МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ТІТАРЕНКО РУСЛАН АНДРІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 2025 р.

РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ДОДАТКУ ДЛЯ ВЗАЄМОДІЇ КОРИСТУВАЧІВ НА ОСНОВІ СПІЛЬНИХ ІНТЕРЕСІВ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Потапова Н. А., доцент кафедри
інформаційних технологій,
к. е. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____

Вінниця - 2025

АНОТАЦІЯ

Тітаренко Р.А. Розробка серверної частини мобільного додатку для взаємодії користувачів на основі спільних інтересів. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2025.

У кваліфікаційній (бакалаврській) роботі досліджено теоретичні та практичні аспекти розробки серверної частини мобільного додатку для взаємодії користувачів на основі спільних інтересів. Проведено аналіз технологій та інструментів для створення серверної частини, а також функціональних вимог до мобільного додатку. Розроблено структуру серверної частини, механізми взаємодії користувачів та безпеку даних. Оцінено ефективність роботи через тестування.

Ключові слова: серверна частина, мобільний додаток, база даних, соціальні мережі.

52 с., 5 рис., 2 дод., 47 джерел.

ABSTRACT

Titarenko R.A. Development of the server part of the mobile application for user interaction based on common interests. Specialty 122 «Computer Science», educational program "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia 2025.

This qualification (bachelor's) thesis explores the theoretical and practical aspects of developing the server part of a mobile application for user interaction based on common interests. An analysis of technologies and tools for creating the server part, as well as the functional requirements for the mobile application, has been conducted. The structure of the server part, user interaction mechanisms, and data security are developed. The effectiveness of the system is evaluated through testing.

Keywords: server part, mobile application, database, social networks.

52 p., 5 figures, 2 applications, 47 sources.

ЗМІСТ

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій значно зросла роль мобільних застосунків, що забезпечують комунікацію та взаємодію між користувачами. Онлайн-платформи, які сприяють об’єднанню осіб за спільними інтересами, стали важливими інструментами для підтримки зв’язків, формування спільнот та обміну інформацією. З огляду на динаміку розвитку інформаційних технологій, актуальним є створення нових засобів взаємодії, що відповідатимуть сучасним вимогам до зручності, швидкодії, безпеки та масштабованості. Особливого значення при цьому набуває якісна реалізація серверної частини, яка виступає основою функціонування більшості мобільних сервісів.

Актуальність теми дослідження зумовлена зростаючими вимогами до персоналізованої взаємодії користувачів у цифровому середовищі, а також потребою у створенні ефективних програмних рішень, що забезпечуватимуть надійний обмін даними та управління контентом. Попри наявність великої кількості платформ для комунікації, залишається недостатньо дослідженим напрям побудови систем, які адаптуються до користувачів у реальному часі.

Метою кваліфікаційної роботи є розробка серверної частини мобільного додатку, призначеного для організації взаємодії користувачів на основі їхніх спільних інтересів.

Об’єктом дослідження є процес програмної реалізації серверної частини мобільного додатку, що підтримує взаємодію користувачів.

Предметом дослідження є технології, архітектурні підходи та програмні засоби реалізації функціональних і комунікаційних можливостей серверної частини мобільного додатку.

Для досягнення поставленої мети було визначено такі *завдання*:

– проаналізувати особливості функціонування сучасних соціальних платформ;

– виявити та обґрунтувати вимоги до мобільного додатку;

– дослідити сучасні технології та інструменти для створення серверної частини;

– реалізувати структуру серверної частини мобільного додатку;

– забезпечити механізми безпечної взаємодії та обробки запитів;

– здійснити тестування процесів функціонування серверної частини та проаналізувати результати.

Структура роботи містить 3 основних розділи. В першому розділі висвітлюються питання теоретичних засад розвитку інформаційних технологій соціальних мереж. Другий розділ присвячений питанням аналізу технологій та середовища розробки серверної частини мобільного додатку. Третій розділ містить результати програмної реалізації серверної частини додатку.

Теоретичне значення роботи полягає в узагальненні підходів до побудови серверної архітектури з урахуванням принципів ефективної взаємодії користувачів у мобільному середовищі.

Практичне значення полягає у створенні працюючого серверного рішення, яке може бути використане як основа для подальшого розширення функціоналу мобільного додатку або впровадження у схожі програмні продукти.

Таким чином, стрімкий розвиток цифрових технологій зумовлює необхідність у створенні сучасних мобільних рішень, орієнтованих на ефективну взаємодію користувачів. Реалізація надійної серверної частини є ключовим аспектом забезпечення стабільної роботи таких систем та їхньої адаптації до потреб цільової аудиторії.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ РОЗВИТКУ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ СОЦІАЛЬНИХ МЕРЕЖ

1.1 Теоретичні засади розвитку соціальних мереж

Соціальна мережа – це онлайн-платформа, що забезпечує користувачам можливість створення профілю, встановлення контактів, обміну інформацією, взаємодії в межах спільнот за інтересами, а також формування соціального капіталу за допомогою цифрових інструментів. Основною функцією таких мереж є комунікація, проте спектр їх можливостей значно розширився за останні роки – від особистого спілкування до професійного нетворкінгу, реклами та просування контенту [1].

