker Hub або GitHub Container Registry, звідки їх легко завантажувати в автоматизованих CI/CD-pipeline або

безпосередньо на сервери для розгортання.

2.2 Вибір та огляд сховищ для зберігання даних

Бази даних — є спеціалізованими системами для організації, зберігання та обробки інформації. Вони забезпечують структурований доступ до даних, дозволяючи користувачам виконувати операції створення, читання, оновлення та видалення (CRUD). Основна мета баз даних — надання надійного механізму для управління інформацією в умовах одночасного доступу,великих обсягів даних і потреби в захисті від несанкціонованих дій [10].

Архітектурно база даних складається з трьох основних компонентів: даних, системи управління базами даних (СУБД) та апаратного забезпечення. Дані представлені у вигляді структурованих одиниць, таких як таблиці, документи чи графи, залежно від типу бази. СУБД є програмним забезпеченням, яке керує доступом до даних, виконує запити, забезпечує транзакційну обробку та підтримує механізми безпеки, такі як аутентифікація й шифрування. Апаратне забезпечення включає сервери, системи зберігання (диски SSD або HDD) і мережеві компоненти, які забезпечують фізичне виконання операцій.

Бази даних вирішують низку проблем, серед який:

— цілісність даних, забезпечуючи коректність даних при одночасному доступі кількох користувачів;

— масштабованість, підтримуючи зростання обсягів даних і кількість запитів без втрати продуктивності;

— безпека, захищаючи дані від стороннього доступу чи пошкодження;

— доступність, що забезпечує швидке виконання запитів навіть у розподілених системах.

Наприклад, у системі електронної комерції база даних забезпечує збереження інформації про товари, користувачів і замовлення, дозволяючи одночасно обробляти запити клієнтів і оновлювати складські дані. У наукових дослідженнях бази даних використовуються для зберігання великих наборів експериментальних даних, забезпечуючи швидкий доступ для аналізу.

Самі ж бази даних бувають різних типів, і в основному поділяються на реляційні (SQL) і нереляційні (NoSQL), кожна з яких має унікальні архітектурні принципи, сценарії використання, переваги та обмеження.

Реляційні бази даних ґрунтуються на реляційній моделі, запропонованій Едгаром Коддом у 1970-х роках (рис 2.3.). Дані в них організовані у вигляді таблиць, де кожен рядок представляє запис, а кожен стовпець — атрибут. Таблиці пов’язані між собою за допомогою первинних і зовнішніх ключів, що дозволяє моделювати складні зв’язки. Наприклад, у базі даних бібліотеки таблиця книг може бути пов’язана з таблицею авторів через унікальний ідентифікатор автора.

Робота з реляційними базами здійснюється за допомогою мови SQL, яка підтримує створення запитів для вибірки, вставки, оновлення чи видалення даних. СУБД, такі як PostgreSQL, MySQL, Oracle або Microsoft SQL Server, забезпечують виконання цих запитів, підтримуючи транзакції за принципом ACID:

— атомарність, виконання операції відбувається повністю, або не відбувається взагалі;

— консистентність, після завершення транзакції база даних залишається в узгодженому стані;

— ізольованість, виконання однієї транзакції не впливає на виконання іншої;

— довговічність, зміни які були внесені транзакцією, збережуться навіть у разі збою.

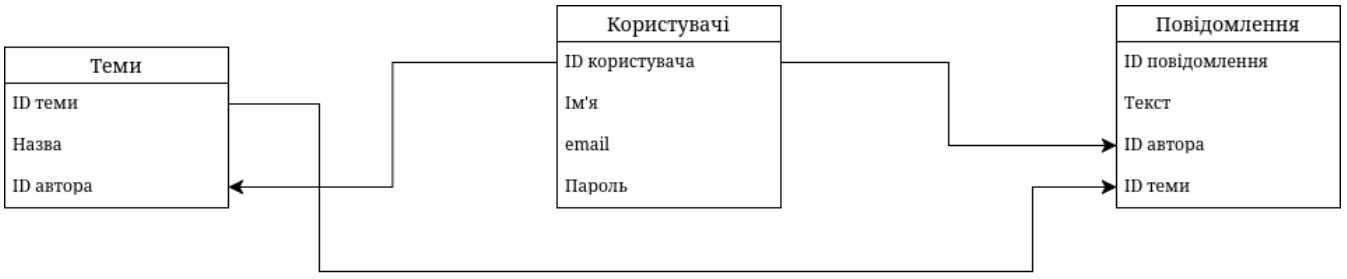

Рисунок 2.3 — Приклад зберігання даних в реляційній СУБД

Ці властивості роблять реляційні бази незамінними для застосунків, де потрібна висока надійність, наприклад, у банківських системах, де транзакція переказу коштів між рахунками має бути виконана без помилок.

Переваги реляційних баз включають чітку структуру даних, що полегшує моделювання зв’язків, стандартизовану мову SQL і широку підтримку інструментів для адміністрування. Однак їхня масштабованість обмежена при обробці великих обсягів неструктурованих даних. Горизонтальне масштабування (додавання серверів) ускладнене через необхідність синхронізації даних між вузлами. Крім того, реляційні бази можуть бути менш ефективними для застосунків із високою частотою читання/запису, таких як соціальні мережі.

Нереляційні бази даних, або NoSQL, виникли як рішення для подолання обмежень реляційних систем у контексті великих даних і розподілених архітектур. Вони відмовляються від традиційної табличної структури, пропонуючи гнучкі моделі даних, що адаптуються до різних потреб.

Такі бази можуть зберігати пари ключ-значення для простих операцій (рис. 2.4).

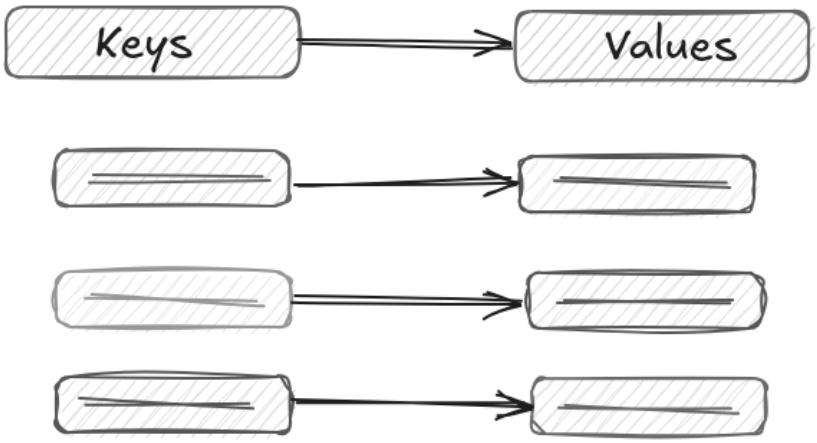

Рисунок 2.4 — Принцип роботи ключ-значення

Це дозволяє швидко отримувати дані за унікальним ключем, що ідеально підходить для кешування чи сесій.

Для складних структур NoSQL підтримує документи у форматі JSON або BSON (рис. 2.5). Такий підхід забезпечує гнучкість при роботі з ієрархічними даними, наприклад, профілями користувачів.

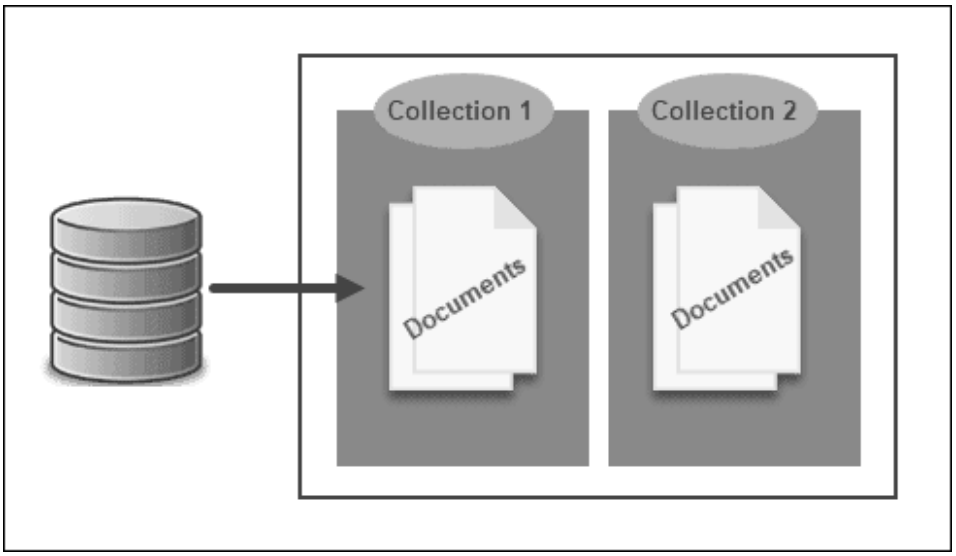

Рисунок 2.5 — Принцип організації зберігання документів у документо-орієнтовані СУБД

Також існують моделі, оптимізовані для аналітичних запитів на великих обсягах даних (рис. 2.6). Вони ефективні для обробки великих масивів інформації, як у бізнес-аналітиці.

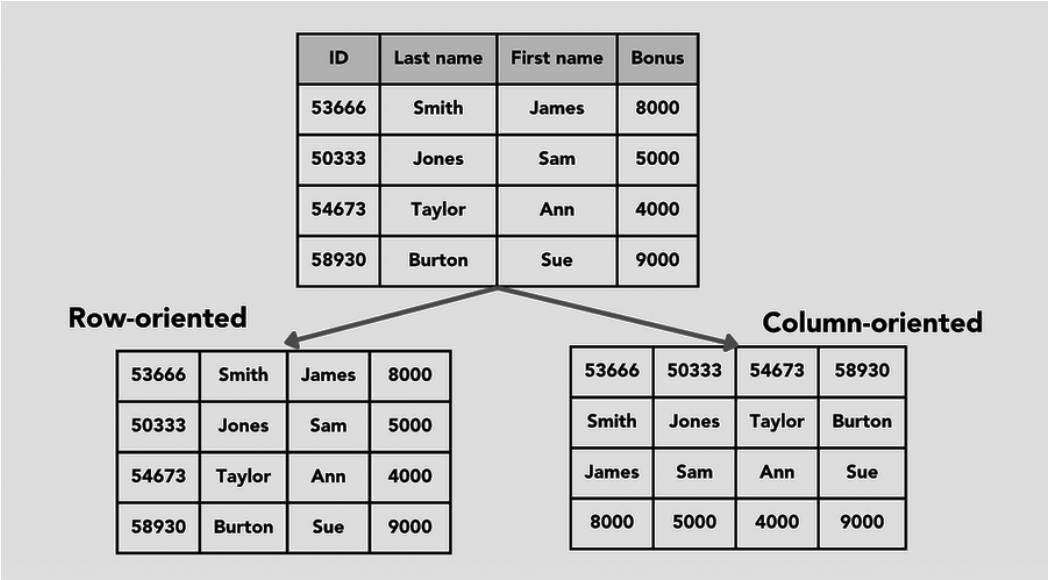

Рисунок 2.6 — Принцип роботи колонкової бази даних

Нарешті, NoSQL включає моделі для роботи зі складними зв’язками між об’єктами (рис. 2.7). Це корисно для систем, де потрібно швидко аналізувати зв’язки, наприклад, у соціальних мережах.

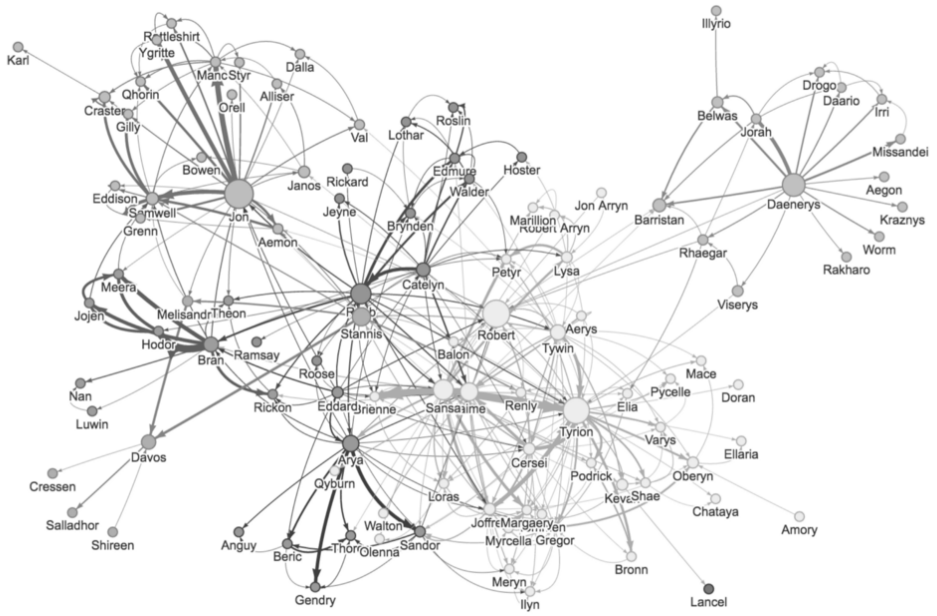

Рисунок 2.7 — Приклад візуалізації графа

NoSQL-бази використовують модель BASE (Basically Available, Soft state, Eventual consistency), що забезпечує високу доступність і толерантність до розподілених збоїв, але допускає тимчасову неконсистентність даних. Наприклад, у системі обміну повідомленнями NoSQL-база може спочатку зберегти повідомлення на одному сервері, а синхронізація з іншими вузлами відбудеться пізніше.

Переваги NoSQL включають високу продуктивність при обробці неструктурованих даних, легкість горизонтального масштабування та гнучкість у роботі з різнорідною інформацією. Наприклад, MongoDB дозволяє зберігати документи з різними полями в одній колекції, що спрощує розробку застосунків із динамічною структурою даних. Однак відсутність стандартизації і складність забезпечення консистентності можуть створювати проблеми в системах, де потрібна висока надійність.

Вибір між SQL і NoSQL залежить від специфіки проєкту. Реляційні бази підходять для систем із чітко визначеною структурою та необхідністю суворої консистентності, таких як фінансові чи логістичні системи. NoSQL ефективні для застосунків із великими обсягами неструктурованих даних, високою частотою

запитів і потребою в масштабуванні, наприклад, у соціальних мережах, системах рекомендацій чи Інтернеті речей. Порівняння типів баз даних наведено в таблиці 2.1.

Таблиця 2.1 — Порівняльна таблиця SQL та NoSQL баз даних

Критерій	SQL Реляційні БД	NoSQL Нереляційні БД
Структура даних	Таблиці з рядками та стовпцями. Жорстка схема.	Документи, ключ-значення, графи, колонки. Гнучка схема.
Мова запитів	SQL (стандартизована).	Різні API (MongoDB Query, CQL для Cassandra).
Масштабування	Вертикальне	Горизонтальне
Транзакції	ACID (атомарність, узгодженість, ізоляція, стійкість).	BASE (Basically Available, Soft state, Eventually consistent).
Швидкість запису	Повільніший через індекси та транзакції.	Швидший через відсутність JOIN та схем.
Швидкість читання	Оптимізовано для складних JOIN-запитів.	Швидкий доступ за ключем або ID.
Використання	Фінанси, ERP, системи зі строгими даними.	Соцмережі, IoT, кешування, великі дані.
Приклади СУБД	PostgreSQL, MySQL, Oracle, MS SQL Server.	MongoDB (документи), Redis (ключ-значення), Cassandra (колонки), Neo4j (графи).
Недоліки	Обмежене масштабування, складність схеми.	Відсутність транзакцій, eventual consistency.

Для даної роботи обрано реляційну базу даних PostgreSQL та ORM (Object-Relational Mapping) інструмент Entity Framework для взаємодії з нею.

PostgreSQL — це потужна реляційна система управління базами даних з відкритим кодом, відома своєю надійністю, гнучкістю та широкими функціональними можливостями. Вона забезпечує повну підтримку транзакцій за принципом ACID, що гарантує атомарність, консистентність, ізольованість і довговічність операцій. Це особливо важливо для систем, де помилки в транзакціях, таких як перекази коштів, можуть мати критичні наслідки. Система

підтримує розширені типи даних, зокрема JSONB, що дозволяє гнучко обробляти неструктуровані дані, наближаючи її до можливостей NoSQL-баз без втрати реляційних переваг. PostgreSQL також пропонує потужні інструменти для складних запитів, включаючи віконні функції, повнотекстовий пошук і підтримку геопросторових даних через розширення PostGIS. Завдяки оптимізації продуктивності, такій як паралельне виконання запитів і різноманітні типи індексів, система ефективно працює з помірними та великими обсягами даних. Відкритий код і активна спільнота забезпечують регулярні оновлення, безпеку та доступ до широкої документації [11].

MySQL, як і PostgreSQL, є СУБД з відкритим кодом, що здобула популярність завдяки простоті та швидкості обробки операцій читання, особливо у веб-додатках. Вона ефективно справляється з простими реляційними структурами та високими навантаженнями на читання, що робить її привабливою для базових проектів. Однак PostgreSQL переважає в кількох аспектах. Вона пропонує ширший набір функцій,, що забезпечує більшу гнучкість для аналітичних і гібридних сценаріїв. Надійність транзакцій у PostgreSQL є вищою, оскільки MySQL, навіть із рушієм InnoDB, може стикатися з обмеженнями при складних операціях або високих навантаженнях. Крім того, PostgreSQL має кращу підтримку стандартів SQL і розширюваність через власні функції та розширення.

SQLite — це легковагова реляційна база даних, призначена для вбудованих систем, мобільних додатків або невеликих проєктів, де не потрібен окремий сервер. Її головна перевага полягає в простоті розгортання, оскільки база зберігається у вигляді єдиного файлу, що усуває необхідність налаштування серверного середовища. SQLite ефективна для сценаріїв із низьким навантаженням і одиничним доступом, таких як локальні додатки. Проте вона має суттєві обмеження порівняно з PostgreSQL. SQLite не підтримує паралельний доступ кількох користувачів на запис, що робить її непридатною для серверних систем із високою конкуренцією [12].

Microsoft SQL Server — комерційна СУБД, що широко застосовується в корпоративних середовищах, особливо в екосистемі .NET. Вона пропонує потужні

інструменти для транзакцій, аналітики та інтеграції з продуктами Microsoft, такими як Azure. SQL Server забезпечує високу продуктивність і підтримку складних запитів, подібно до PostgreSQL. Однак PostgreSQL має кілька переваг. Вона є безкоштовною, що знижує витрати на ліцензування, тоді як SQL Server вимагає значних фінансових вкладень, особливо для великих проєктів. Порівняння характеристик СУБД наведено в таблиці 2.2.

Таблиця 2.2 — Порівняння PostgresSQL з іншими реляційними СУБД

Характеристика	PostgreSQL	MySQL	SQLite	SQL Server
Тип бази	Серверна	Серверна	Вбудована	Серверна
Модель консистентності	ACID	ACID	ACID	ACID
Підтримка JSON	JSONB	JSON	Обмежена	JSON
Складні запити	Висока	Середня	Низька	Висока
Вартість	Безкоштовна	Безкоштовна	Безкоштовна	Платна
Кросплатформність	Висока	Висока	Висока	Середня
Масштабованість	Помірна	Помірна	Низька	Помірна
Паралельний доступ	Високий	Високий	Низький	Високий

2.3 Огляд сучасних фреймворків для написання веб-серверних застосунків

Розробка веб-серверних застосунків вимагає використання фреймворків, які забезпечують швидке створення масштабованих, безпечних і продуктивних систем. Фреймворки спрощують роботу з HTTP-запитами, маршрутизацією, доступом до баз даних і забезпечують структуровану організацію коду. У даному проєкті для створення веб-застосунку обрано фреймворк ASP.NET Core та ORM інструмент Entity Framework Core, шо спрощує взаємодію з базами даних, дозволяючи розробникам працювати з об’єктами замість прямих SQL-запитів.

ASP.NET Core — це кросплатформовий фреймворк із відкритим кодом від

Microsoft, призначений для створення високопродуктивних веб-застосунків. Він базується на платформі .NET і підтримує розробку REST API, веб-додатків із рендерингом на сервері та реального часу через SignalR. Вбудована підтримка залежностей (Dependency Injection) спрощує управління компонентами. ASP.NET Core оптимізований для продуктивності завдяки асинхронній обробці запитів і мінімальним накладним витратам. Його кросплатформність дозволяє розгортати застосунки на Linux, Windows і macOS, що робить фреймворк універсальним для хмарних і локальних середовищ [13].

В екосистемі Core існує декілька інструментів для управління базами даних, найпопулярніший з них Entity Framework. Entity Framework Core — це ORM для .NET, що забезпечує відображення об’єктів C# на реляційні таблиці. Він підтримує PostgreSQL через провайдер Npgsql, дозволяючи використовувати специфічні типи даних, такі як JSONB. LINQ-запити спрощують вибірку та маніпуляцію даними, а автоматизація міграцій полегшує управління схемою бази. EF Core оптимізований для продуктивності завдяки ледачому завантаженню, кешуванню та асинхронним операціям [14].

Django — це фреймворк для мови Python, відомий своєю швидкістю розробки та принципом “батарейки в комплекті”. Він включає вбудовані інструменти для маршрутизації, аутентифікації, адміністрування та роботи з базами даних через власний ORM (Django ORM). Фреймворк дотримується моделі MTV (Model-Template-View), що подібна до MVC, і забезпечує високу безпеку завдяки захисту від поширених вразливостей, таких як SQL-ін’єкції чи XSS-атаки. Django ідеально підходить для швидкого прототипування та створення складних веб-додатків, таких як системи управління контентом. Однак його продуктивність нижча порівняно з ASP.NET Core через інтерпретовану природу Python.

Spring Boot — це фреймворк для мови Java, що спрощує розробку веб-застосунків на основі екосистеми Spring. Він пропонує автоматичну конфігурацію, вбудований сервер і підтримку REST API, MVC і мікро сервісів. Spring Boot інтегрується з Hibernate, популярним ORM для Java, що забезпечує ефективну роботу з базами даних. Фреймворк підходить для великих

корпоративних систем завдяки своїй масштабованій архітектурі та підтримці розподілених систем. Проте складність конфігурації та великий час запуску можуть бути недоліками для невеликих проєктів. Крім того, Java має високі вимоги до пам’яті, що може впливати на витрати в хмарних середовищах.

Express.js — це мінімалістичний фреймворк для Node.js, що використовується для створення REST API та легковагових веб-додатків. Він пропонує гнучкість у виборі бібліотек і інструментів, що дозволяє розробникам налаштовувати стек відповідно до потреб. Express.js працює в асинхронному режимі, що забезпечує високу швидкість обробки запитів, особливо для застосунків із великою кількістю I/O-операцій. Однак його мінімалізм вимагає додаткових бібліотек для таких завдань, як аутентифікація чи робота з базами даних. Порівняння веб фреймворків наведено в таблиці 2.3.

Таблиця 2.3 — Порівняння фреймворків для веб-серверних застосунків

Мова програмування	C#	Python	Java	JavaScript
Продуктивність	Висока	Середня	Висока	Висока (легкі запити)
Швидкість розробки	Висока	Дуже висока	Середня	Середня
Рівень безпеки	Висока	Висока	Висока	Середня
Масштабованість	Висока	Середня	Висока	Середня
Інтеграція з БД	EF Core	Django ORM	Hibernate	Sequelize
Кросплатформність	Висока	Висока	Висока	Висока

Django ORM дозволяє працювати з базами даних через Python-об’єкти. Він підтримує PostgreSQL та інші СУБД, забезпечуючи зручний синтаксис для запитів і автоматичне створення міграцій. Django ORM є інтуїтивним для Python-розробників, але менш гнучким порівняно з EF Core, особливо в складних реляційних сценаріях. Його продуктивність нижча через накладні витрати Python.

Hibernate використовується в Spring Boot. Він пропонує потужні можливості для відображення складних реляційних структур і підтримки транзакцій. Hibernate

підтримує PostgreSQL і забезпечує гнучкість у налаштуванні через анотації або XML. Однак його конфігурація складніша порівняно з EF Core, а продуктивність може бути нижчою.

Sequelize — це ORM для Node.js, що працює з Express.js. Він дозволяє визначати моделі та зв’язки через JavaScript або TypeScript, але його функціональність обмеженіша порівняно з EF Core. Sequelize вимагає ручного налаштування міграцій і менш інтегрований із фреймворком, що ускладнює розробку складних систем. Порівняння ORM-інструментів наведено в таблиці 2.4.

Таблиця 2.4 — Порівняння ORM-інструментів

Характеристика	EF Core	Django ORM	Hibernate	Sequelize
Мова програмування	C#	Python	Java	JavaScript/TypeScript
Інтеграція з фреймворком	ASP.NET Core	Django	Spring Boot	Express.js, Nest.js
Продуктивність	Висока	Середня	Висока	Середня
Гнучкість	Висока	Середня	Висока	Середня
Підтримка міграцій	Автоматизовані	Автоматизовані	Ручні/автоматизовані	Ручні
Підтримка PostgreSQL	Повна	Повна	Повна	Повна

2.4 Огляд сучасних рішень для написання клієнтських додатків

У контексті сучасної веб-розробки спостерігається зростання ролі як клієнтських, так і серверних підходів до побудови інтерфейсів, що зумовлює підвищену увагу до архітектур Single Page Application (SPA) та Server Side Rendering (SSR). Обидва варіанти є відповіддю на потреби в інтерактивності, швидкодії та оптимізації для пошукових систем.

Технологія SSR полягає у генерації HTML-документів на сервері, що надсилаються клієнту вже у готовому вигляді. Це забезпечує миттєве відображення контенту та підвищену індексованість сторінок. SSR особливо

актуальний для сайтів з публічним доступом до контенту, таких як новинні портали, каталоги чи блоги. Але також цей підхід передбачає додаткові витрати ресурсів на сервері, більшу складність налаштування та підтримки. Схеми взаємодії підходів показано на рисунку 2.8.

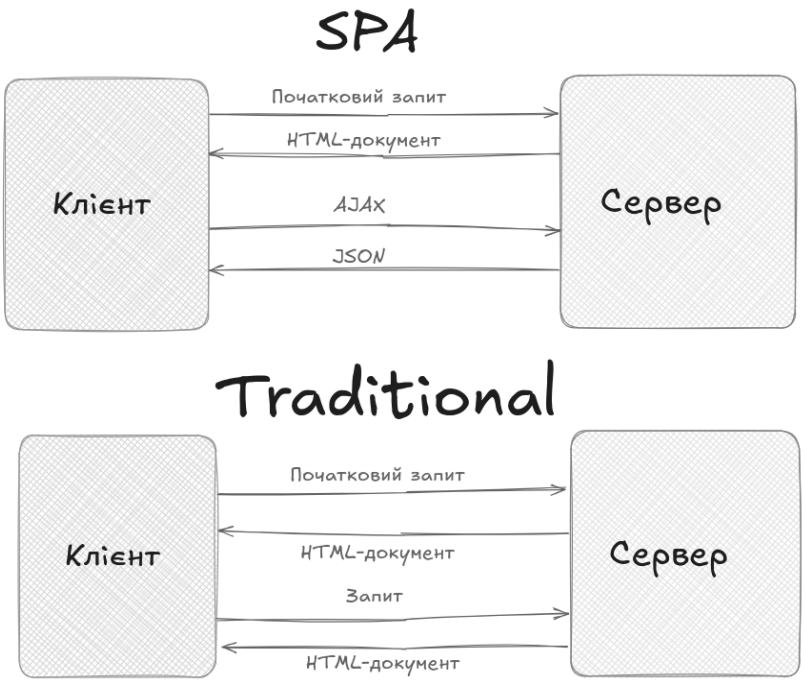

Рисунок 2.8 — Схема взаємодії сервера та клієнта SPA або SSR

SPA — це архітектурний підхід, за якого уся логіка взаємодії інкапсульована в одному HTML-документі, що динамічно оновлюється за допомогою JavaScript. Після первинного завантаження користувач отримує досвід, подібний до десктопного застосунку, без необхідності перезавантаження сторінок. До переваг можна віднести високу інтерактивність, що дозволяє швидко реагувати на дії користувача. Головним недоліком є ускладнення оптимізації для SEO та потенційна затримка при першому завантаженні сторінки.

Одним із основних інструментів розробки клієнтських застосунків є Vite. Vite — це сучасний інструмент фронтенд-розробки, який поєднує швидкий сервер розробки з оптимізованим процесом збірки для продакшену. Створений для прискорення робочого процесу, він пропонує ефективну альтернативу традиційним інструментам, усуваючи їхні основні недоліки, такі як повільний запуск і складна

конфігурація.

Vite працює на основі нативних ES-модулів (ESM), які дозволяють браузеру завантажувати код модульно без попереднього об’єднання в бандли. Це значно скорочує час запуску сервера розробки, роблячи його майже миттєвим навіть у великих проєктах. Замість того, щоб перезбирати весь код при кожній зміні, Vite оновлює лише модифіковані модулі через гарячу заміну модулів (HMR), забезпечуючи швидкі ітерації під час розробки.

Для продакшену Vite використовує Rollup, який створює компактні та оптимізовані бандли. Додатково, завдяки інтеграції з esbuild, Vite прискорює обробку залежностей, зменшуючи кількість HTTP-запитів і покращуючи продуктивність. Такий підхід дозволяє створювати легкі застосунки, які швидко завантажуються в браузері.

Vite вирішує проблему повільних серверів розробки, усуваючи потребу в попередньому бандлінгу. Налаштування спрощене завдяки інтуїтивному конфігураційному файлу, який працює для більшості проєктів без додаткових зусиль. Підтримка TypeScript, JSX і CSS-препроцесорів вбудована, що економить час на інтеграціях.

Універсальність Vite дозволяє працювати з Vue.js, React, Svelte та іншими фреймворками, адаптуючись до різних потреб. Оптимізація коду через розбиття на частини та видалення невикористовуваних фрагментів забезпечує швидке завантаження застосунків у продакшені.

Vite вирішує проблему повільних серверів розробки, усуваючи потребу в попередньому бандлінгу. Налаштування спрощене завдяки інтуїтивному конфігураційному файлу, який працює для більшості проєктів без додаткових зусиль. Підтримка TypeScript, JSX і CSS-препроцесорів вбудована, що економить час на інтеграціях.

Vite надає готові шаблони для ініціалізації проєктів із React, Vue.js, Svelte і Angular, що дозволяє розпочати розробку за лічені хвилини. Завдяки вбудованій підтримці JSX, TypeScript і специфічних для фреймворків функцій, Vite мінімізує необхідність додаткових налаштувань.

Vite підходить для React завдяки швидкій обробці JSX і підтримці HMR, що дозволяє миттєво бачити зміни в компонентах. Його здатність оптимізувати залежності, такі як бібліотеки React, зменшує розмір бандлів, що критично для SPA з великою кількістю компонентів. Початковий проєкт React, який створює Vite показано на рисунку 2.9.

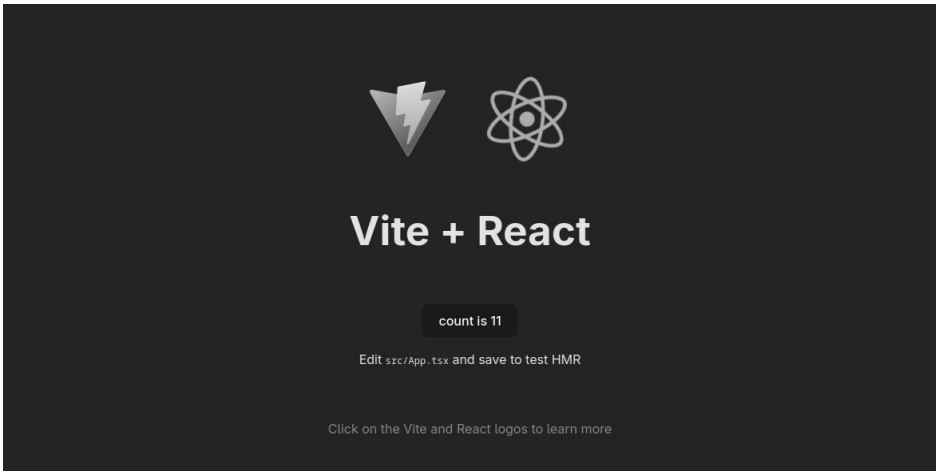

Рисунок 2.9 — Початковий проект Vite+React

Як інструмент, створений автором Vue.js, Vite має глибоку інтеграцію з цим фреймворком. Він оптимізує однофайлові компоненти, забезпечуючи швидке оновлення шаблонів, стилів і логіки. Приклад проєкту показано на рисунку 2.10. Vite також підтримує Vue-специфічні плагіни, що робить його стандартом для Vue-проєктів, особливо в поєднанні з VitePress або Nuxt.

Vite також працює з Svelte, зокрема через SvelteKit, офіційний фреймворк для створення Svelte-застосунків. Завдяки транспіляції Svelte-коду в чистий JavaScript і мінімальній потребі в рантайм-коді, Vite створює надзвичайно легкі бандли. Швидкість HMR ідеально доповнює реактивну природу Svelte, роблячи розробку простішою. Приклад проєкту наведено на рисунку 2.11.

Рисунок 2.10 — Початковий проект Vite+Vue

Рисунок 2.11 — Початковий проект Vite+Svelte

Angular традиційно асоціюється зі складнішими інструментами, але Vite також інтегрується з Angular через експериментальні плагіни або кастомні конфігурації. Він прискорює обробку TypeScript і великих проєктів, але може потребувати додаткових налаштувань для повної сумісності з Angular CLI. Початковий проєкт, показано на рисунку 2.12.

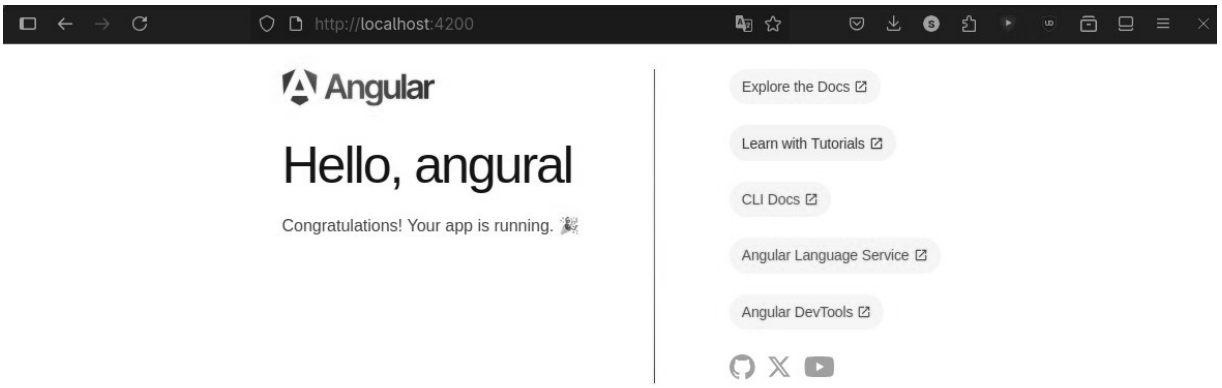

Рисунок 2.12 — Початковий проект Vite+Angular

Серед фреймворків, орієнтованих на серверний рендеринг (SSR), особливу роль відіграють Next.js, Nuxt, SvelteKit та Angular Universal. Ці рішення інтегрують механізми рендерингу сторінок на сервері до моменту їх відправлення на клієнт, що дозволяє досягти високої продуктивності, покращити доступність для пошукових систем та забезпечити стабільний користувацький досвід.

Next.js є найпоширенішим фреймворком SSR для React. Його ключовою особливістю є підтримка кількох стратегій рендерингу: повного SSR, статичної генерації, інкрементальної генерації та гібридних моделей. Next.js автоматично оптимізує маршрути, кешування та попереднє завантаження, що забезпечує високу продуктивність навіть у динамічних застосунках. Структура Крім того, наявність API-роутів дозволяє об'єднати frontend та backend у межах одного проєкту, що особливо зручно для повноцінних веб платформ (рис. 2.13). Next.js також підтримує TypeScript, CSS-модулі та інтеграцію з CDN-платформами, такими як Vercel, що робить його провідним вибором для корпоративних і публічних рішень.