Ідея соціальних мереж бере свій початок у соціологічних теоріях про взаємозв'язки між людьми, соціальні структури та канали комунікації. В цифровому форматі ці концепції почали втілюватися наприкінці XX століття, коли розвиток Інтернету створив передумови для масової інтерактивної взаємодії між користувачами.

Перші прообрази соціальних мереж з'явилися у 1980-х роках у вигляді електронних дошок оголошень, форумів та електронної пошти. Вони дозволяли людям з різних куточків світу залишати повідомлення, обговорювати теми та підтримувати зв'язки, хоча й не мали персоналізованого підходу чи розвиненої візуальної складової [1].

Формальним початком цифрових соціальних мереж вважається запуск у 1995 році платформи Classmates.com, яка дозволяла знаходити та підтримувати контакт з колишніми однокласниками. Незабаром, у 1997 році, з'являється SixDegrees.com, що стала першою платформою, яка поєднала створення особистих профілів, додавання друзів та перегляд соціальних зв'язків між користувачами. Саме ця модель лягла в основу подальших сервісів [2].

Період з 2002 по 2006 роки став етапом активного експериментування. У 2002 році з'явився Friendster, який зробив акцент на онлайн-знайомствах. У 2003

році стартував LinkedIn, орієнтований на професійні зв'язки, а також MySpace, що швидко здобув популярність серед молоді завдяки широким можливостям для персоналізації профілю, додавання музики та ведення блогів.

У 2004 році відбувся ключовий прорив – запуск Facebook, створеного Марком Цукербергом. Спершу доступний лише для студентів Гарварду, сервіс стрімко поширився серед інших університетів, а згодом став відкритим для всіх охочих. Facebook запровадив низку інновацій: стрічку новин, «уподобайки», коментарі, персональні сторінки та можливість створювати групи за інтересами. Ця платформа задала нові стандарти цифрової соціалізації [2].

Подальші роки характеризуються спеціалізацією соціальних мереж. У 2006 році з'являється Twitter, що запровадив формат мікроблогінгу – коротких повідомлень (твітів). У 2010 році запускається Instagram, платформа для обміну фотографіями, що надалі стане популярною завдяки візуальному контенту. У 2016 році велику популярність здобуває TikTok, орієнтований на короткі відео з елементами творчості та розваг.

В Україні розвиток соціальних мереж набрав обертів із поширенням глобальних сервісів, таких як Facebook, Instagram та TikTok. Завдяки зростанню рівня цифрової грамотності, збільшенню доступу до інтернету та інтеграції цифрових технологій у повсякденні справи, соціальні мережі стали невід'ємною частиною життя більшості людей [1].

Невпинний розвиток цифрових технологій та зростання темпів урбанізації суттєво вплинули на характер соціальної взаємодії у сучасному світі. Соціальні мережі стали відповіддю на зростаючу потребу людей в комунікації, приналежності та самовираженні в умовах інформаційного суспільства [3]. Особливість цього феномену полягає не лише в кількісному зростанні користувачів, а й у зміні якості взаємодії між ними. Якщо раніше основним засобом соціалізації виступали офлайн-контакти, то сьогодні віртуальні платформи забезпечують гнучкий та безперервний зв'язок, який не обмежується ні часовими, ні географічними рамками.

Одним із ключових мотивів людей приєднання до соціальних мереж є

бажання залишатися на зв’язку з іншими, підтримувати відчуття соціальної приналежності, знаходити нових друзів або однодумців, демонструвати особисті досягнення, брати участь у дискусіях або розважатися [3]. Вони задовольняють також потребу у самопрезентації та визнанні, дозволяючи користувачам конструювати цифрову ідентичність та керувати нею у зручний спосіб.

Психологічна привабливість цих платформ зумовлена реакцією на такі фундаментальні потреби, як схвалення, емоційна підтримка та соціальне порівняння. Системи сповіщень, уподобайок, коментарів, підписок створюють своєрідну модель оперантного обумовлення – коли користувачі отримують винагороду у вигляді позитивної реакції на свій контент, що стимулює подальшу активність [4].

Водночас, дослідники звертають увагу на потенційні ризики надмірного залучення до соціальних мереж. До них належать зниження рівня живої комунікації, формування залежності від цифрового схвалення, спотворене уявлення про соціальну реальність, а також інформаційне перевантаження [4]. У деяких випадках взаємодія у соцмережах може спричинити зниження самооцінки, особливо серед молоді, яка схильна порівнювати своє життя з ідеалізованими образами інших користувачів [3].

Попри наявні загрози, соціальні мережі залишаються потужним інструментом суспільної мобілізації, інформування, формування громадської думки та самореалізації. Усе частіше вони виступають платформою для просування ініціатив, освіти, бізнесу, громадської активності та навіть психоемоційної підтримки. У цьому контексті важливим завданням сучасного користувача є усвідомлене та критичне використання соціальних мереж – з урахуванням як можливостей, так і викликів, які вони породжують [4].

1.2 Сутність та складові мобільних інформаційних технологій в соціальних комунікаціях

У сучасному цифровому середовищі мобільні інформаційні технології відіграють ключову роль у трансформації способів соціальної взаємодії,

комунікації та обміну даними. Одним із найвиразніших прикладів цієї еволюції є саме мобільні соцмережі, що стали невід'ємною складовою повсякденного життя мільйонів користувачів. Вони забезпечують постійний зв'язок із віртуальним середовищем, надають змогу підтримувати контакт із людьми в режимі реального часу, а також створюють умови для самовираження, обміну новинами та участі у цифрових спільнотах незалежно від фізичної присутності.

Функціонал таких застосунків є надзвичайно широким. Користувачі отримують доступ до інструментів для створення й налаштування особистих профілів, що виступають цифровою візитівкою. Через стрічку дописів відбувається активна динамічна комунікація – можна переглядати нові публікації, реагувати на них, залишати відгуки, поширювати їх. Ці дії не лише забезпечують взаємозв'язок, а й посилюють відчуття залученості до онлайн-спільноти. Важливу роль відіграють push-сповіщення, які оперативно інформують про активність у мережі, створюючи ефект постійної присутності у віртуальному соціумі [5].

Мобільні соцмережі тісно пов'язані з апаратними ресурсами смартфонів – доступ до камери, мікрофона, геолокації чи списку контактів дозволяє оперативно створювати інтерактивний та візуальний контент без виходу за межі додатку. Це стимулює розвиток нових форматів цифрової творчості: історій, трансляцій, коротких відео чи постів, що стали характерною рисою сучасної комунікації. До того ж, внутрішні системи пошуку дають змогу швидко знаходити релевантний контент, користувачів або тематичні групи, що підвищує зручність користування [6].

Не менш значущою є й безпека персональних даних та зручність входу в облікові записи – сучасні сервіси підтримують авторизацію через Google, Apple ID, Microsoft або інші сторонні облікові системи. Вони також пропонують гнучке керування приватністю та доступом. Додаткову цінність становить вбудована аналітика у деякі додатки: автори сторінок, бізнеси та контентмейкери мають змогу оцінювати охоплення публікацій, реакції аудиторії, середній час перегляду тощо – це сприяє вдосконаленню підходів до взаємодії з підписниками [7].

Окрему увагу заслуговує сумісність із різними платформами. Більшість таких сервісів доступні як на Android і iOS, так і через веббраузери, що забезпечує гнучкий доступ з різних пристроїв.

Таким чином, мобільні соціальні застосунки виступають багатофункціональними інструментами цифрової взаємодії, які об’єднують спілкування, персоналізацію, креатив, зворотний зв’язок і конфіденційність у межах єдиної екосистеми. Їхнє використання не лише підтримує соціальні зв’язки, а й дозволяє активно впливати на інформаційне поле, у якому формується цифрова культура.

1.3 Аналіз ринку онлайн-платформ соціальних комунікацій

На сучасному етапі цифрового розвитку платформи для соціальної взаємодії стали ключовими інструментами для комунікації, обміну інформацією та формування спільнот. Найпопулярнішими застосунками, що об'єднують мільйони користувачів по всьому світу, є Instagram, TikTok, Facebook, Reddit, Threads та X (раніше Twitter). Їх аналіз дозволяє глибше зрозуміти очікування аудиторії, технологічні рішення і функціональні підходи, які забезпечують залученість та ефективну взаємодію.

Instagram є візуально орієнтованою платформою, що пропонує простий механізм публікації фото, відео, історій та коротких відео «Reels». Його сильна сторона – візуальна привабливість контенту, можливість швидкого охоплення широкої аудиторії через алгоритмічні рекомендації, а також інтеграція з іншими продуктами компанії Meta. Завдяки великій кількості інструментів для редагування зображень і відео, Instagram залишається актуальним для блогерів, брендів і різних фахівців, що прагнуть візуального самовираження. Окрім стрічки, важливу роль відіграють «Stories» – формат тимчасового контенту, який стимулює щоденну активність. Також платформа активно розвиває комерційний сегмент: інтеграції з магазинами та просування товарів через рекламу. Водночас алгоритмічна стрічка часто знижує органічне охоплення менш популярних

авторів, що може ускладнювати початківцям зростання без рекламних вкладень [6].

TikTok став провідним прикладом застосунку, що надає доступ до короткоформатного відеоконтенту з потужним алгоритмом персоналізації. Його інтерфейс сприяє швидкому споживанню контенту: платформа побудована на нескінченній стрічці відео. Це створює ефект глибокого занурення і сприяє високому рівню залученості. Креативні інструменти для зйомки, монтажу, додавання музики, ефектів і фільтрів дають змогу будь-якому користувачу створювати контент, що має потенціал стати популярним. Особливою рисою TikTok є трендова культура – користувачі активно долучаються до челенджів, мемів і тематичних рубрик, формуючи спільне інформаційне поле. Однак недоліком платформи можна вважати поверхневий характер взаємодії: зосередженість на переглядах та вподобаннях частково витісняє глибшу комунікацію, а надмірна залежність від алгоритмів робить процес зростання аудиторії менш контрольованим [7].

Facebook зберігає позиції на ринку онлайн-комунікації завдяки своєму широкому функціоналу. Це платформа, що об'єднує одразу кілька напрямів взаємодії: від особистих повідомлень і стрічки новин до управління публічними сторінками, маркетинговими кампаніями та бізнес-інструментами. Завдяки цьому Facebook активно використовується організаціями, брендами, медіа та освітніми проєктами. Проте платформа все частіше сприймається як «застаріла» серед молодшої аудиторії, яка віддає перевагу більш динамічним і візуальним середовищам. Крім того, інтерфейс Facebook можна вважати перевантаженим: велика кількість вкладок, функцій і повідомлень створює відчуття інформаційного перенасичення. Не всі інструменти інтуїтивно зрозумілі, що ускладнює перше знайомство з платформою та її активне використання[8].