Nuxt, як SSR-фреймворк для Vue, забезпечує аналогічну функціональність у контексті екосистеми Vue.js. Він підтримує кілька режимів роботи: universal (SSR), SPA та static, дозволяючи розробникам адаптувати архітектуру до завдань проєкту. Автоматична маршрутизація, глибока інтеграція з Vuex, Vue Router, а також модульна система Nuxt Modules спрощують розробку та структурування коду (рис. 2.14). З виходом Nuxt 3 фреймворк отримав підтримку Composition API, покращену продуктивність та перехід на Vite як білдер за замовчуванням, що

значно знизило час розробки та збірки. Структура Nuxt, показана на рисунку 2.14.

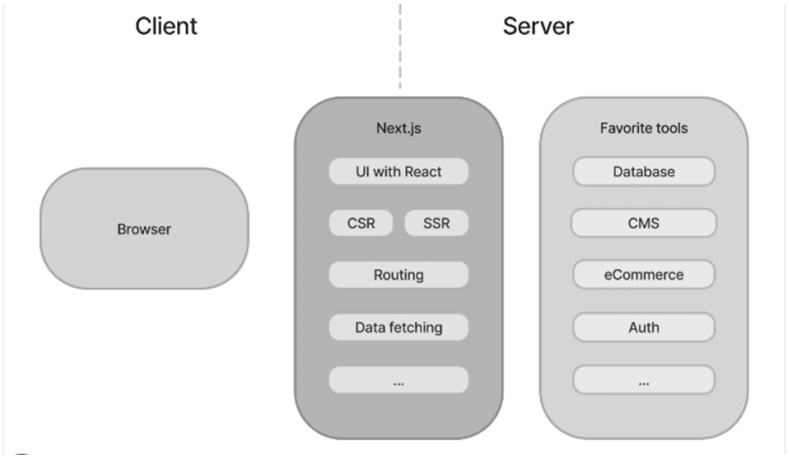

Рисунок 2.13 — Структура фреймворка Next.js

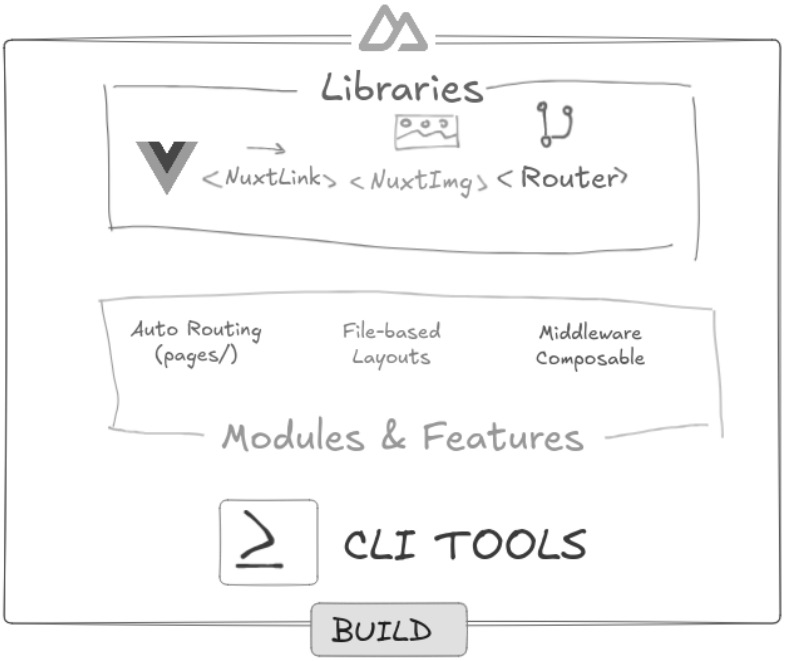

Рисунок 2.14 — Структура фреймворка Nuxt

SvelteKit — офіційний SSR-фреймворк для Svelte, що поєднує компіляційні переваги Svelte з серверною генерацією HTML. Він підтримує як повний SSR, так і статичну генерацію та edge-функції. Особливістю SvelteKit є відсутність рантайм-перевантаження, завдяки чому сторінки завантажуються майже миттєво. Використання адаптерів дозволяє розгортати застосунок у різних середовищах — від Node.js-серверів до serverless-платформ (наприклад, Netlify або Vercel).

Розробка з SvelteKit орієнтована на продуктивність і простоту, що робить його особливо привабливим для проєктів, які потребують високої швидкодії при мінімальних витратах на підтримку.

Angular Universal є офіційним рішенням для SSR у фреймворку Angular. На відміну від інших фреймворків, Angular має строго типізовану структуру, що включає інжекцію залежностей, декоратори та модульну архітектуру, які повністю зберігаються при переході на SSR. Angular Universal дозволяє генерувати повноцінні HTML-документи на сервері, що суттєво покращує час початкового рендерингу та SEO. Однак, через складність налаштування середовища SSR (окрема конфігурація серверної частини, кешування, маршрутизація) він найчастіше використовується в корпоративному сегменті, де виправдана висока початкова вартість розгортання.

Для реалізації клієнтської частини веб застосунку пошуку знайомств було обрано фреймворк Nuxt, оскільки він поєднує в собі гнучкість сучасних інструментів розробки та підтримку серверного рендерингу. Це забезпечує належний рівень продуктивності, особливо на етапі першого завантаження сторінки, і дозволяє досягти необхідної видимості в пошукових системах, що є важливим для публічних платформ.

Nuxt має чітко структуровану архітектуру, що сприяє впорядкованому розвитку проєкту та полегшує підтримку коду. Його інтеграція з Vue забезпечує знайоме середовище для реалізації інтерфейсів, а можливість налаштування рендерингу дозволяє адаптувати поведінку додатку під конкретні завдання.

2.5 Клієнт-серверна взаємодія

Клієнт-серверна взаємодія — це основа архітектури більшості сучасних комп’ютерних систем, зокрема веб застосунків, мобільних додатків, хмарних сервісів та корпоративних систем. Суть цієї взаємодії полягає в тому, що клієнт надсилає запит до сервера, а сервер обробляє цей запит і повертає результат. Незважаючи на видиму простоту, під капотом відбувається багато складних процесів, які забезпечують ефективність, масштабованість і надійність системи

(рис 2.15).

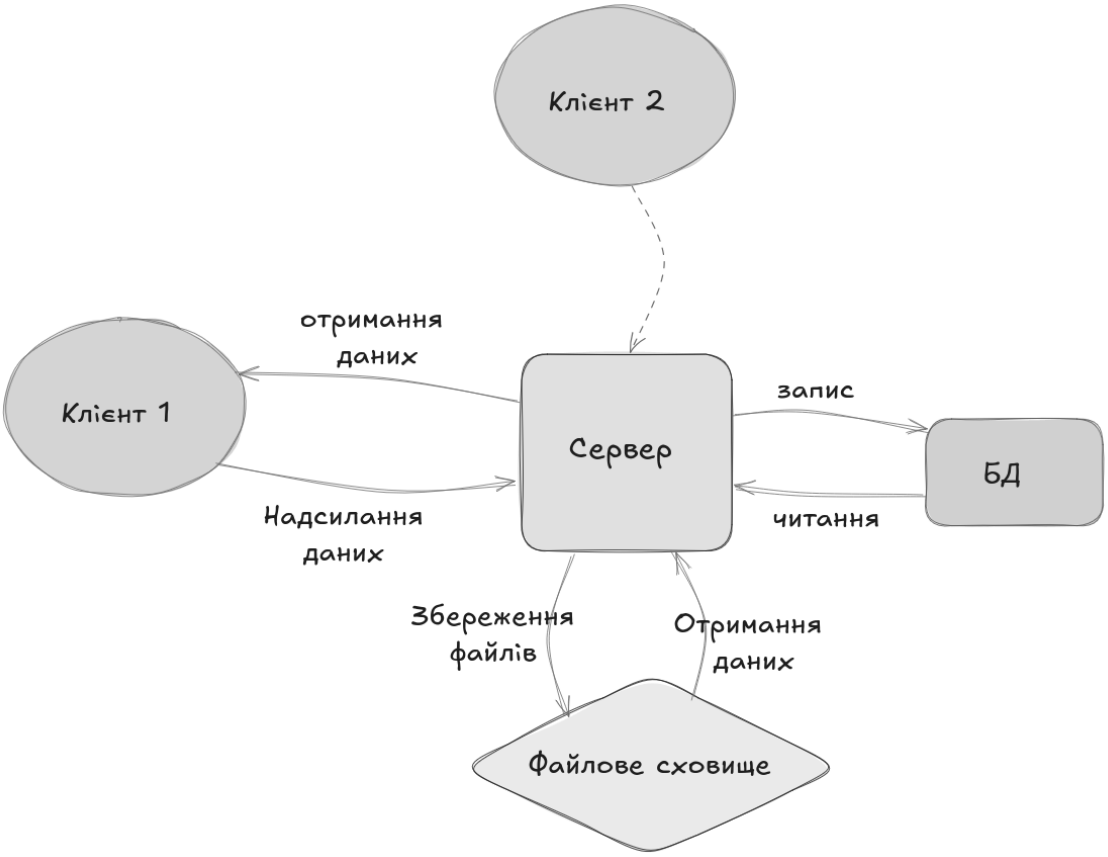

Рисунок 2.15 — Схема клієнт-серверної взаємодії

У цій моделі клієнт ініціює з’єднання. Наприклад, користувач вводить адресу сайту в браузері. Браузер виступає як клієнт і формує HTTP-запит до сервера, IP-адреса якого отримується через DNS. Запит доходить до веб сервера, який визначає, як його обробити: чи повернути статичний HTML-файл, чи звернутись до серверної логіки, яка генерує відповідь динамічно, зокрема через звернення до бази даних. Після цього результат відправляється назад клієнту у вигляді HTTP-відповіді.

Цей обмін може бути синхронним або асинхронним. Синхронна взаємодія передбачає, що клієнт чекає на відповідь, перш ніж перейти до наступної дії. Асинхронна — дозволяє клієнту продовжувати роботу, не чекаючи відповіді, що критично важливо для реального часу, як-от у чатах або відеоконференціях. У таких випадках використовують технології на зразок WebSocket або SignalR.

На рисунку представлено клієнт-серверну взаємодію, де зображено:

— клієнти (Клієнт 1 і Клієнт 2), які надсилають дані або отримують їх;

— сервер, що виступає посередником і координатором всієї взаємодії.

— базу даних, до якої звертається сервер та виконує операції читання або запису;.

— файлове сховище, яке використовується для збереження і отримання різних даних.

Важливо, що клієнт і сервер зазвичай є незалежними системами, які можуть бути реалізовані різними мовами програмування, працювати на різних платформах, але при цьому використовувати єдиний протокол для обміну — найчастіше HTTP або HTTPS. Протокол забезпечує стандартизований спосіб передавання даних, дозволяючи системам взаємодіяти без глибокого розуміння внутрішньої реалізації одна одної. У межах цього протоколу клієнт формує запити різних типів залежно від наміру взаємодії. Найпоширенішим є GET-запит, який використовується для отримання інформації з сервера — наприклад, сторінки або списку товарів. Якщо клієнт надсилає дані, як-от форму з реєстрацією, застосовується POST-запит, який несе в собі тіло з інформацією для обробки. Для оновлення даних використовуються PUT або PATCH — перший замінює ресурс повністю, другий вносить часткові зміни. Видалення ресурсу відбувається за допомогою DELETE-запиту. Такий розподіл дозволяє побудувати чітку та передбачувану логіку взаємодії, зберігаючи одночасно простоту та масштабованість системи.

3 РОЗРОБКА ВЕБ ЗАСТОСУНКУ

3.1 Розробка структури бази даних

Центральною частиною є таблиця Users, яка виступає основним сховищем даних про користувачів. Вона містить унікальний ідентифікатор для кожного учасника, а також основну інформацію, таку як ім’я, вік, стать, біографія, хеш пароля для безпеки, місто проживання, email, статус активності, час останньої активності та уподобання щодо статі потенційного партнера. Ця таблиця стає ядром системи, від якого розходяться зв’язки до інших компонентів, дозволяючи організовано зберігати й обробляти всі взаємодії.

Щоб забезпечити гнучке управління ролями, наприклад, для розмежування прав звичайних користувачів і адміністраторів, створюється таблиця Roles, яка через проміжну таблицю UserRoles пов’язується з Users. Такий підхід реалізує зв’язок "багато-до-багатьох", дозволяючи одному користувачу мати кілька ролей одночасно, а одній ролі відповідати багатьом користувачам. Це додає гнучкості в управлінні доступом і спрощує масштабування системи.

Для зберігання фотографій користувачів створюється таблиця Photos, яка прив’язується до Users через зовнішній ключ. Кожен запис у цій таблиці містить ідентифікатор користувача, тип вмісту зображення та URL для доступу до нього, що дозволяє одному користувачу мати кілька фотографій, які відображаються в його профілі. Це важливий елемент для сервісу знайомств, адже візуальна складова відіграє ключову роль у створенні першого враження.

Таблиця Matches відображає успішні взаємні симпатії між користувачами. Вона фіксує пари через ідентифікатори двох учасників, дозволяючи швидко визначити, хто з ким має збіг, що є основою функціоналу платформи знайомств. Аналогічно, таблиця Likes призначена для зберігання односторонніх симпатій, де вказується, хто саме виявив інтерес до іншого користувача, що дає можливість системі відстежувати активність і пропонувати потенційні збіги.

Таблиця ProfileHistories, яка прив’язана до Users, зберігає історію переглянутих анкет користувачів, щоб виключити повторення. Основні таблиці

бази даних наведено на рисунку 3.1.

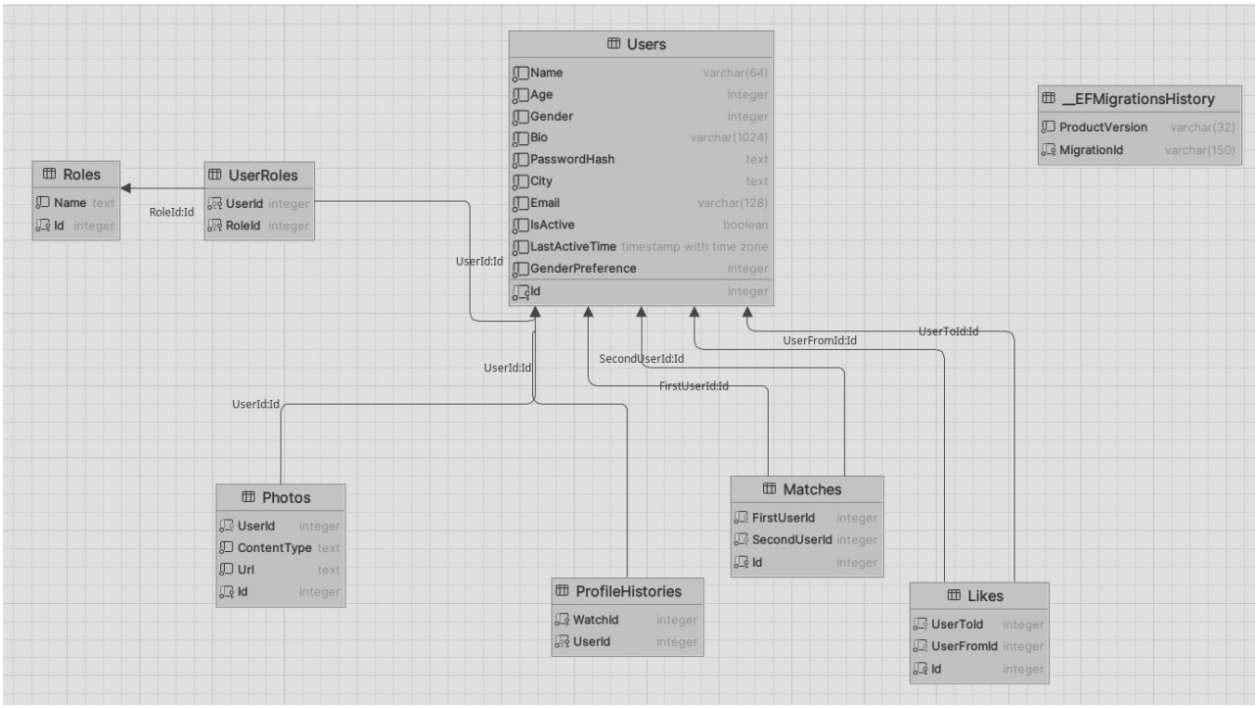

Рисунок 3.1— Таблиці та зв’язки бази даних

3.2 Створення та базове налаштування серверної частини

Створення серверної частини починається з ініціалізації нового проєкту ASP.NET Core Web API. Для цього виконується команда dotnet new webapi -n WebSocket у терміналі, що створює порожній проєкт із базовою структурою для веб-API. Після створення проєкту відкривається сформована директорія, де можна приступити до його налаштування. Спочатку додаються необхідні залежності, такі як Entity Framework Core для роботи з базою даних та SignalR для реалізації чату в реальному часі. Це досягається через команду dotnet add package Microsoft.EntityFrameworkCore та dotnet add package Microsoft.AspNetCore.SignalR, а також підключення провайдера для PostgreSQL через dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL.

Для забезпечення роботи з базою даних налаштовується контейнер PostgreSQL за допомогою Docker. У файлі docker-compose.yml визначається служба postgres із такими параметрами:

```
services:
  postgres:
    image: postgres:15
    container_name: postgresDB
    environment:
      POSTGRES_USER: ymonada
      POSTGRES_PASSWORD: 111
      POSTGRES_DB: appDB
    ports:
      - "5433:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - backend_network
```

Ця конфігурація запускає контейнер із PostgreSQL версії 15 під назвою postgresDB, використовуючи користувача ymonada з паролем 111 та створюючи базу даних appDB. Порт 5432 всередині контейнера мапиться на 5433 на хості, а том postgres_data забезпечує збереження даних між перезапусками. Мережа backend_network дозволяє ізолювати взаємодію між службами. Для запуску контейнера виконується команда docker-compose up -d, що розгортає базу даних у фоновому режимі.