Reddit вирізняється унікальною структурою побудови спільнот – так званих subreddits, кожен з яких присвячений окремій темі, інтересу або явищу. Це дозволяє користувачам швидко знайти однодумців та приєднатись до тематичних обговорень. Reddit наповнений дискусіями, де велике значення

мають не популярність автора, а якість контенту. Система голосування за публікації та коментарі дозволяє спільноті самостійно формувати порядок денний, піднімаючи найбільш корисні або дотепні дописи нагору. Однак така модель має і недоліки: новим користувачам складно орієнтуватися в структурі та етиці взаємодії, а механізм модерації залежить від конкретної спільноти і може бути як ефективним, так і суб'єктивним, що не завжди гарантує конструктивний діалог [9].

Threads – відносно нова платформа від компанії Meta, яка позиціонується як альтернатива X (колишній Twitter). Основна ідея сервісу – створити більш спокійний і менш токсичний простір для обміну думками, коментарями та короткими оновленнями. Головною перевагою Threads є його тісна інтеграція з Instagram: користувачі можуть швидко створити акаунт на основі наявного профілю, що значно полегшує початок взаємодії. Функціонал Threads ще перебуває на стадії активного розвитку: відсутні деякі звичні інструменти, як-от пошук за хештегами. Крім того, поведінкові моделі спільноти лише формуються, і поки не зовсім зрозуміло, чи зможе платформа запропонувати унікальну цінність, відмінну від існуючих сервісів для текстової комунікації [10].

X (Twitter) продовжує залишатися однією з найпопулярніших платформ для поширення коротких повідомлень у режимі реального часу. Його головна сила – в оперативності: користувачі можуть миттєво реагувати на новини, запускати тренди, поширювати особисті думки та брати участь у глобальних дискусіях. Система хештегів і ретвітів створює ефект «вірусності», дозволяючи темам швидко виходити в топ. Платформа активно використовується журналістами, політиками та представниками творчих індустрій. Проте X має й суттєві недоліки: висока швидкість інформаційного потоку сприяє поверхневому споживанню контенту, що часто знижує якість обговорень. Поширення дезінформації, надмірна поляризація думок і токсичність частини аудиторії залишаються серйозними викликами. Останнім часом також спостерігається зміна вектору розвитку платформи після зміни власника, що призвело до змін у модераційній політиці, алгоритмах стрічки та монетизації контенту, викликавши

неоднозначну реакцію користувачів [11].

Аналіз існуючих платформ для соціальної взаємодії показав, що всі вони активно використовують технології для забезпечення інтерактивності та залучення користувачів. Ключовими аспектами є ефективне використання алгоритмів для персоналізації контенту, а також гнучкість в інтеграції з іншими сервісами та додатками. При розробці серверної частини мобільного додатку для взаємодії користувачів важливо звертати увагу на механізми взаємодії з базою даних, зокрема, забезпечення швидкості та ефективності обробки запитів.

Інтерфейсні аспекти, хоча й не є частиною даної роботи, відіграють важливу роль у взаємодії з сервером та базою даних, оскільки користувачі повинні мати доступ до персоналізованого контенту без затримок. Однак ключовим аспектом розробки серверної частини є саме налаштування архітектури бази даних та ефективне управління взаємодією з користувачами, що дозволяє зберігати та обробляти великі обсяги інформації.

Таким чином, доцільно орієнтуватися на покращення швидкості відповіді сервера на запити користувачів, впровадження оптимальних схем зберігання даних та забезпечення надійної взаємодії з базою даних, що дозволить досягти високої ефективності роботи сервісу для користувачів.

РОЗДІЛ 2
АНАЛІЗ ТЕХНОЛОГІЙ ТА СЕРЕДОВИЩА РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ДОДАТКУ ДЛЯ ВЗАЄМОДІЇ КОРИСТУВАЧІВ НА ОСНОВІ СПІЛЬНИХ ІНТЕРЕСІВ

2.1 Аналіз та розробка функціональних вимог мобільного додатку

Формулювання функціональних вимог до серверної частини мобільного додатку є фундаментальним етапом, що визначає структуру, безпеку та ефективність обміну даними між клієнтом і сервером. Вони повинні бути побудовані з урахуванням цілей застосунку – забезпечення персоналізованої комунікації між користувачами на основі їхніх інтересів – та загальних принципів побудови надійних веб-сервісів. Враховуючи специфіку мобільного додатку, до серверної частини висуваються специфічні функціональні вимоги. Їх розробка повинна забезпечити надійну обробку запитів, захист конфіденційної інформації, безпечну реєстрацію та автентифікацію, а також ефективну підтримку взаємодії між користувачами в режимі реального часу. Нижче наведено основні функціональні вимоги з відповідним технічним обґрунтуванням.

Реєстрація користувача повинна забезпечувати перевірку унікальності ключових ідентифікаційних атрибутів – зокрема, електронної пошти та імені користувача. Такий підхід є критично важливим для забезпечення цілісності системи, уникнення дублювання облікових записів і запобігання конфліктам при ідентифікації користувачів під час автентифікації або обміну даними. Згідно з практиками побудови багатокористувацьких систем, кожен обліковий запис має мати унікальний первинний ідентифікатор, що дозволяє однозначно встановлювати зв’язки між користувачем та його діями в системі [12]. Унікальність таких атрибутів, як email, необхідно реалізовувати на рівні бази даних шляхом встановлення відповідних обмежень, а також на рівні бізнес-логіки під час реєстрації – для забезпечення додаткової перевірки перед записом.

Збереження паролів повинно здійснюватися у вигляді криптографічних

хешів із використанням стійких до атак алгоритмів, таких як BCrypt, Argon2 або PBKDF2. Найбільш популярним серед них є алгоритм BCrypt, який був спеціально розроблений для захисту паролів і включає можливість налаштування складності обчислень, що дозволяє адаптувати рівень захисту до сучасних обчислювальних потужностей. Це робить атаки типу brute-force і dictionary attack обчислювально дорогими [13]. Оскільки паролі – це конфіденційна інформація, їх зберігання у відкритому вигляді суперечить стандартам безпечної розробки та може призвести до масового витоку персональних даних у разі компрометації бази даних. Відповідно до рекомендацій OWASP (Open Web Application Security Project), усі паролі користувачів повинні бути захешовані за допомогою адаптивних хеш-функцій, а також захищені сольовими параметрами для запобігання атакам із використанням попередньо обчислених хешів [14].

Автентифікація користувачів повинна реалізовуватись за допомогою JWT (JSON Web Token) – компактного відкритого стандарту, що дозволяє здійснювати перевірку ідентичності користувача та його прав доступу. Такий підхід передбачає, що після успішної автентифікації сервер генерує підписаний токен, який передається клієнту та включається до кожного наступного запиту. На відміну від традиційної сесійної автентифікації, JWT не потребує збереження сесійної інформації на сервері, що відповідає принципам RESTful-архітектури та сприяє горизонтальному масштабуванню додатку. Крім того, уміст JWT можна розширювати, додаючи до нього додаткові поля, що містять, наприклад, рівень доступу, унікальний ідентифікатор користувача, термін дії тощо [15]. Однак важливо реалізовувати механізми валідації підпису токена і його терміну дії, щоб запобігти несанкціонованому доступу.

Після реалізації автентифікації за допомогою JWT виникає потреба у впровадженні повноцінної авторизації, що регулює доступ до захищених функцій і даних додатку. Кожен запит до серверного API повинен містити дійсний JWT-токен, на основі якого система здійснюватиме перевірку прав доступу. Такий механізм забезпечує контрольований розподіл функціональності

між авторизованими та неавторизованими користувачами, дозволяє обмежити доступ до критичних операцій і персоніфікувати доступ до окремих сервісів або контенту. Авторизація на основі JWT відповідає принципам RESTful-архітектури, оскільки не потребує збереження сесійної інформації на сервері, тим самим сприяє масштабованості та зменшенню навантаження [16].

У процесі розробки функціональної логіки додатку критично важливо забезпечити не лише перевірку дійсності токена, але й впровадження механізмів контролю доступу до конкретних ресурсів. Зокрема, під час виконання потенційно небезпечних або привілейованих дій – таких як редагування, видалення або взаємодія з вмістом (пости, чати, коментарі) – система повинна перевіряти належність запитуваного ресурсу користувачеві, який ініціює операцію. Такий підхід запобігає несанкціонованому доступу до даних інших осіб і дозволяє забезпечити відповідність принципу найменших привілеїв, який передбачає надання користувачеві лише тих прав, що є необхідними для виконання конкретного завдання [17]. Згідно з рекомендаціями в галузі інформаційної безпеки, цей принцип є ключовим для мінімізації ризиків зловживання правами доступу та зменшення площі потенційних атак [18]. Таким чином, ефективна реалізація авторизаційної логіки повинна містити:

- перевірку дійсності й терміну дії JWT-токена;
- валідацію цифрового підпису токена для запобігання його підробці;
- перевірку відповідності користувача конкретному ресурсу перед виконанням чутливих операцій;
- централізовану обробку помилок авторизації та логування неуспішних запитів з метою аудиту безпеки.

Ключову роль у реалізації функціональності додатку відіграє структура бази даних, яка повинна забезпечувати ефективне та надійне зберігання основних сутностей – таких як користувацькі профілі, пости, чати, повідомлення, інтереси та взаємодії. Ці об’єкти є не лише джерелом контенту, але й основою для побудови логіки персоналізованої взаємодії між користувачами. Завдяки

правильному структуруванню та підтримці цих сутностей забезпечується можливість формування віртуального профілю, відображення активностей, підбору контенту на основі інтересів, а також обробки соціальних зв’язків у межах додатку.