Далі у проєкті створюється структура для сутностей та їх конфігурацій. У папці WebSocket/Entity визначаються класи, такі як User, Role, UserRole, Photo, Like, Match, ProfileHistory та Message, які представляють таблиці бази даних. Для кожної сутності створюються відповідні конфігураційні класи в папці WebSocket.db/Configurations.

```
public void Configure(EntityTypeBuilder<Match> builder){
      builder.HasIndex(m => new { m.FirstUserId, m.SecondUserId }).IsUnique();
      builder.HasOne(u=>u.FirstUser)
         .WithMany(u=>u.Matches)
         .HasForeignKey(u=>u.FirstUserId)
         .OnDelete(DeleteBehavior.Restrict);
      builder.HasOne(u=>u.SecondUser)
         .WithMany()
         .HasForeignKey(u=>u.SecondUserId)
         .OnDelete(DeleteBehavior.Restrict);
      builder.HasMany(m=>m.MessagesHistory)
```

```
            .WithOne(m=>m.Match)
            .HasForeignKey(m=>m.MatchId)
            .OnDelete(DeleteBehavior.Restrict);}
```

Контекст бази даних AppDbContext створюється в директорії WebSocket/db для інтеграції всіх сутностей та задання таблиць за допомогою властивості DbSet<T>, де T — сутність.

```
public class AppDbContext : DbContext{
public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
  public DbSet<User> Users { get; set; }
  public DbSet<Like> Likes { get; set; }
  public DbSet<Role> Roles { get; set; }
  public DbSet<UserRole> UserRoles { get; set; }
  public DbSet<Match> Matches { get; set; }
  public DbSet<ProfileHistory> ProfileHistories { get; set; }
  public DbSet<Photo> Photos { get; set; }
  public DbSet<Message> Messages { get; set; }
  protected override void OnModelCreating(ModelBuilder modelBuilder){
    base.OnModelCreating(modelBuilder);
    modelBuilder.ApplyConfiguration(new UserConfiguration());
    modelBuilder.ApplyConfiguration(new LikeConfiguration());
    modelBuilder.ApplyConfiguration(new RoleConfiguration());
    modelBuilder.ApplyConfiguration(new UserRoleConfiguration());
    modelBuilder.ApplyConfiguration(new PhotoConfiguration());
    modelBuilder.ApplyConfiguration(new ProfileHistoryConfiguration());
    modelBuilder.ApplyConfiguration(new MatchConfiguration());}}
```

Налаштування сервера відбувається в Program.cs, де ініціалізується додаток через var builder = WebApplication.CreateBuilder(args). Додаються сервіси та конфігурації.

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddControllers();
builder.Services.AddSignalR();
builder.Services.AddSwaggerGen(options =>{
   options.SwaggerDoc("v1", new OpenApiInfo { Title = "Some API v1", Version = "v1" });
   options.AddSignalRSwaggerGen();});
builder.Services.AddDbContext<AppDbContext>(options =>options.UseNpgsql(builder.Configuration.GetConnectionString("PostgresConnection")));
```

Рядок підключення до бази даних береться з файлу appsettings.json та має вигляд "PostgresConnection": "Host=localhost;Port=5433;Database=appDB;Username =ymonada;Password=111"

Хаб SignalR для чату мапиться на /chat через app.MapHub("/chat"), а логування налаштовується з рівнем Information для загальних подій та Warning для Microsoft.AspNetCore.

3.3 Реалізація основної логіки

Процес починається з налаштування маршрутів і обробки HTTP-запитів через контролери, кожен з яких захищений атрибутом [Authorize], що гарантує автентифікований доступ до ресурсів. Ідентифікатор користувача витягується з токена автентифікації за допомогою методу User.FindFirstValue(ClaimTypes.NameIdentifier), що дозволяє асоціювати дії з конкретним профілем. Логіка розподілена між контролерами та сервісами, де контролери виступають як інтерфейс для API, а сервіси беруть на себе складні операції з даними та бізнес-логіку, взаємодіючи з базою даних через AppDbContext.

У застосунку контролери та сервіси відіграють ключову роль, вони тісно пов’язані між собою через чіткий розподіл обов’язків: контролери працюють із HTTP-запитами та відповідають за зовнішній інтерфейс взаємодії клієнта з сервером, тоді як сервіси містять бізнес-логіку, пов’язану з перевіркою даних, роботою з базою даних, а також формуванням відповідей.

Контролер AuthController — це звичайний клас ASP.NET Core, який позначений атрибутом [ApiController], що автоматично обробляє перевірку моделі та додає типові поведінки, як-от автоматичне повернення 400, якщо модель недійсна. Він маршрутизований за допомогою атрибуту [Route("[controller]")], що означає, що всі його методи будуть доступні за шляхом /auth. Містить методи Login([FromBody] LoginDto request), RegisterUser([FromBody] RegisterDto user), Logout() та CheckAuth().

Метод Login отримує об'єкт типу LoginDto, що містить дані користувача для

входу, як-от email та пароль. Ці дані передаються у відповідний метод сервісу, де перевіряється існування користувача в базі даних, відповідність пароля та формується результат у вигляді обгортки, яка містить логічний прапорець успішності, повідомлення та, у разі успіху, об’єкт типу ClaimsPrincipal. Останній містить інформацію про користувача, яка буде закодована в сесії. Після цього викликається SignInAsync, який створює сесію через cookie-механізм.

Реєстрація працює аналогічно, але передбачає створення нового користувача у базі. Якщо все проходить успішно, сесія створюється одразу після реєстрації. Це дозволяє уникнути додаткового логіну після підтвердження.

Метод Logout видаляє аутентифікаційний cookie, та метод CheckAuth, який перевіряє, чи є поточний користувач автентифікованим. У цьому методі використовується властивість User.Identity.IsAuthenticated, яка повертає true, якщо сервер ідентифікував користувача як такого, що пройшов автентифікацію.

Уся логіка, пов’язана з перевіркою пароля, створенням ClaimsPrincipal, пошуком у базі та формуванням відповідей, винесена у AuthService. Таким чином, контролер залишається максимально простим і сфокусованим лише на HTTP-рівні.

Окремо варто розглянути TokenService, який відповідає за генерацію JWT-токенів. У цьому класі жорстко задано секретний ключ, термін дії токена, а також видавця та аудиторію. При створенні токена до нього додаються всі необхідні claims: ідентифікатор користувача, його email, а також список ролей. Це дозволяє використовувати створений токен для авторизації доступу до обмежених ресурсів.

Токен створюється за допомогою класів JwtSecurityTokenHandler, JwtSecurityToken, SymmetricSecurityKey та SigningCredentials. Підписання токена відбувається через HMAC SHA256.

```
[HttpPost("login")]
 public async Task<IActionResult> Login([FromBody] LoginDto request){
   var result = await _authService.Login(request);
   if (result.IsSuccess)
          {          await HttpContext.SignInAsync(CookieAuthenticationDefaults
.AuthenticationScheme, result.Data);
```

```
    return Ok();} }
  return Unauthorized(result.Message);
```

Контролер UserController разом із відповідним сервісом UserService обслуговують функціонал, пов’язаний з отриманням, оновленням та активацією профілів користувачів, а також з пошуком інших користувачів для перегляду. Уся взаємодія здійснюється на основі автентифікованого користувача, ідентифікатор якого отримується з JWT-токена за допомогою ClaimTypes.NameIdentifier.

Запит GET /profile дозволяє автентифікованому користувачу отримати інформацію про свій профіль. Контролер витягує ідентифікатор користувача з токена, передає його у відповідний метод сервісу GetUserInfo і, в залежності від результату, повертає або Ok з даними, або BadRequest з повідомленням про помилку.

Ендпоїнт POST /setActive викликає метод сервісу, який активує всі профілі в базі, не лише поточного користувача.

У методі GET /find реалізується логіка пошуку анкет користувачів. Сервіс повертає список користувачів, які активні, ще не переглянуті поточним користувачем, і мають заповнений профіль.

Запит PUT /updateProfile дозволяє автентифікованому користувачу оновити свій профіль, передаючи нові дані у форматі UserUpdateProfileDto. При успішному оновленні повертається повідомлення про успіх.

Сервіс UserService інкапсулює всю бізнес-логіку, пов’язану з профілями користувачів.

Метод GetUserInfo витягує користувача з бази разом із його ролями, фото та отриманими лайками. Якщо користувач існує, будується DTO об’єкт UserProfileDto, який повертається як результат. Інакше повертається повідомлення про помилку.

Допоміжний метод GetUserDto будує об'єкт DTO зі звичайного об'єкта User, включаючи лише потрібні поля: ім’я, вік, місто, опис, стать, перевагу та фото.

Метод UpdateMyProfile оновлює дані користувача через LINQ-запит ExecuteUpdateAsync, що дозволяє виконати оновлення без завантаження сутності в

пам’ять. Також перевіряється валідність віку. Метод повертає об'єкт ServiceResult з відповідним повідомленням.

Для масового оновлення активності існує метод SetActiveAll, який встановлює IsActive = true для всіх користувачів у базі даних.

Метод Finding реалізує основну логіку пошуку анкет. Спочатку витягується поточний користувач разом із пов’язаними об’єктами: фото, лайками, ролями та історією переглядів. Далі відбувається спроба знайти активних користувачів, яких цей користувач ще не переглядав. Якщо таких менше одного, історія переглядів очищується, і пошук повторюється. Якщо навіть після очищення користувачів не знаходиться, повертається повідомлення про помилку. Інакше повертається список DTO з профілями користувачів.

```
public async Task<ServiceResult<List<UserProfileDto>>> Finding(int userId){
    var currentUser = await _context.Users
       .Include(u => u.Photos)
       .Include(u=>u.ReceiveLikes)
       .Include(u=>u.Roles)
       .ThenInclude(ur => ur.Role)
       .Include(u=>u.ProfileHistory)
       .FirstOrDefaultAsync(u => u.Id == userId);
    if (currentUser == null){
       return new ServiceResult<List<UserProfileDto>>{
          IsSuccess = false, Message = "User not found"};}
    var activeUser = await GetActivityUsers(currentUser);
    if (activeUser.Count >= 1)
       return Responce();
    await ClearHistory(userId);
    activeUser = await GetActivityUsers(currentUser);
    if (activeUser.Count <= 1){
       return new ServiceResult<List<UserProfileDto>>{
          IsSuccess = false, Message = "Users not found"};}
    return Responce();
    ServiceResult<List<UserProfileDto>> Responce(){
       return new ServiceResult<List<UserProfileDto>>{
          IsSuccess = true,
          Message = "Users found",
          Data = activeUser.Select(u => new UserProfileDto(
             u.Id
             , u.Name
```

```
                , u.Age
                , u.City
                , u.Bio
                , u.Gender
                ,u.GenderPreference
            , u.Photos.Select(r => new PhotoDto(r.Id, r.Url, r.ContentType)).
ToList())).ToList()};}}
```

У методі GetActivityUsers фільтруються користувачі, які:

— активні;

— ще не переглянуті поточним користувачем;

— не є самим поточним користувачем;

В результаті вибирається максимум 5 анкет, впорядкованих за часом останньої активності.

Метод AddToHistory додає запис до таблиці ProfileHistories, якщо такий ще не існує для пари користувачів (той, хто переглядає, і той, кого переглянули). Також після перегляду активується профіль користувача, який переглянув анкету.

Окремо існує допоміжний метод ClearHistory, який повністю очищає історію переглядів для заданого користувача. Це використовується, коли знайти нових користувачів для перегляду не вдалося.

Також міститься функціонал для роботи з фотографіями користувачів, реалізований у контролері PhotoController та сервісі PhotoService. Користувач, який пройшов автентифікацію, може завантажувати свої фотографії через HTTP-запит із формою. Після отримання файлів контролер звертається до PhotoService, який виконує всі необхідні операції зі збереження даних. Спочатку видаляються всі попередні фотографії користувача — як з бази даних, так і з файлової системи. Потім кожен файл обробляється окремо: з нього зчитується MD5-хеш, на основі якого створюється багаторівнева структура директорій, що допомагає розподіляти файли по підкаталогах. Файли зберігаються в каталозі wwwroot/photos, а URL-шляхи до них разом із типом вмісту записуються до таблиці бази даних.

Після збереження даних користувач отримує список посилань на завантажені фотографії. У будь-який момент можна виконати запит на видалення всіх

зображень користувача, після чого програма видаляє як записи в базі, так і відповідні файли. Крім того, є можливість отримати конкретну фотографію за вказаним шляхом. Для цього система декодує переданий шлях, шукає файл у файловій системі, визначає MIME-тип за розширенням та повертає файл як відповідь.

```
[HttpPut("/upload")]
 [Authorize]
   public async Task<IActionResult> UploadPhoto([FromForm] List<IFormFile>
files) {
 var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
 var result = await _photoService.UpdateMyProfilePhotos(int.Parse(userId), files);
     if (result.IsSuccess){
        return Ok(result.Data);}
     return BadRequest(result.Message);  }
```

У модулі взаємодії з лайками реалізовано ключовий механізм соціального функціоналу — надсилання вподобань між користувачами, реакції на ці вподобання та формування пар. Основна логіка зосереджена в контролері LikeController та сервісі LikeService.

Коли користувач відправляє лайк, його ідентифікатор береться з JWT-токена, який розшифровується через ClaimTypes.NameIdentifier. Потім сервіс перевіряє, чи вже існує лайк між цими користувачами. Якщо так — повертається відповідь про наявний лайк. Якщо ні — створюється новий об'єкт типу Like, додається до бази даних, і паралельно оновлюється історія переглядів, щоб відобразити факт взаємодії.

При відправці дизлайку об'єкт у базу не записується — натомість лише зберігається запис у історії взаємодій. Цей механізм дозволяє не дублювати інформацію і зменшити кількість операцій з базою.

Коли користувач отримує лайк, він може відповісти на нього. Якщо відповідь — лайк, і це вподобання є взаємним, то програма формує нову пару. Для цього перевіряється наявність зворотного лайка і, якщо пара ще не створена, записується новий об'єкт Match. Якщо ж користувач відповідає дизлайком, то лайк видаляється, і взаємодія припиняється без формування пари.

Також реалізовано можливість перегляду всіх користувачів, які поставили лайк поточному користувачеві. Для цього відбувається пошук у таблиці Likes за ідентифікатором отримувача, і по кожному знайденому лайку формується DTO-модель, яка містить повну публічну інформацію про користувача: ім’я, вік, місто, біографію, стать, вподобання та фотографії.

Система побудована з урахуванням перевірок: усі запити вимагають автентифікації, а в разі відсутності потрібного користувача або лайка сервіс повертає відповідне повідомлення про помилку.

Для управління діалогами між користувачами, які мають взаємні симпатії використовується ChatController, він визначає базовий шлях /Match для всіх своїх ендпоінтів, забезпечуючи чітку організацію API. Контролер залежить від контексту бази даних AppDbContext, який передається через конструктор для доступу до даних про матчі, користувачів і повідомлення. Усі методи контролера захищені атрибутом [Authorize], що вимагає автентифікації, а ідентифікатор поточного користувача витягується з токена автентифікації через User.FindFirstValue(ClaimTypes. NameIdentifier) для асоціації запитів із конкретним профілем.

Метод, Dialogs(), відповідає за повернення списку користувачів, з якими поточний користувач має активні матчі, тобто потенційні діалоги. Спочатку метод перевіряє наявність ідентифікатора користувача в токені. Якщо ідентифікатор відсутній або порожній, повертається статус 401 Unauthorized з повідомленням про помилку. Після успішного отримання ідентифікатора, який конвертується в ціле число, виконується запит до таблиці Matches. Запит фільтрує записи, де поточний користувач є або FirstUserId, або SecondUserId, і витягує ідентифікатор партнера по матчу, використовуючи умовний вираз для визначення, хто з пари є співрозмовником. Якщо матчів немає, метод повертає порожній список об’єктів UserProfileDto, що дозволяє клієнту коректно обробити відсутність діалогів.

Далі, якщо матчі знайдено, виконується запит до таблиці Users, який включає пов’язані фотографії через Include(u => u.Photos) і фільтрує користувачів за списком ідентифікаторів партнерів. Для кожного користувача створюється

об’єкт UserProfileDto, що містить ідентифікатор, ім’я, вік, місто, біографію, стать, уподобання щодо статі та список фотографій, представлених як PhotoDto з ідентифікатором, URL і типом вмісту. Результат повертається зі статусом 200 OK, надаючи клієнту повний список потенційних співрозмовників із їхніми профілями.

Метод, GetChatHistory, відповідає за отримання історії повідомлень між поточним користувачем і конкретним партнером, ідентифікатор якого передається як параметр partnerId. Метод також починається з отримання ідентифікатора поточного користувача, але якщо токен відсутній, генерується виняток UnauthorizedAccessException. Після цього виконується запит до таблиці Matches, який шукає матч між двома користувачами, перевіряючи обидва можливі напрямки зв’язку (FirstUserId і SecondUserId). Якщо матч не знайдено, повертається статус 404 Not Found із повідомленням про відсутність матчу, що сигналізує клієнту про неможливість отримати історію діалогу.