Проєктування бази даних має передбачати реляційну модель з чітко визначеними зв’язками між сутностями – наприклад, зв'язок типу «один до багатьох» між користувачем і постами, або «багато до багатьох» між користувачами та їхніми інтересами [19]. Для забезпечення цілісності та уніфікованості доступу до цих даних доцільно використовувати ORM-фреймворк (Object-Relational Mapping), зокрема Entity Framework Core. Його застосування дозволяє абстрагуватись від написання сирих SQL-запитів, забезпечити узгодженість між об'єктною моделлю програми та базою даних, а також спростити управління міграціями при зміні структури. Крім того, ORM забезпечує типізований доступ до інформації, що знижує ймовірність помилок на етапі розробки та сприяє підвищенню безпеки шляхом запобігання SQL-ін’єкціям.

Таким чином, добре спроєктована база даних у поєднанні з сучасним ORM-інструментом є не лише технічною основою додатку, а й запорукою його стабільної та безпечної роботи в умовах багатокористувацької взаємодії.

На основі розробленої структури бази даних необхідно реалізувати функціональність взаємодії користувача з контентом, зокрема можливість створення, редагування та видалення власних постів. Ці дії мають бути доступними лише автентифікованому користувачу, якому належить відповідний запис, що узгоджується з авторизаційною логікою додатку. Наявність повного контролю над власним контентом не лише сприяє активності користувачів і підтримує динамічність стрічки новин, але й відповідає сучасним вимогам щодо захисту персональних даних і конфіденційності.

Для збереження гнучкості архітектури та спрощення підтримки додатку, обробка взаємодій з контентом – таких як додавання вподобань, коментування, а також позначення постів відповідними тематичними тегами – повинна бути

винесена в окремий контролер. Це дозволяє централізовано керувати подібними діями, спрощує повторне використання логіки у різних частинах додатку та полегшує масштабування функціоналу. Зібрані дані про реакції користувачів є цінним джерелом для побудови віртуальних портретів, які формуються на основі переваг, інтересів та активностей у стрічці. Надалі ці портрети можуть використовуватись для реалізації персоналізованого добору контенту, підвищуючи релевантність і залучення користувачів до соціальної взаємодії всередині платформи.

З огляду на комунікаційну спрямованість додатку, користувач повинен мати можливість створювати як особисті, так і групові чати. Такий поділ дозволяє адаптувати форму взаємодії залежно від контексту. Структура чатів повинна враховувати як відправника, так і список учасників, а також тип чату, що дозволяє забезпечити правильну маршрутизацію повідомлень.

Повідомлення мають зберігатися у базі даних разом із метаданими: ідентифікатором чату, автором, часом надсилання, а також статусом прочитання. Така структура не лише дозволить реалізувати відображення історії розмови, але й забезпечить можливість подальшого масштабування – наприклад, додавання реакцій, видалення чи редагування повідомлень. Використання реляційної моделі надасть можливість забезпечити чіткі зв’язки між таблицями, а застосування ORM-фреймворку (зокрема Entity Framework Core) спрощуватиме реалізацію відповідних CRUD-операцій [20].

Для забезпечення ефективного обміну даними між мобільним клієнтом та серверною частиною додатку доцільно використовувати RESTful API. Архітектурний стиль REST дозволяє створювати масштабовані сервіси з чітким розділенням відповідальностей, що є особливо важливим у багатокористувацьких системах. Кожна сутність (наприклад, користувач, пост, повідомлення) повинна мати окремий набір ендпоїнтів, організованих за принципами CRUD, що спрощує як реалізацію логіки, так і її тестування [21].

Обмін даними між клієнтом і сервером доцільно реалізувати у форматі JSON – він є легким, структурованим і легко читається як людиною, так і

машиною. Серверні відповіді повинні включати як дані, так і чітко визначені коди стану HTTP (наприклад, 200 OK, 400 Bad Request, 401 Unauthorized), що дозволяє клієнтській частині однозначно інтерпретувати результат операції та відповідно реагувати у межах користувацького інтерфейсу. Такий підхід підвищує узгодженість і передбачуваність поведінки додатку, а також сприяє зменшенню помилок під час інтеграції [22].

Останньою функціональною вимогою є реалізація ефективного механізму перетворення моделей бази даних у DTO (Data Transfer Objects). Це необхідно для того, щоб передавати на клієнт лише ті дані, які справді потрібні, зберігаючи при цьому інкапсуляцію логіки та структури внутрішніх моделей.

Для автоматизації цього процесу доцільно використовувати бібліотеку AutoMapper, яка дозволяє централізовано керувати трансформацією об’єктів між різними шарами програми. Її застосування значно знижує обсяг повторюваного коду, покращує читабельність і зменшує ймовірність помилок при ручному копіюванні даних. Крім того, AutoMapper підтримує дотримання принципів чистої архітектури, згідно з якими внутрішні моделі повинні бути відокремлені від зовнішніх структур, що використовуються у відповідях API [23].

Згідно з офіційною документацією, AutoMapper є рекомендованим інструментом у .NET-середовищі для реалізації безпечного та масштабованого мапінгу між об’єктами [24].

Отже, сформовані вимоги до серверної частини мобільного додатку ґрунтуються на принципах безпеки, масштабованості та відповідності користувацьким сценаріям. Вони дозволяють реалізувати стабільну інфраструктуру для підтримки персоналізованої комунікації між користувачами, що є основною метою розроблюваного програмного продукту.

2.2 Обґрунтування вибору інструментів для розробки мобільного додатку

Для реалізації серверної частини мобільного додатку було обрано мову програмування C# та фреймворк ASP.NET Core, що є частиною екосистеми

платформи .NET, розробленої корпорацією Microsoft.