```
[HttpGet("/dialogs/{partnerId}/history")]
  [Authorize]
  public async Task<IActionResult> GetChatHistory(int partnerId){
        var userId = int.Parse(User.FindFirstValue(ClaimTypes.NameIdentifier) ??
throw new UnauthorizedAccessException("User ID not found"));
      var match = await _context.Matches
        .FirstOrDefaultAsync(m => (m.FirstUserId == userId && m.SecondUserId
== partnerId) ||
   (m.FirstUserId == partnerId && m.SecondUserId == userId));
      if (match == null)
        return NotFound("Match not found");
      var messages = await _context.Messages
        .Where(m => m.MatchId == match.Id)
        .OrderBy(m => m.SentAt)
        .Select(m => new MessageDto{
          SenderId = m.SenderId,
          Content = m.Content,
          SentAt = m.SentAt,
          IsRead = m.IsRead})
        .ToListAsync();
      return Ok(messages); }
```

При наявності матчу виконується запит до таблиці Messages, який фільтрує повідомлення за ідентифікатором матчу (MatchId), сортує їх за часом відправлення

(SentAt) і перетворює в об’єкти MessageDto. Кожен MessageDto містить ідентифікатор відправника, вміст повідомлення, час відправлення та статус прочитання. Результат повертається зі статусом 200 OK, надаючи клієнту впорядковану історію повідомлень для відображення в інтерфейсі чату. Використання асинхронних операцій із ToListAsync і включення пов’язаних даних через Include забезпечує ефективну роботу з базою даних, мінімізуючи затримки. Схема роботи веб сервера показана на рисунку 3.2.

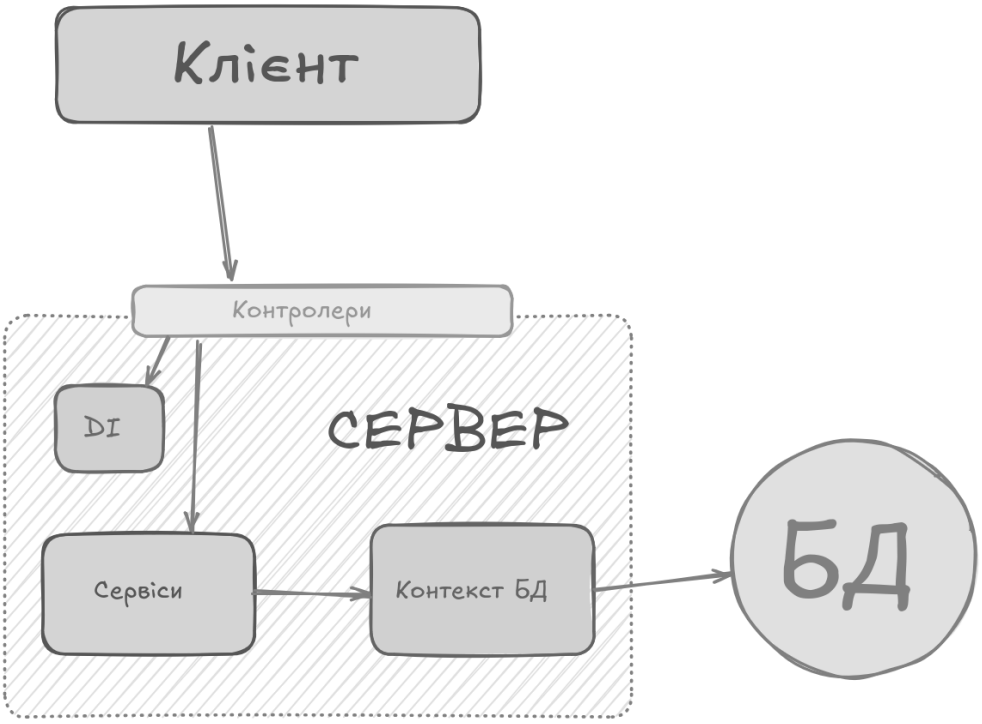

Рисунок 3.2— Схема веб серверу

3.4 Тестування та перевірка коректності роботи

Тестування веб-сервера відбувається за допомогою Swagger. Запит POST /auth/register (рис. 3.3), з тестовими даними name: "dima", email: "dima12@mail.com", password: "dima", відправленими у JSON-тілі без параметрів, який повертає код 200 з заголовками, що включають автентифікаційні cookie, підтверджуючи успішну реєстрацію.

Рисунок 3.3 — Запит на реєстрацію

Далі виконується запит GET /profile (рис. 3.4), без параметрів, який повертає код 200 з JSON-об’єктом користувача, що відображає початковий стан профілю.

Рисунок 3.4 — Запит на отримання профілю користувача

Потім виконується PUT /upload (рис. 3.5.), з фотографіями профілю, (наприклад, наприклад два файли 78.webp та beach.webp) у форматі multipart/form-data, який повертає код 200 з JSON-масивом photos, що містить URL завантажених зображень, підтверджуючи успішне завантаження.

Рисунок 3.5 — Запит на завантаження фотографій користувача

Повторний запит GET /profile (рис 3.6), повертає код 200 з оновленим JSON-об’єктом, де поле photos тепер включає URL зображень, узгоджених із завантаженням, демонструючи коректне оновлення профілю.

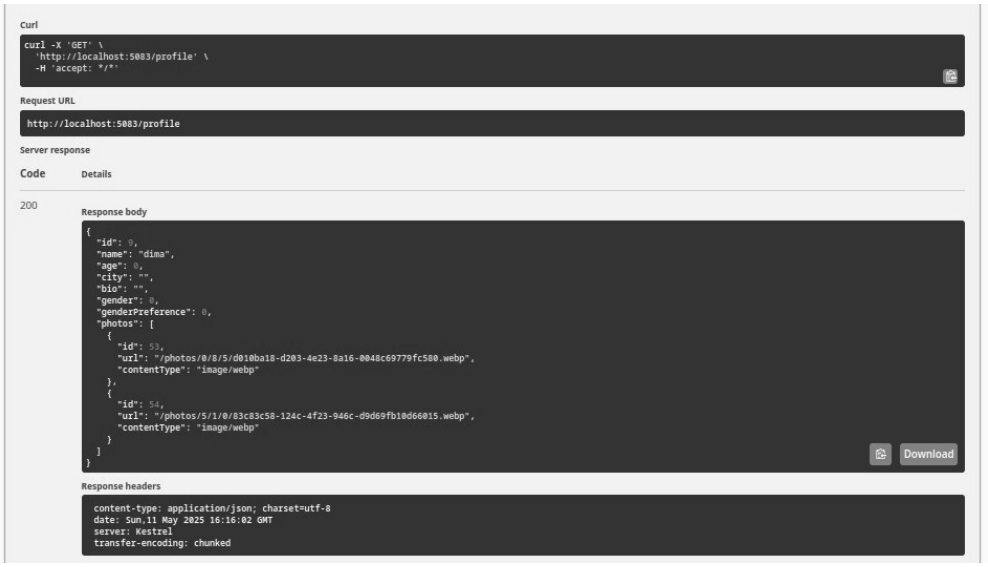

Рисунок 3.6 — Повторний запит на отримання профілю користувача

Запит GET /find (рис 3.7), без параметрів повертає код 200 з масивом, що містять дані інших користувачів, підтверджуючи успішний пошук профілів.

Рисунок 3.7— Запит на отримання інших користувачів

3.5 Налаштування CORS для взаємодії між клієнтом та сервером

Nuxt, як фреймворк для клієнтської частини, може працювати на домені, наприклад, https://localhost:3000, тоді як сервер API на ASP.NET розміщений на http://localhost:5083. Браузери за замовчуванням блокують кросдоменні запити через політику Same-Origin, тому сервер має явно дозволити доступ із клієнтського домену. CORS вирішує цю проблему, дозволяючи серверу вказати, які джерела, методи та заголовки є допустимими.

ASP.NET Core підтримує вбудовану конфігурацію CORS через middleware. У проєкті потрібно налаштувати CORS-політику в Program.cs. Спочатку визначається політика, яка вказує дозволені домени, методи та заголовки.

```
builder.Services.AddCors(options =>
{
  options.AddPolicy("AllowFrontend", policy =>
     policy.WithOrigins("https://localhost:3000")
        .AllowAnyHeader()
        .AllowAnyMethod()
        .AllowCredentials());
});
```

Ця конфігурація дозволяє запити лише з https://localhost:3000, підтримує будь-які методи (GET, POST, PUT, DELETE) і заголовки, а також передачу автентифікаційних даних, що важливо для сесій або токенів. AllowCredentials() вимагає конкретного джерела.

Також слід увімкнути HTTPS для всіх запитів, щоб уникнути витоку даних. У ASP.NET Core це налаштовується через app.UseHttpsRedirection() у Program.cs.

3.6 Створення та налаштування клієнтської частини

Розробка клієнтської частини починається з налаштування середовища Nuxt 3, яке забезпечує модульну архітектуру, автоматичне маршрутизування, серверний рендеринг і сучасний підхід до побудови SPA. Для стилізації і готових UI-компонентів обрано Quasar Framework, що інтегрується з Vue.js і дозволяє швидко реалізовувати адаптивні, інтуїтивні інтерфейси без необхідності писати багато CSS.

Конфігурація проєкту зосереджена у файлі nuxt.config.ts, де прописані базові параметри: підтримка TypeScript, інтеграція Quasar як UI-бібліотеки, налаштування маршрутизації, і налаштування серверного середовища. Важливим кроком є підключення плагінів, які відповідають за роботу з API, управління станом, а також підтримку авторизації.

Кореневий компонент app.vue задає структуру головної сторінки, де розміщуються навігація, динамічний контент сторінок та глобальні елементи інтерфейсу. Така структура дозволяє організувати чітку ієрархію UI та полегшує розробку.

Компоненти Quasar, такі як кнопки, форми, діалоги, списки, використовуються для реалізації інтерфейсу на всіх сторінках. Для інтерактивних елементів — наприклад, слайдерів у профілях чи інших розділах — застосовується Swiper, що забезпечує плавну та гнучку анімацію.

Усі сторінки реалізовані як окремі Vue-компоненти в папці pages, що відповідає автоматичному маршрутизуванню Nuxt. Це дає можливість додавати новий функціонал просто шляхом створення нових файлів із логічними іменами.

Механізм аутентифікації користувача — один із найважливіших у проєкті, оскільки він забезпечує безпеку та персоналізацію роботи.

Форма входу (login.vue) включає стандартні поля email і пароль, оформлені компонентами Quasar, що підтримують валідацію і відображення помилок. На клієнті виконується базова перевірка коректності введених даних — наявність значень, правильність формату email. Після цього відправляється асинхронний запит на бекенд за допомогою функції loginFetch, яка інкапсулює логіку HTTP-виклику.

У разі успішної аутентифікації сервер повертає JWT-токен, який клієнт зберігає у HTTPOnly cookie. Збереження у cookie, недоступному для JavaScript, суттєво знижує ризики атак типу XSS. Клієнтський застосунок у відповідь оновлює внутрішній стан сесії, активує middleware Nuxt для захисту приватних маршрутів і перенаправляє користувача на основну сторінку.

```
<template>
 <div v-if="!isAuth" class="loginForm">
    <q-input class="dataInput" standout="bg-blue text-white" label-color="white" v-model="email" type="email" label="Email" />
    <q-input class="dataInput" standout="bg-blue text-white" label-color="white" v-model="password" type="password" label="Password" />
   <q-btn @click="login" color="primary" class="loginBtn">Login</q-btn>
 </div>
</template>
<script setup lang="ts">
import { onMounted } from 'vue';
import { fetchLogin } from '@/services/fetchService'
import { checkAuth } from '@/services/fetchService';
let isAuth = ref<boolean>(false);
const email = ref<string>('');
const password = ref<string>('');
const login = async () => {
await fetchLogin(email.value, password.value)
await navigateTo('/profile')
}
onMounted(async ()=>{
 isAuth.value = await checkAuth();
 if(isAuth.value){
   navigateTo('/profile')}
```

```
})
</script>
```

Процес реєстрації реалізований у файлі register.vue подібним чином. Форма містить поля для введення додаткових даних — ім’я, пароль з підтвердженням. Валідація охоплює логіку перевірки відповідності паролів та форматів. Після відправлення інформації створюється новий користувач на сервері. У разі успіху, користувач перенаправляється на сторінку зі своїм профілем.

Вихід із системи реалізується через виклик API, який інформує сервер про припинення сесії, а також через видалення cookie з токеном на клієнті. Після цього відбувається редирект на сторінку логіну.

Перевірка активності сесії відбувається на сторінці active.vue, де виконується запит до сервера. Якщо сесія не активна, користувача перенаправляють на сторінку входу.

Для реалізації функціоналу пошуку інших користувачів у системі передбачено окрему сторінку, компоненти якої взаємодіють із бекендом через API та динамічно оновлюють інтерфейс згідно з критеріями пошуку.

Пошук анкет користувачів реалізований через компонент, який завантажує профілі з сервера, відображає їх у вигляді карток із можливістю перегляду фотографій та інтерактивними кнопками дій. Для виводу фотографій використовується компонент SwiperComponent, що забезпечує плавну навігацію між зображеннями, а основна інформація профілю відображається через ProfileComponent.

Дані про профілі завантажуються при монтуванні компонента за допомогою асинхронної функції fetchLikes(), яка звертається до API через сервіс fetchFindProfiles. Отримані профілі зберігаються у реактивній змінній profile, а індекс поточного перегляду — у current.

Перемикання між профілями відбувається через функцію nextProfile(). Якщо користувач переглянув усі наявні анкети, відбувається повторне завантаження нових профілів із сервера. Це забезпечує безперервний потік анкет для перегляду.

Інтерактивність досягається через блок ActionButtons, який надсилає дії

користувача (наприклад, лайк, повідомлення чи дизлайк) до API відповідними викликами: fetchSendLike або fetchSendDislike. Після виконання дії автоматично відбувається перехід до наступного профілю.

```
onMounted(() => {
 fetchLikes();
});
function nextProfile() {
 if (profile.value && current.value < profile.value.length - 1) {
   current.value++;
 } else {
   profile.value = null;
   fetchLikes(); // Завантажуємо нові профілі
 }}
async function handleAction(action: string): Promise<void> {
 if (!profile.value || !profile.value[current.value]) return;
 const userId = profile.value[current.value].id;
 if (action === 'approve') {
   await fetchSendLike(userId);
 } else if (action === 'message') {
   console.log('Надіслати повідомлення');
 } else if (action === 'refresh') {
   await fetchSendDislike(userId);}
```

У шаблоні компонента реалізовано умовне відображення: якщо профілі завантажені, показується картка з фотографіями та інформацією про користувача; у разі відсутності даних виводиться індикатор завантаження. Такий підхід підвищує UX, інформуючи користувача про статус отримання даних.

Взаємодія з бекендом інкапсульована у сервісних функціях (fetchService), що дозволяє централізовано контролювати API-запити і спрощує тестування.

Таким чином, реалізація пошуку анкет організована як зручний для користувача карусельний перегляд із можливістю швидко приймати рішення щодо кожного профілю, що підвищує ефективність і привабливість функції.

Функціонал чатів у вебзастосунку реалізовано за допомогою декількох Vue-компонентів, які забезпечують повноцінний обмін повідомленнями між користувачами. Основний компонент — ChatComponent.vue — відповідає за відображення активного діалогу, історії листування та відправку нових

повідомлень. Компонент chatPreviewComponent.vue відображає короткий прев'ю кожного діалогу у списку чатів, дозволяючи швидко вибирати співрозмовника.

При монтуванні основного компонента виконується завантаження списку діалогів користувача з бекенда за допомогою асинхронної функції fetchGetChats(). Отримані профілі співрозмовників зберігаються у реактивній змінній chatsPreview. Для завантаження історії повідомлень із конкретним співрозмовником використовується функція fetchChatHistory(partnerId), яка повертає масив повідомлень.

```
async function fetchChats() {
 try {
   const chats = await fetchGetChats();
   console.log('Fetched chats:', chats);
   chatsPreview.value = chats;
 } catch (error) {
   console.error('Error fetching chats:', error);
 }}
async function loadChatHistory(partnerId: number) {
 try {
   const history = await fetchChatHistory(partnerId);
   messages.value = history.map(msg => ({
     senderId: msg.senderId.toString(),
     text: msg.content
   }));
 } catch (error) {
   console.error('Error loading chat history:', error);
 }}
function getCurrentId(): string | undefined {
 return currentUser.value?.id.toString();}
```

Вибір діалогу здійснюється через функцію handleOpen(profile), що задає поточного співрозмовника, очищує локальні повідомлення та завантажує історію листування.

Для надсилання повідомлень використовується інтеграція з SignalR-сервісом, що дозволяє забезпечити реальний час обміну даними. Метод sendMessage надсилає текстове повідомлення на сервер, а локально одразу додає його до списку для миттєвого відображення.

```
function handleOpen(profile: UserProfile) {
 selectedChat.value = profile;
 messages.value = [];
 loadChatHistory(profile.id);
 console.log('Selected chat:', profile);}
function closeChat() {
 selectedChat.value = null;
 messages.value = [];}
async function sendMessage(message: string) {
 if (selectedChat.value) {
   await signalRService.sendMessage(selectedChat.value.id.toString(), message);
   messages.value.push({ senderId: 'me', text: message }); }}
```

Компонент ChatComponent.vue містить UI для відображення повідомлень із поділом на відправлені користувачем та отримані, а також поле введення тексту з кнопкою відправлення. Повідомлення групуються за відправником, кожне має відповідне стилізоване оформлення.

Компонент chatPreviewComponent.vue у списку чатів відображає аватар, ім’я співрозмовника та прев’ю останнього повідомлення. Елемент списку реагує на подію кліку, яка відкриває повноцінний чат.

Важливою частиною є управління підписками на події SignalR — під час монтування компонента встановлюється прослуховування вхідних повідомлень, які у реальному часі додаються до відповідного списку повідомлень. Під час демонтажу компонента відбувається відписка, що запобігає витоку пам’яті.

ВИСНОВКИ

У ході виконання бакалаврської роботи було розроблено та впроваджено інформаційну систему онлайн-сервісу пошуку знайомств, яка відповідає сучасним вимогам цифрової епохи та забезпечує ефективну взаємодію користувачів у глобальному просторі. Основна мета полягала у створенні інтерактивної веб-платформи, що дозволяє реєструватися, виконувати пошук та фільтрацію анкет за різноманітними критеріями, а також підтримувати обмін повідомленнями в режимі реального часу. Розроблена система орієнтована на широку аудиторію користувачів, що прагнуть швидко і безпечно знаходити соціальні контакти.

У процесі реалізації було проаналізовано та використано сучасні веб технології, зокрема Nuxt.js для створення клієнтської частини, що забезпечує високу швидкодію, реактивність і SEO-оптимізацію, а також ASP.NET для серверної частини, що гарантує надійність, масштабованість і захист даних. Особливу увагу було приділено інтеграції цих технологій, що дозволило забезпечити стабільну та ефективну взаємодію між фронтендом і бекендом. Інтерфейс системи спроектовано з урахуванням адаптивності для роботи на різних пристроях, а також з використанням сучасних UI-компонентів.

Структура системи базується на розділених функціональних модулях, які включають управління профілями, механізми пошуку і фільтрації анкет, а також систему обміну повідомленнями між користувачами. Кожен з цих модулів розроблений із застосуванням практик безпеки, включаючи захист персональних даних і контроль доступу. Це дозволяє користувачам почуватися впевнено під час роботи із платформою і гарантує конфіденційність їхньої інформації.

Впроваджена система забезпечує логічний, інтуїтивно зрозумілий процес пошуку і комунікації, що дозволяє користувачам самостійно керувати процесом знайомств, налаштовувати параметри пошуку та вести ефективне спілкування. Реалізація сучасних технологічних рішень дає змогу платформі ефективно працювати з великим навантаженням і відкриває можливості для подальшого розвитку та масштабування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сухина Є. В., Добровольська Н. В. Інформаційна система як засіб цифрової соціалізації // Матеріали всеукраїнської науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців

«Молодь в науці: дослідження, проблеми, перспективи (МН-2025)» (15 листопада 2024 р . - 14 червня 2025 р., Вінниця) : — Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/viewFile/25371/20954

2. Цифрова трансформація суспільства URL: https:///lib.iitta.gov.ua/id/eprint/742497/1/Ничкало Лукянова Овчарук -74-98.pdf

3. 10 вимог до сучасного веб-сайту URL: laweb.com.ua/uk/blog/top-10-vimog-do-suchasnogo-veb-saytu

4. Створення сайту знайомств [Електронний ресурс]. — outsourcing.team/ua/blog/development/stvorennya-sajtu-znajomstv/

5. Tinder | Dating, Make Friends & Meet New People [Електронний ресурс]. — https://tinder.com/

6. Bumble | Date, Chat, Meet New People & Network Better [Електронний ресурс]. — https://bumble.com/

7. OkCupid Dating: Date [Електронний ресурс]. — Singles https://www.okcupid.com/

8. Docker Inc. Docker Documentation URL: https://docs.docker.com/.

9. Docker Inc. Docker Compose Overview URL: https://docs.docker.com/compose/.

10. Що таке бази даних? URL: university.sigma.software/what-is-database/

11. PostgreSQL Global Development Group. PostgreSQL documentation URL: https://www.postgresql.org/docs/.

12. When to use SQLite URL: https://www.sqlite.org/whentouse.html

13. Microsoft. .NET documentation URL: https://learn.microsoft.com/en-us/dotnet/.

14. Microsoft. Entity Framework Core documentation URL:

https://learn.microsoft.com/en-us/ef/core/.

15. Microsoft. ASP.NET Core SignalR URL: https://learn.microsoft.com/en-us/aspnet/core/signalr/.

16. Nuxt Team. Nuxt 3 Documentation URL: https://nuxt.com/docs.

17. Quasar Framework. Quasar Documentation URL: https://quasar.dev/start/installation.

18. Swiper.js. Swiper API and Demos URL: https://swiperjs.com/.

19. SmartBear. Swagger OpenAPI Documentation URL: https://swagger.io/docs/.

20. Войцеховська О. В., Черняк О. І. Шаблони проектування програмного забезпечення : навч. посіб. [Електронний ресурс] / О. В. Войцеховська, О. І. Черняк. — Вінниця : ВНТУ, 2015. — 100 с. (4,5 авт. арк. / 3 авт. арк.). — Режим доступу: https://iq.vntu.edu.ua/card.php?id=5106.

21. Азаров О. Д., Богомолов С. В., Швець С. І. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спец. 123 «Комп’ютерна інженерія». — Вінниця : ВНТУ, 2020. — 48 с.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О.Д..
10.03.2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання бакалаврської кваліфікаційної роботи
«Інформаційна система онлайн-сервісу пошуку знайомств»
08-54.БКР.012.00.000.ТЗ

Науковий керівник: доцент к.пед.н.
_____ Добровольська Н. В.
Студент групи 1КІ-21б
_____ Сухина Є. В.

ВНТУ 2025

1 Підстави для виконання бакалаврської кваліфікаційної роботи (БКР)

1.1 Актуальність дослідження

Актуальність дослідження обумовлена потребою у створенні ефективних цифрових рішень для організації онлайн-комунікації та знайомств, що зумовлено зростанням попиту на зручні, безпечні й доступні засоби соціальної взаємодії в умовах цифровізації суспільства. Зокрема, важливість дослідження визначається необхідністю розробки сервісу, який відповідатиме сучасним запитам користувачів і забезпечуватиме якісний досвід взаємодії.

1.2 Наказ про затвердження теми БКР

Тема бакалаврської кваліфікаційної роботи затверджена наказом ректора університету № 97 від 21.03.2025 року.

2 Мета БКР і призначення розробки

2.1 Мета роботи

Метою роботи є створення сучасної інформаційної системи онлайн-сервісу пошуку знайомств із використанням Nuxt.js для клієнтської частини та ASP.NET для серверної логіки.

2.2 Призначення розробки

Призначення розробки полягає у створенні інформаційної системи онлайн-сервісу знайомств, яка забезпечує зручну, безпечну та ефективну взаємодію користувачів для пошуку соціальних контактів.

3 Вихідні дані для виконання БКР

3.1. Базові знання з розробки веб-застосунків у клієнт-серверній архітектурі;

3.2. Розуміння принципів взаємодії frontend та backend частин;

3.3. Ознайомлення з можливостями фреймворків ASP.NET і Nuxt;

3.4. Навички проєктування баз даних і створення REST API;

3.5. Знання сучасних підходів до розробки UI/UX інтерфейсів;

3.6. Основи захисту даних, аутентифікації та авторизації користувачів;

3.7. Досвід використання інструментів для розробки й тестування

веб-застосунків.

4 Вимоги до виконання БКР

4.1 Провести аналіз функціональних потреб користувачів онлайн-сервісу знайомств;

4.2 Розробити архітектуру інформаційної системи з урахуванням вимог зручності, безпеки та ефективності;

4.3. Створити серверну та клієнтську частини системи;

4.4 Реалізувати функціонал реєстрації, авторизації, перегляду профілів, пошуку за фільтрами та обміну повідомленнями;

4.5 Забезпечити збереження та обробку даних за допомогою бази даних;

4.6. Протестувати працездатність системи та підготувати пояснювальну записку.

5 Етапи БКР та очікувані результати

А.1. Таблиця А.1 — Етапи БКР

№ з/п	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз задачі	21.03.25	23.03.25	Аналітичний огляд джерел, задачі досліджень,
2	Огляд аналогічних сервісів, аналіз функціональних вимог	23.03.25	27.03.25	Розділ 1
3	Вивчення предметної області та наповнення веб застосунку	28.03.25	04.04.25	Розділ 2
4	Розробка та програмна реалізація для системи	05.04.25	12.04.25	Розділ 3
5	Оформлення пояснювальної записки, графічного матеріалу і презентації	12.04.25	10.05.25	ПЗ, графіч. матеріал і презентація
6	Підготовка супроводжуючих документів, проходження нормоконтролю та тесту на плагіат	10.05.25	13.05.25	Оформлені документи

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, графічні та ілюстративні матеріали, протокол попереднього захисту на кафедрі, відгуки наукового керівника та опонента, протоколи складання державних екзаменів, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БКР

8.1 При оформлюванні БКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21.

ДОДАТОК Б

Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень

Назва роботи: Інформаційна система онлайн-сервісу пошуку знайомств

Тип роботи: бакалаврська кваліфікаційна дипломна робота
(БДР, МКР)

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет)

Показники звіту подібності StrikePlagiarism

Оригінальність 98,5% Схожість _____

Аналіз звіту подібності (відмітити потрібне):

✓ Запозичення, виявлення у роботі, оформлені коректно і не містять ознак плагіату.

☐Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.

☐Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Захарченко С.М.
(підпис) (прізвище, ініціали)

Ознайомлені з повним звітом подібності, який був згенерований системою StrikePlagiarism щодо роботи.

Автор роботи _____ Сухина Є. В.
(підпис) (прізвище, ініціали)

Керівник роботи _____ Добровольська Н. В.
(підпис) (прізвище, ініціали)

ДОДАТОК В

Графічна частина

ІНФОРМАЦІЙНА СИСТЕМА ОНЛАЙН-СЕРВІСУ ПОШУКУ ЗНАЙОМСТВ

СХЕМА БАЗИ ДАНИХ ТА UML-ДІАГРАМА КЛАСІВ

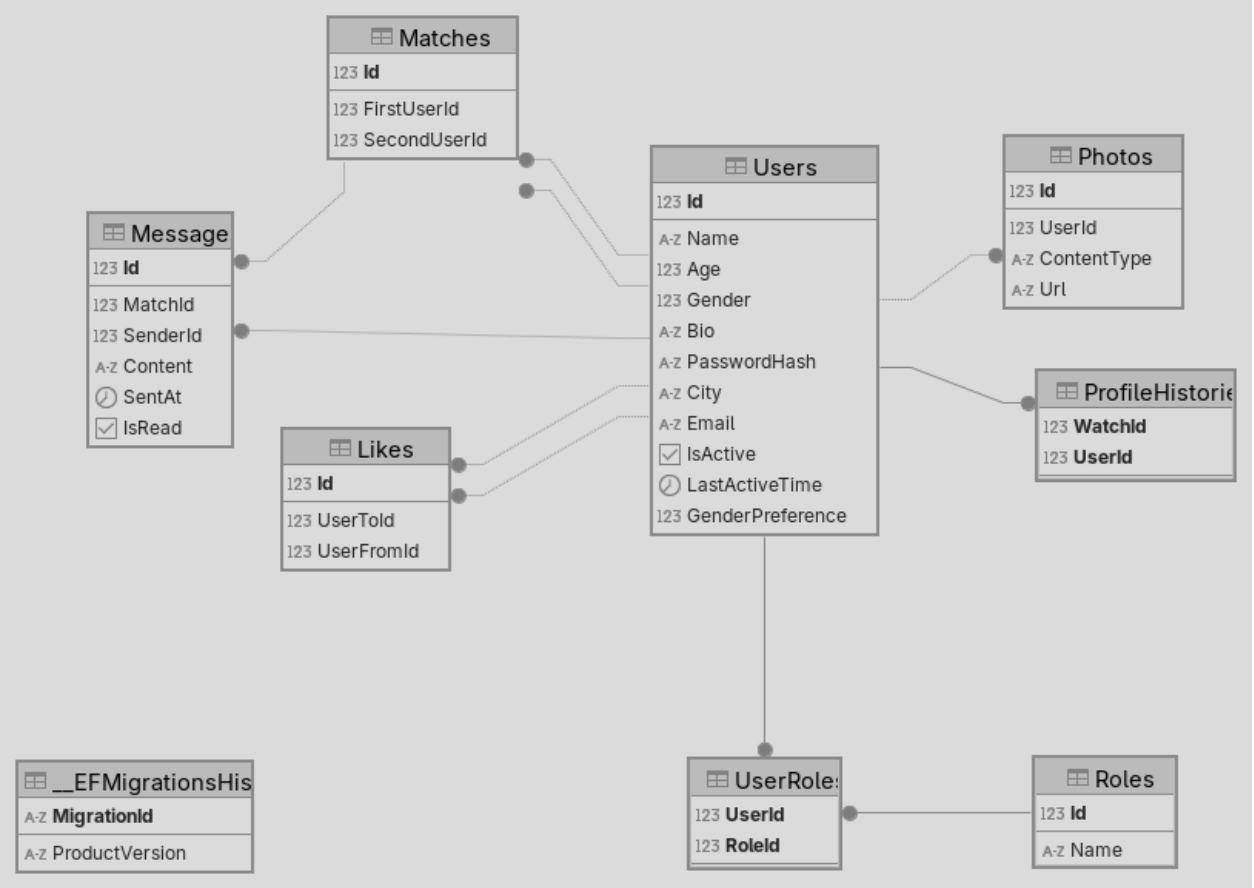

Рисунок В.1 — Схема бази даних

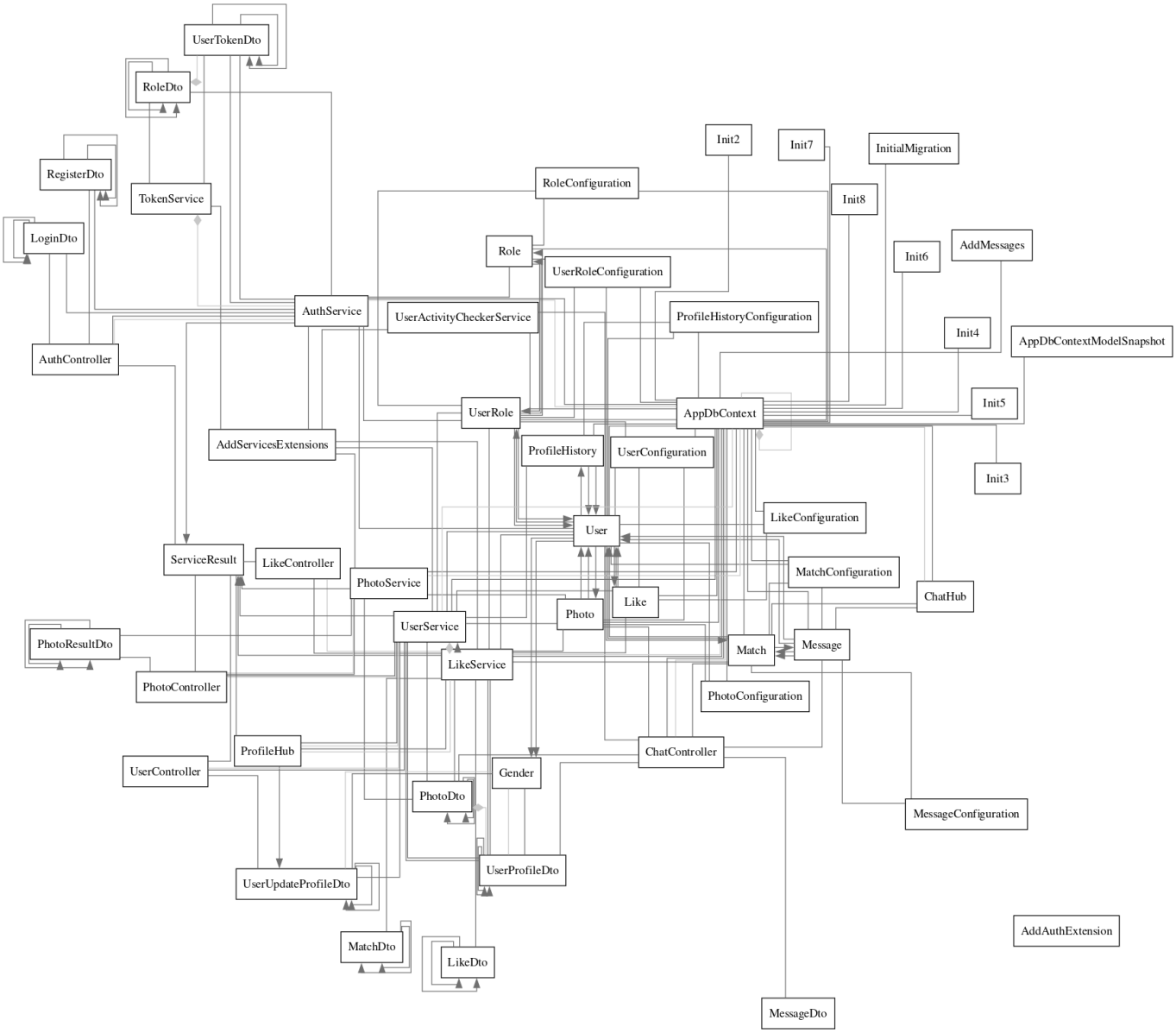

Рисунок В.2 — UML-діаграма класів

ДОДАТОК Г

Ілюстративна частина

ІНФОРМАЦІЙНА СИСТЕМА ОНЛАЙН-СЕРВІСУ ПОШУКУ ЗНАЙОМСТВ

ЗОВНІШНІЙ ВИГЛЯД ОСНОВНИХ СТОРІНОК

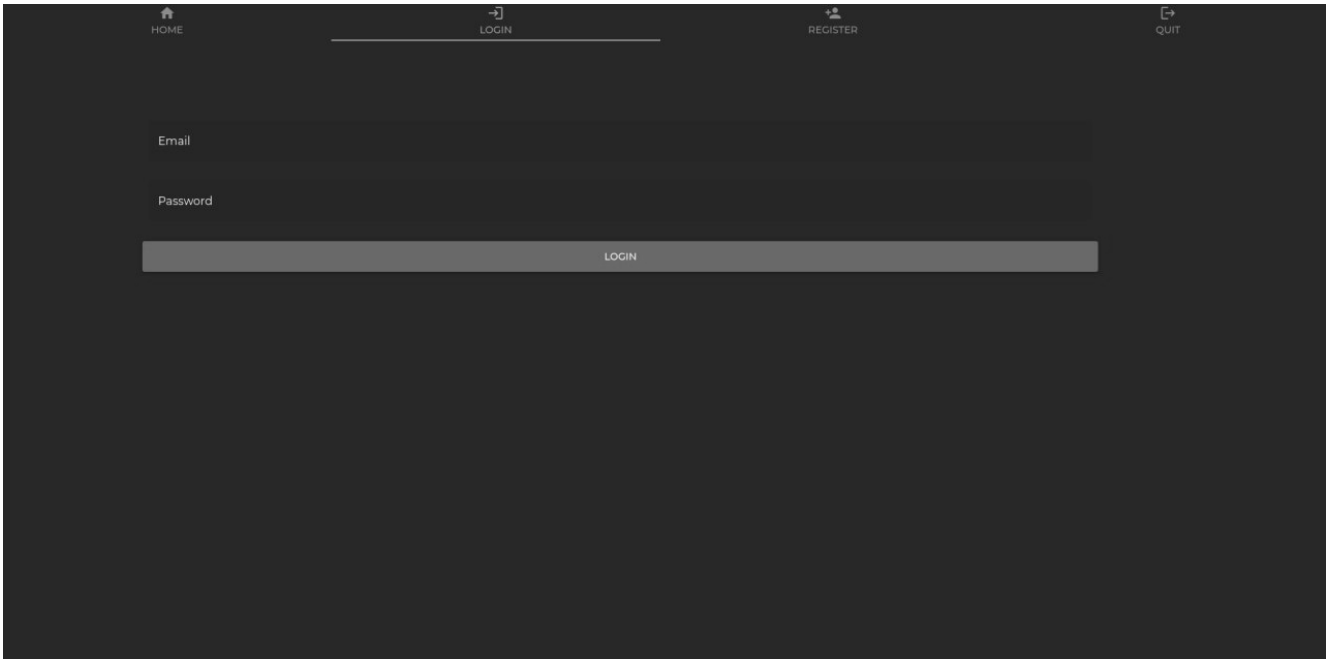

Рисунок Г.1 — Сторінка Входу

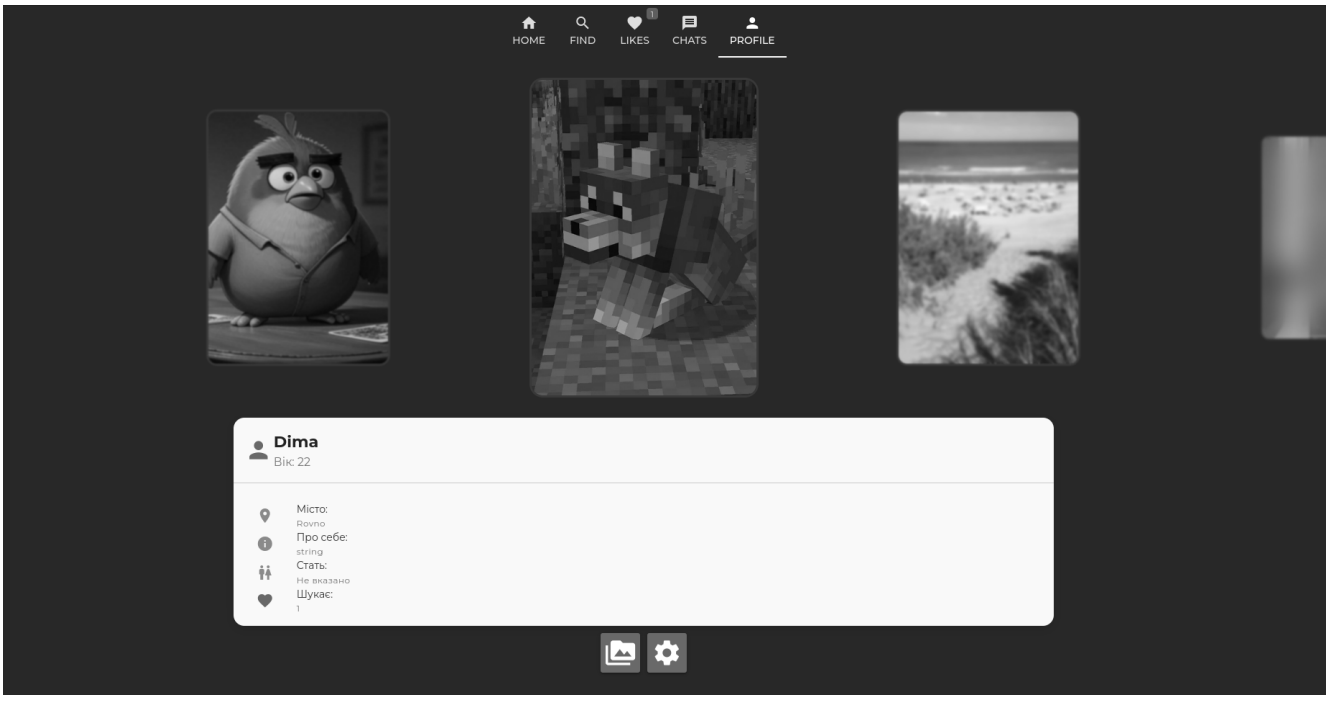

Рисунок Г.2 — Сторінка профілю користувача

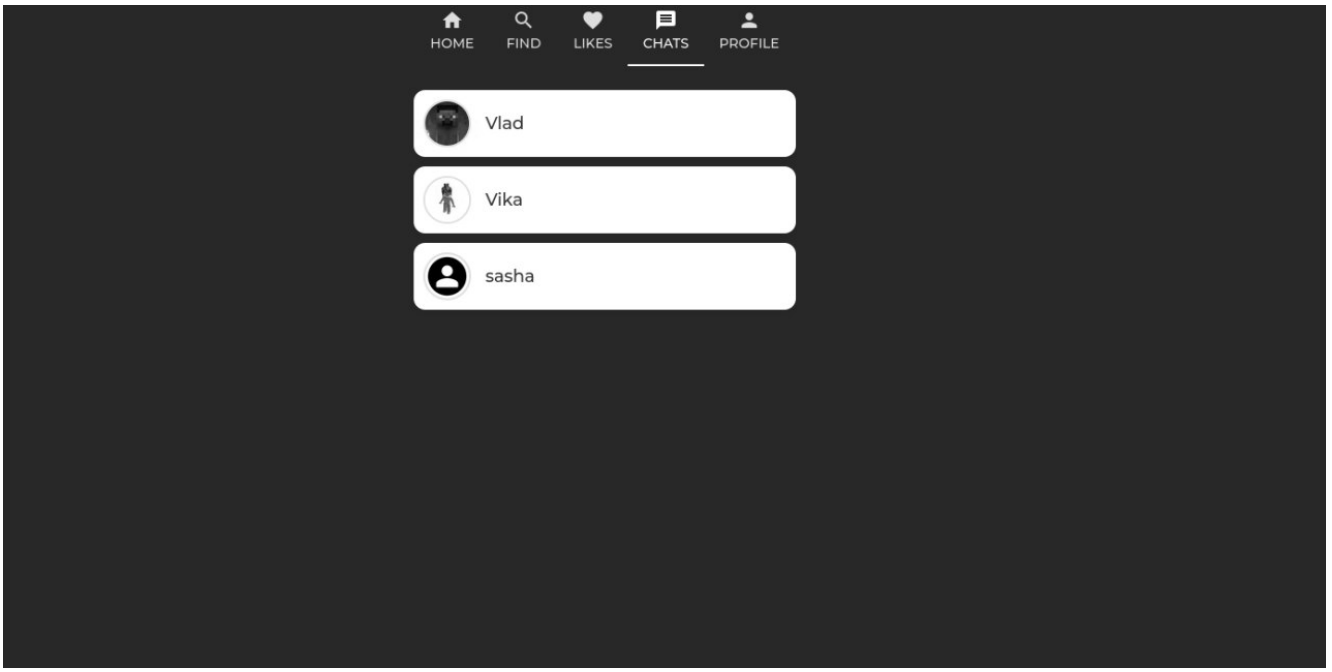

Рисунок Г.3 — Сторінка з чатами користувача

Рисунок Г.4 — Сторінка з окремим вікном чату

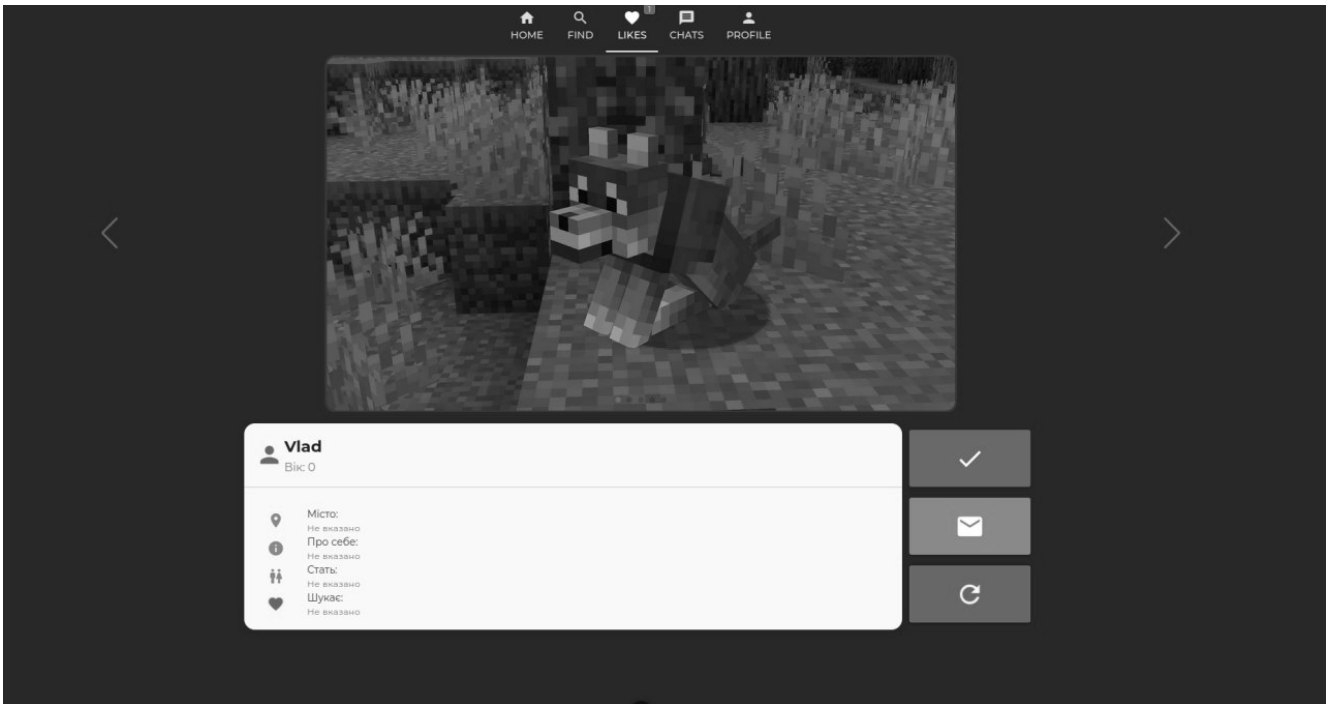

Рисунок Г.5 — Сторінка з лайками користувача

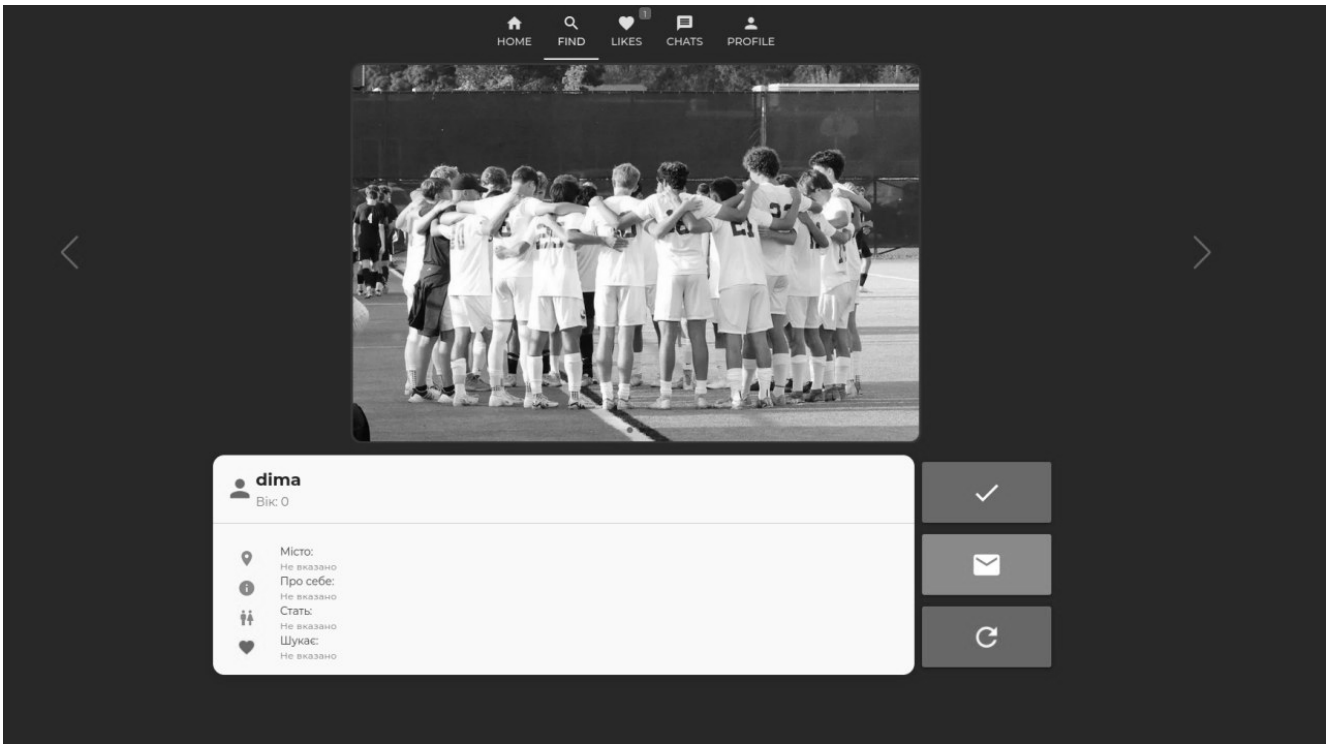

Рисунок Г.6 — Сторінка пошуку

ДОДАТОК Д

Лістинг програмного забезпечення

Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services
        .AddServices()
        .AddAuth();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddControllers();
builder.Services.AddSignalR();
builder.Services.AddSwaggerGen(options =>
{       options.SwaggerDoc("v1", new OpenApiInfo { Title = "Some API v1", Version = "v1" });
        options.AddSignalRSwaggerGen();});
builder.Services.AddDbContext<AppDbContext>(options =>
        options.UseNpgsql(builder.Configuration.GetConnectionString("PostgresConnection")));
builder.Services.AddCors(options =>{
        options.AddPolicy("AllowFrontend", policy =>
        policy.WithOrigins("https://localhost:3000")
        .AllowAnyHeader()
        .AllowAnyMethod()
        .AllowCredentials());});
var app = builder.Build();
app.UseCors("AllowFrontend");
app.UseStaticFiles();
app.UseRouting();
app.MapControllers();
app.UseAuthentication();
app.UseAuthorization();
if (app.Environment.IsDevelopment()){
        app.UseSwagger();
        app.UseSwaggerUI();}
app.MapHub<ChatHub>("/chat");
app.Run();
```