C# (C-Sharp) – це сучасна об’єктно-орієнтована мова програмування, яка поєднує в собі високу продуктивність та зручний синтаксис. Вона активно використовується для розробки як настільних, так і веб- та мобільних застосунків, а також має потужну інтеграцію з .NET-інструментарієм, що дозволяє ефективно працювати з базами даних, файлами, мережевими запитами та іншими системними компонентами. C# є строго типізованою мовою, що дозволяє виявляти помилки на етапі компіляції, тим самим підвищуючи надійність програмного забезпечення. Завдяки підтримці сучасних парадигм, таких як LINQ, асинхронного програмування, шаблонів узагальнення та інкапсуляції, C# є оптимальним вибором для серверного програмування [25].

ASP.NET Core – це кросплатформенний, відкритий фреймворк для створення високопродуктивних веб-застосунків, API та мікросервісів. На відміну від попередніх версій ASP.NET, ASP.NET Core повністю переписаний для підтримки роботи на різних операційних системах – Windows, Linux та macOS, що значно розширює можливості його використання. Однією з головних переваг фреймворку є його вбудована підтримка побудови RESTful API, що є ключовим елементом для мобільних додатків, які потребують швидкого, стандартизованого та безпечного обміну даними з сервером [26].

Крім того, ASP.NET Core надає вбудовані засоби для реалізації механізмів автентифікації, авторизації, обробки HTTP-запитів, взаємодії з базами даних через Entity Framework Core, а також масштабування та логування. Його модульна структура дозволяє легко додавати або видаляти компоненти залежно від потреб застосунку, що забезпечує гнучкість у розробці. Також варто зазначити активну підтримку з боку спільноти та регулярні оновлення від Microsoft, що сприяє довготривалій підтримці та безпеці розроблених рішень [27].

Таким чином, поєднання C# та ASP.NET Core забезпечує високу продуктивність, безпеку, зручність розробки та масштабованість серверної

частини мобільного застосунку, що робить цей вибір технічно обґрунтованим і стратегічно доцільним.

З метою оптимізації процесу обробки даних між сервером і клієнтською частиною, а також для забезпечення структурованого представлення інформації, у проєкті буде інтегровано бібліотеку AutoMapper. Цей інструмент виконує функцію автоматичного перетворення між об’єктами різних типів, зокрема між сутностями, що представляють структуру бази даних, та об’єктами передачі даних – DTO (Data Transfer Object) [24]. Такий підхід дозволяє розділити внутрішню логіку зберігання даних від логіки їх представлення в API, що є важливою практикою для побудови безпечних і масштабованих систем.

Для забезпечення безпечної обробки облікових даних користувачів буде інтегрована бібліотека BCrypt.Net, яка реалізує криптографічно стійке хешування паролів. Алгоритм BCrypt враховує фактор обчислювальної складності, що дозволяє адаптувати рівень захисту до сучасних обчислювальних потужностей, значно ускладнюючи можливість brute-force атак [28].

З іншого боку, для обробки JWT-токенів (JSON Web Tokens), що використовуються для аутентифікації та авторизації користувачів, буде застосовано бібліотеку System.IdentityModel.Tokens.Jwt. Ця бібліотека надає можливість генерації, підписування та валідації JWT-токенів, що є популярним механізмом обміну авторизаційною інформацією між сервером і клієнтом у сучасних веб- та мобільних додатках. JWT-токени використовуються для передачі даних про користувача (наприклад, його ідентифікаційних даних та прав доступу) у вигляді безпечних підписаних об'єктів. Це дозволяє серверам ефективно перевіряти автентичність запитів без необхідності зберігати стан сесії на сервері, що є важливим для масштабованості застосунку. Використання JWT гарантує швидку і безпечну аутентифікацію та зручне управління сесіями користувачів у масштабованих системах [29].

Таким чином, сукупність вибраних інструментів відповідає вимогам продуктивності, безпеки, масштабованості та зручності підтримки, що є критично важливими факторами для розробки серверної частини сучасного

мобільного додатку. Використання мови програмування C# та фреймворку ASP.NET Core дозволяє забезпечити високу продуктивність та кросплатформеність, що є важливим для взаємодії з різними операційними системами. Інтеграція бібліотек, таких як BCrypt.Net для захисту паролів та System.IdentityModel.Tokens.Jwt для аутентифікації, підвищує безпеку обробки чутливих даних. Завдяки цим інструментам, розробка мобільного додатку стає не тільки більш ефективною, але й стійкою до потенційних загроз безпеки, що робить систему надійною та зручною в підтримці на всіх етапах її експлуатації.

2.3 Принципи та механізми роботи з даними

Організація зберігання, доступу та обробки даних у мобільному застосунку є ключовим аспектом побудови надійної та масштабованої системи. Вибір відповідного механізму роботи з даними напряму впливає на продуктивність, безпеку, гнучкість архітектури та стабільність роботи всього сервісу.

На етапі проєктування інфраструктури системи необхідно розглянути три основні підходи до зберігання даних: реляційні бази даних (SQL), нереляційні бази даних (NoSQL) та гібридні рішення, які поєднують елементи обох типів.