AppDbContext

```
public class AppDbContext : DbContext
{       public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
        public DbSet<User> Users { get; set; }
        public DbSet<Like> Likes { get; set; }
        public DbSet<Role> Roles { get; set; }
        public DbSet<UserRole> UserRoles { get; set; }
        public DbSet<Match> Matches { get; set; }
        public DbSet<ProfileHistory> ProfileHistories { get; set; }
        public DbSet<Photo> Photos { get; set; }
        public DbSet<Message> Messages { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
        base.OnModelCreating(modelBuilder);
        modelBuilder.ApplyConfiguration(new UserConfiguration());
        modelBuilder.ApplyConfiguration(new LikeConfiguration());
        modelBuilder.ApplyConfiguration(new RoleConfiguration());
        modelBuilder.ApplyConfiguration(new UserRoleConfiguration());
        modelBuilder.ApplyConfiguration(new PhotoConfiguration());
        modelBuilder.ApplyConfiguration(new ProfileHistoryConfiguration());
        modelBuilder.ApplyConfiguration(new MatchConfiguration());          }}
```

UserService

```
public class UserService
{       private readonly AppDbContext _context;
        public UserService(AppDbContext context)
```

```
        {           _context = context;           }
        private  async Task<User?> GetCurrentUser(int userId)           {
        return await _context.Users
        .Include(u => u.Roles)
        .ThenInclude(ur => ur.Role)
        .Include(u=>u.Photos)
        .Include(u=>u.ReceiveLikes)
        .ThenInclude(u=>u.UserFrom)
        .FirstOrDefaultAsync(u=> u.Id == userId);   }
        private  UserProfileDto GetUserDto(User user)          {
        return new UserProfileDto(
        user.Id
        , user.Name
        , user.Age
        , user.City
        , user.Bio
        , user.Gender
        , user.GenderPreference
        , user.Photos.Select(r => new PhotoDto(r.Id, r.Url, r.ContentType)).ToList()        );                  }
        private async Task<bool> ProfileIsFull(int userId)      {
        var user = await GetCurrentUser(userId);
        if (user is { Photos: not null }
        && !user.City.IsNullOrEmpty()
        && user.Age > 6)            {
        return true;          }
        return false;         }
        public async Task<ServiceResult<UserProfileDto>> GetUserInfo(int idUser)      {
        var currentUser = await GetCurrentUser(idUser);
        if (currentUser == null)      {
        return new ServiceResult<UserProfileDto>()            {
        IsSuccess = false, Message = "No user found"           };          }
        var currentUserDto = GetUserDto(currentUser);
        return new ServiceResult<UserProfileDto>()            {
        IsSuccess = true
        , Message = "ok"
        , Data =  new UserProfileDto(
        currentUser.Id
         , currentUser.Name
        , currentUser.Age
        , currentUser.City
        , currentUser.Bio
        , currentUser.Gender
        , currentUser.GenderPreference
         , currentUser.Photos.Select(r => new PhotoDto(r.Id, r.Url, r.ContentType)).ToList()          )          }          }
        public async Task<ServiceResult<UserProfileDto>> UpdateMyProfile(int userId, [FromBody]
UserUpdateProfileDto newProfileDto)         {
        var currentUser = await GetCurrentUser(userId);
        var badResult = new ServiceResult<UserProfileDto>              {
        IsSuccess = false              };
        if (currentUser == null)      {
        badResult.Message = "User not found";
        return badResult;             }
        if(newProfileDto.Age is < 10 or > 80)          {
        badResult.Message = "Age is not valid";
        return badResult;             }
        await _context.Users.Where(u=>u.Id == userId)
        .ExecuteUpdateAsync(i => i
        .SetProperty(o=>o.Name, newProfileDto.Name)
        .SetProperty(o=>o.Age, newProfileDto.Age)
        .SetProperty(o=>o.City, newProfileDto.City)
        .SetProperty(o=>o.Bio, newProfileDto.Bio)
        .SetProperty(o=>o.Gender, newProfileDto.Gender)
        .SetProperty(o=>o.GenderPreference, newProfileDto.GenderPreference)
```

```
.SetProperty(o=>o.LastActiveTime, DateTime.UtcNow));
await _context.SaveChangesAsync();
return new ServiceResult<UserProfileDto>                {
IsSuccess = true,
Message = "User updated"            };          }
public async Task<string> DropDatabase()    {
await _context.Users
.Where(u=>u.Age > 0)
.ExecuteDeleteAsync();
return "ok";        }
public async Task<ServiceResult<string>> SetProfileActive(int userId) {
var user = await _context.Users.FirstOrDefaultAsync(u => u.Id == userId);
if (user == null)
return new ServiceResult<string>                {
IsSuccess = false, Message = "Profile not found"        };
if (user.IsActive)
return new ServiceResult<string>                {
IsSuccess = true, Message = "Profile is active"         };
user.IsActive = true;
await _context.SaveChangesAsync();
return new ServiceResult<string>                {
IsSuccess = true, Message = "Profile is active"         };          }
public async Task SetActiveAll()    {
await _context.Users.ExecuteUpdateAsync(u => u
.SetProperty(r => r.IsActive, true));
await _context.SaveChangesAsync();          }
public async Task AddToHistory(int userId, int archiveId)       {
var hs = await _context.ProfileHistories
.FirstOrDefaultAsync(h => h.UserId == userId && h.WatchId == archiveId);
if(hs == null)      {
await _context.ProfileHistories.AddAsync(new ProfileHistory()           {
UserId = userId,
WatchId = archiveId         });
await _context.SaveChangesAsync();          }
await SetProfileActive(userId);      }
public async Task<ServiceResult<List<UserProfileDto>>> Finding(int userId)   {
var currentUser = await _context.Users
.Include(u => u.Photos)
.Include(u=>u.ReceiveLikes)
.Include(u=>u.Roles)
.ThenInclude(ur => ur.Role)
.Include(u=>u.ProfileHistory)
.FirstOrDefaultAsync(u => u.Id == userId);
if (currentUser == null)    {
return new ServiceResult<List<UserProfileDto>>                {
IsSuccess = false, Message = "User not found"           };          }
var activeUser = await GetActivityUsers(currentUser);
if (activeUser.Count >= 1)
return Responce();
await ClearHistory(userId);
activeUser = await GetActivityUsers(currentUser);
if (activeUser.Count <= 1)              {
return new ServiceResult<List<UserProfileDto>>                {
IsSuccess = false, Message = "Users not found"          };          }
return Responce();
ServiceResult<List<UserProfileDto>> Responce()    {
return new ServiceResult<List<UserProfileDto>>                {
IsSuccess = true,
Message = "Users found",
Data = activeUser.Select(u => new UserProfileDto(
            u.Id
            //, u.IsActive
            , u.Name
```

```
                , u.Age
                , u.City
                , u.Bio
                , u.Gender
                ,u.GenderPreference
                , u.Photos.Select(r => new PhotoDto(r.Id, r.Url, r.ContentType)).ToList())).ToList()};        }        }
        private async Task<List<User>> GetActivityUsers(User currentUser)    {
        var viewedUserIds = currentUser.ProfileHistory
        .Select(p => p.WatchId).ToList();
        return await _context.Users
        .Where(u => u.IsActive && !viewedUserIds
        .Contains(u.Id) && u.Id != currentUser.Id)
        .Include(u => u.Photos)
        .OrderByDescending(u => u.LastActiveTime)
        .Take(5)
        .ToListAsync();   }
        private async Task ClearHistory(int userId)   {
        var user = await _context.ProfileHistories
        .Where(u => u.UserId == userId)
        .ExecuteDeleteAsync();
        await _context.SaveChangesAsync();          }}
```

PhotoService

```
public class PhotoService{
        private readonly AppDbContext _context;
        public PhotoService(AppDbContext context)            {
        _context = context;          }
        public async Task<ServiceResult<List<string>>> UpdateMyProfilePhotos(int userId, List<IFormFile> files{
        await DeletePhotos(userId);
        string basePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", "photos");
        var userPhotos = new List<Photo>();
        foreach (var file in files)    {
        using var md5 = System.Security.Cryptography.MD5.Create();
        using var stream1 = file.OpenReadStream();
        byte[] hash = await md5.ComputeHashAsync(stream1);
        var level0 = Guid.NewGuid();
        int level1 = hash[0] % 256 % 10;
        int level2 = hash[1] % 256 % 10;
        int level3 = hash[2] % 256 % 10;
        string nestedDirectory = Path.Combine(basePath
        , level1.ToString()
        , level2.ToString()
        , level3.ToString());
        if (!Directory.Exists(nestedDirectory))        {
        Directory.CreateDirectory(nestedDirectory);            }
        string fileName = $"{level0.ToString()
                        +level3.ToString()
                        +level2.ToString()
                        +level1.ToString()}{Path.GetExtension(file.FileName)}";
        string filePath = Path.Combine(nestedDirectory, fileName);
        await using var stream = new FileStream(filePath, FileMode.Create);
        await file.CopyToAsync(stream);
        string photoUrl = $"/photos/{level1}/{level2}/{level3}/{fileName}";
        userPhotos.Add(new Photo()          {
        Url = photoUrl
        , UserId = userId
        , ContentType = file.ContentType                        });        }
        if (userPhotos.Count < 1)           {
        return new ServiceResult<List<string>>() { IsSuccess = false, Message = "No photos" };            }
        await _context.Photos.AddRangeAsync(userPhotos);
        await _context.SaveChangesAsync();
        return new ServiceResult<List<string>>()    {
        IsSuccess = true
```

```
, Message = "Photos updated"
, Data = userPhotos.Select(u => u.Url).ToList()          };          }
public async Task<ServiceResult<List<PhotoDto>>> GetPhotosUrlByUserId(int userId)  {
var userPhotos = await _context.Photos
.Where(u => u.UserId == userId)
.Select(u => new PhotoDto(u.Id, u.Url, u.ContentType))
.ToListAsync();
if (userPhotos.Count < 1)                {
return new ServiceResult<List<PhotoDto>>() { IsSuccess = false, Message = "No photos" };         }
return new ServiceResult<List<PhotoDto>>() { IsSuccess = true, Message = "Ok", Data = userPhotos };     }
public async Task<ServiceResult<PhotoResultDto>> GetPhotoByUrl(string path)            {
string decodedPath = Uri.UnescapeDataString(path);
string filePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot", decodedPath.TrimStart('/'));
if (!File.Exists(filePath))              {
return new ServiceResult<PhotoResultDto> { IsSuccess = false, Message = "File not found" };      }
byte[] fileBytes = await File.ReadAllBytesAsync(filePath);
string mimeType = GetMimeType(filePath);
return new ServiceResult<PhotoResultDto>()            {
IsSuccess = true,
Data =  new PhotoResultDto(fileBytes, mimeType)    };          }
public async Task<ServiceResult<string>> DeletePhotos(int userId)      {
var photos = await _context.Photos.Where(u => u.UserId == userId).ToListAsync();
foreach (var photo in photos)          {
string filePath = Path.Combine(Directory.GetCurrentDirectory(), "wwwroot",photo.Url.TrimStart('/'));
if (File.Exists(filePath))               {
await _context.Photos.Where(u=>u.Id == photo.Id ).ExecuteDeleteAsync();
File.Delete(filePath);        }          }
await _context.Photos.Where(u => u.UserId == userId).ExecuteDeleteAsync();
await _context.SaveChangesAsync();
return new ServiceResult<string>()             {
IsSuccess = true, Message = "Photos", Data = "OK"              };          }
private string GetMimeType(string filePath)  {
var provider = new FileExtensionContentTypeProvider();
if (!provider.TryGetContentType(filePath, out var contentType))          {
contentType = "application/octet-stream";     }
return contentType;          }}
```

ChatController

```
[Route("/Match")]
[ApiController]
public class ChatController : ControllerBase{
private readonly AppDbContext _context;
public ChatController(AppDbContext context)          {
_context = context;          }
[HttpGet("/dialogs")]
[Authorize]
public async Task<IActionResult> Dialogs(){
var userIdString = User.FindFirstValue(ClaimTypes.NameIdentifier);
if (string.IsNullOrEmpty(userIdString))
return Unauthorized("User ID not found in claims");
var userId = int.Parse(userIdString);
var result = await _context.Matches
.Where(u => u.FirstUserId == userId || u.SecondUserId == userId)
.Select(u => userId == u.FirstUserId ? u.SecondUserId : u.FirstUserId)
.ToListAsync();
if (result.Count == 0)
return Ok(new List<UserProfileDto>());
var usersDialog = await _context.Users
.Include(u => u.Photos)
.Where(u => result.Contains(u.Id))
.Select(u => new UserProfileDto(
u.Id,
u.Name,
```

```
            u.Age,
            u.City,
            u.Bio,
            u.Gender,
            u.GenderPreference,
            u.Photos.Select(r => new PhotoDto(r.Id, r.Url, r.ContentType)).ToList())            )
            .ToListAsync();
            return Ok(usersDialog);     }
            [HttpGet("/dialogs/{partnerId}/history")]
            [Authorize]
            public async Task<IActionResult> GetChatHistory(int partnerId)          {
            var userId = int.Parse(User.FindFirstValue(ClaimTypes.NameIdentifier) ?? throw new
UnauthorizedAccessException("User ID not found"));
            var match = await _context.Matches
            .FirstOrDefaultAsync(m => (m.FirstUserId == userId && m.SecondUserId == partnerId) ||
                                    (m.FirstUserId == partnerId && m.SecondUserId == userId));
            if (match == null)
            return NotFound("Match not found");
            var messages = await _context.Messages
            .Where(m => m.MatchId == match.Id)
            .OrderBy(m => m.SentAt)
            .Select(m => new MessageDto        {
            SenderId = m.SenderId,
            Content = m.Content,
            SentAt = m.SentAt,
            IsRead = m.IsRead           })
            .ToListAsync();
            return Ok(messages);        }
}
```

LikeController

```
[ApiController]
[Route("[controller]")]
public class LikeController : ControllerBase
{
 private readonly LikeService _likeService;
 public LikeController(LikeService likeService)  {
            _likeService = likeService;  }
 [HttpPost("/sendLike")]
 [Authorize]
 public async Task<IActionResult> SendLike([FromBody] int userFollower)  {
            var userSenderId = User.FindFirstValue(ClaimTypes.NameIdentifier);
            if (userSenderId != null)     {
            var result = await _likeService.SendLike(int.Parse(userSenderId), userFollower);
            if (result.IsSuccess)
            return Ok(result.Data);      }
            return BadRequest();  }
 [HttpPost("/sendDislike")]
 [Authorize]
 public async Task<IActionResult> SendDislike([FromBody] int idUserDisliked)  {
            var userSenderId = User.FindFirstValue(ClaimTypes.NameIdentifier);
            if (userSenderId == null)
            return BadRequest();
            await _likeService.SendDislike(int.Parse(userSenderId), idUserDisliked);
            return Ok();  }
 [HttpGet("/mylikes")]
 [Authorize]
 public async Task<IActionResult> GetMyLikes()  {
            var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
            var result = await _likeService.GetLikes(int.Parse(userId));
            if (result.IsSuccess)           {
            return Ok(result.Data);      }
            return Ok(result.Message);  }
```

```
 [HttpPost("/dislikeResponse")]
 [Authorize]
 public async Task<IActionResult> SendDislikeResponse([FromBody] int userDislikedId)  {
          var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
          var result = await _likeService.SendDislikeResponse(int.Parse(userId), userDislikedId);
          if (result.IsSuccess)          {
          return Ok();        }
          return BadRequest(result.Message);  }
 [HttpPost("/likeResponse")]
 [Authorize]
 public async Task<IActionResult> SendLikeResponse([FromBody] int userLikedId)  {
          var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
          var result = await _likeService.SendLikeResponse(int.Parse(userId), userLikedId);
          if (result.IsSuccess)          {
          return Ok();        }
          return BadRequest(result.Message);  }}
```

UserController

```
[ApiController]
[Route("[controller]")]
public class UserController : ControllerBase{
  private readonly UserService _userService;
  public UserController(UserService userService)  {
          _userService = userService;  }
  [HttpGet("/profile")]
  [Authorize]
  public async Task<IActionResult> GetUserProfile()  {
          var userId = int.Parse(User.FindFirstValue(ClaimTypes.NameIdentifier) ?? "0");
          var result = await _userService.GetUserInfo(userId);
          if (result.IsSuccess)          {
          return Ok(result.Data);      }
          return BadRequest(result.Message);  }
  [HttpPost("/setActive")]
  [Authorize]
  public async Task<IActionResult> SetProfileActive()  {
          await _userService.SetActiveAll();
          return Ok();  }
  [HttpGet("/find")]
  public async Task<IActionResult> Find()  {
          var userId = int.Parse(User.FindFirstValue(ClaimTypes.NameIdentifier));
          var result = await _userService.Finding(userId);
          if (result.IsSuccess)          {
          return Ok(result.Data);      }
          return BadRequest(result.Message);  }
  [HttpPut("/updateProfile")]
  [Authorize]
  public async Task<IActionResult> UpdateProfile(UserUpdateProfileDto newProfile)  {
          var userId = int.Parse(User.FindFirstValue(ClaimTypes.NameIdentifier));
          var result = await _userService.UpdateMyProfile(userId, newProfile);
          if (result.IsSuccess)
          return Ok(result.Message);
          return BadRequest(result.Message);  }}
```

finding.vue

```
import type { UserProfile } from '@/data/userProfile';
import ActionButtons from '~/components/Likes/ActionButtons.vue';
import NoLikesComponent from '@/components/Likes/noLikesComponent.vue';
import SwiperComponent from '@/components/SwiperComponent.vue';
import ProfileComponent from '@/components/profile/profileComponent.vue';
import { fetchFindProfiles, fetchSendLike, fetchSendDislike } from '@/services/fetchService';
import { ref, onMounted } from 'vue';
const profile = ref<UserProfile[] | null>([]);
const current = ref(0);
async function fetchLikes() {
```

```
  profile.value = await fetchFindProfiles();
  current.value = 0; }
onMounted(() => {
  fetchLikes();});
function nextProfile() {
  if (profile.value && current.value < profile.value.length - 1) {
            current.value++;
  } else {
            profile.value = null;
            fetchLikes();  }}
async function handleAction(action: string): Promise<void> {
  if (!profile.value || !profile.value[current.value]) return;
  const userId = profile.value[current.value].id;
  if (action === 'approve') {
            await fetchSendLike(userId);
  } else if (action === 'message') {
            console.log('Надіслати повідомлення');
  } else if (action === 'refresh') {
            await fetchSendDislike(userId);  }
  nextProfile();}
</script>
<template>
  <div v-if="profile && profile[current]" class="card">
            <SwiperComponent :photos="profile[current].photos" />
            <div class="panel">
            <ProfileComponent :profile="profile[current]" />
            <ActionButtons @action="handleAction" />
            </div>
  </div>
  <div v-else class="no-profile">
            <q-spinner
            color="primary"
            size="3em"
            />
  </div>
</template>
```

login.vue

```
<template>
  <div v-if="!isAuth" class="loginForm">
            <q-input class="dataInput" standout="bg-blue text-white" label-color="white" v-model="email" type="email"
label="Email" />
            <q-input class="dataInput" standout="bg-blue text-white" label-color="white" v-model="password"
type="password" label="Password" />
            <q-btn @click="login" color="primary" class="loginBtn">Login</q-btn>
  </div>
</template>
<script setup lang="ts">
import { onMounted } from 'vue';
import { fetchLogin } from '@/services/fetchService'
import { checkAuth } from '@/services/fetchService';
let isAuth = ref<boolean>(false);
const email = ref<string>('');
const password = ref<string>('');
const login = async () => {
await fetchLogin(email.value, password.value)
await navigateTo('/profile')
}
onMounted(async ()=>{
  isAuth.value = await checkAuth();
  if(isAuth.value){
            navigateTo('/profile')
  }
```

```
})
</script>
```

profile.vue

```
<script setup lang="ts">
definePageMeta({
  layout: 'active',
})
import { Swiper, SwiperSlide } from 'swiper/vue'
import 'swiper/css'
import 'swiper/css/navigation'
import 'swiper/css/effect-coverflow'
import { onMounted } from 'vue'
import { checkAuth, fetchGetProfile, getBaseUrl } from '~/services/fetchService'
import { Photo } from '@/data/photo'
import { UserProfile } from '@/data/userProfile'
import MainComponent from '@/components/profile/mainComponent.vue'
import NotAuthComponent from '~/components/NotAuthComponent.vue'
const images = ref<string[]>([])
let user = ref<UserProfile | null>(null)
let isAuth = ref<boolean>(false)
onMounted(async () => {
  isAuth.value = await checkAuth()
  if (isAuth.value) {
            const profile = await fetchGetProfile()
            if (profile) {
            user.value = profile
            images.value = profile.photos.map((photo) => getBaseUrl() + photo.url)
            }  }})
function getLength(): number {
  if (user?.value?.photos?.length && user.value.photos.length < 5) {
            return user.value.photos.length  }
  return 5}
</script>
<template>
  <div v-if="isAuth">
            <div class="swiper-container">
            <swiper
            :slides-per-view="getLength()"
            :space-between="18"
            :centeredSlides="true"
            :direction="'horizontal'"
            :navigation="false"
            :mousewheel="true"
            :effect="'coverflow'"
            :coverflowEffect="{
            rotate: 0,
            stretch: 0,
            depth: 150,
            modifier: 2.5,
            slideShadows: false,          }"
            class="mySwiper">
            <swiper-slide v-for="(img, index) in images" :key="index">
            <img :src="img" :ratio="16 / 9" class="photo" />
            </swiper-slide>
            </swiper>
            </div>
            <MainComponent :profile="user" />
  </div>
  <div v-else-if="!isAuth">
            <NotAuthComponent />
  </div>
  <div v-else>
```

```
            <q-spinner-hourglass
            color="primary"
            size="4em"
            />
  </div>
</template>
```

ChatComponent.vue

```
<script setup lang="ts">
import type { UserProfile } from '@/data/userProfile';
import { ref } from 'vue';
import { getBaseUrl } from '~/services/fetchService';
defineProps<{
  profile: UserProfile;
  messages: { senderId: string; text: string }[];
  currentUserId: string|undefined;}>();
defineEmits<{
  (e: 'send', message: string): void;
  (e: 'close'): void;}>();
const messageInput = ref('');
</script>
<template>
  <div class="chat-box">
            <div class="chat-header">
            <button class="close-btn" @click="$emit('close')">×</button>
            <img
            class="avatar"
            :src="profile.photos && profile.photos.length > 0 ? getBaseUrl() + profile.photos[0].url : '/default-image.png'"
            alt="User avatar"              />
            <h3>{{ profile.name }}</h3>
                        </div>
            <div class="chat-messages">
            <div v-for="(msg, index) in messages" :key="index" :class="['message', msg.senderId === 'me' || msg.senderId
=== currentUserId ? 'sent' : 'received']">
            <div class="message-content">
            <span>{{ msg.text }}</span>
            <small>{{ msg.senderId === 'me' ? 'Ви' : msg.senderId }}</small>
            </div>
            </div>
            </div>
            <div class="chat-input">
            <input
            v-model="messageInput"
            placeholder="Введіть повідомлення..."
            @keyup.enter="$emit('send', messageInput); messageInput = ""            />
            <button @click="$emit('send', messageInput); messageInput = "">
            <q-icon name="send"></q-icon>
            </button>
            </div>
  </div>
</template>
```