Реляційні бази даних (РБД) є фундаментальним компонентом сучасних інформаційних систем, зокрема таких, що потребують високої структурованості даних, суворої схеми та підтримки транзакцій. До найпоширеніших представників цього класу належать MySQL, PostgreSQL та Microsoft SQL Server. Ці системи організовують дані у вигляді таблиць, де кожна таблиця складається з рядків і стовпців, що дозволяє ефективно управляти інформацією та забезпечує її цілісність [30].

Однією з ключових переваг РБД є підтримка транзакцій, які гарантують, що всі операції над даними виконуються повністю або не виконуються зовсім, забезпечуючи таким чином консистентність бази даних. Це особливо важливо в системах, де критичною є цілісність даних, наприклад, у соціальних мережах, де існують складні зв’язки між користувачами, постами, чатами та іншими об’єктами [30].

Крім того, РБД забезпечують сувору схему даних, що означає чітке визначення типів даних, обмежень та зв’язків між таблицями. Це сприяє зменшенню ймовірності помилок при введенні та обробці даних, а також полегшує підтримку та масштабування системи [30].

У контексті розробки мобільних застосунків, використання реляційної бази даних дозволяє ефективно організувати зберігання та доступ до структурованих даних, забезпечити їхню цілісність та узгодженість, а також підтримувати складні взаємозв’язки між різними об’єктами системи. Це робить РБД оптимальним вибором для систем, де важлива надійність, масштабованість та ефективність управління даними.

Нереляційні бази даних (NoSQL) представляють собою альтернативу традиційним реляційним системам управління базами даних (СУБД), орієнтовану на зберігання та обробку великих обсягів неструктурованих або напівструктурованих даних. Вони характеризуються гнучкою схемою зберігання, що дозволяє ефективно працювати з динамічно змінюваними структурами даних [31].

Однією з ключових переваг NoSQL є масштабованість. Завдяки горизонтальному масштабуванню, ці бази даних можуть обробляти великі обсяги даних, розподіляючи їх між кількома серверами або вузлами. Це особливо актуально для сучасних веб-додатків та сервісів, які потребують високої доступності та швидкості обробки запитів. Гнучкість NoSQL також проявляється у відсутності жорсткої схеми даних. Це дозволяє розробникам швидко адаптувати структуру бази даних до змін у вимогах бізнесу або користувачів без необхідності значних змін у коді або структурі бази [31].

Проте, варто зазначити, що NoSQL бази даних зазвичай не забезпечують повну підтримку транзакцій за принципами ACID. Замість цього, вони часто дотримуються моделі BASE, що може призвести до тимчасової неузгодженості даних у розподілених системах. Це означає, що в системах, де критичною є жорстка узгодженість даних, використання NoSQL може бути менш ефективним порівняно з реляційними СУБД [31].

Таким чином, вибір між NoSQL та реляційними базами даних залежить від конкретних вимог до системи, зокрема щодо обсягу даних, необхідної гнучкості, масштабованості та рівня узгодженості даних.

Гібридні підходи, що поєднують реляційні (SQL) та нереляційні (NoSQL) бази даних, набувають дедалі більшого поширення в складних інформаційних системах, де необхідно ефективно обробляти різнорідні набори даних. Такий підхід дозволяє використовувати сильні сторони обох типів СУБД: структурованість і транзакційність SQL для критичних бізнес-операцій та гнучкість і масштабованість NoSQL для обробки напівструктурованих або неструктурованих даних.

Однією з ключових переваг гібридної архітектури є можливість оптимального розподілу навантаження. Наприклад, дані, що потребують високої узгодженості, можуть зберігатися в реляційній базі даних, тоді як великі обсяги неструктурованої інформації, як-от журнали подій або мультимедійні файли, – у NoSQL-сховищах. Це дозволяє досягти високої продуктивності та масштабованості системи [32].

Проте впровадження гібридних рішень супроводжується низкою викликів. Зокрема, виникає необхідність забезпечення узгодженості даних між різними типами СУБД, що може ускладнити управління та вимагати додаткових механізмів синхронізації. Крім того, інтеграція різнорідних систем потребує ретельного планування та спеціалізованих знань, що може збільшити складність розробки та супроводу системи [32].

Отже, хоча гібридні підходи відкривають нові можливості для обробки різноманітних даних у складних системах, їх впровадження потребує зваженого підходу, врахування потенційних ризиків та ретельного планування для забезпечення ефективної та надійної роботи системи.

У контексті даного застосунку, який передбачає тісну взаємодію між користувачами, збереження персональних даних, постів, чатів, повідомлень і сувору логіку авторизації, оптимальним рішенням є використання реляційної бази даних MySQL.

MySQL обрано основним інструментом зберігання даних завдяки його продуктивності, стабільності та широкій підтримці у спільноті розробників. Це високопродуктивна система управління базами даних з відкритим кодом, яка підтримує масштабованість, багатопотоковість, а також забезпечує захист даних через вбудовані механізми шифрування та контроль доступу [33]. Серед переваг MySQL варто відзначити:

- підтримку транзакцій, яка є критично важливою для забезпечення узгодженості даних у багатокористувацькому середовищі;
- наявність індексації та оптимізації запитів, що підвищує ефективність доступу до великих обсягів інформації;
- можливість гнучкого масштабування, включаючи реплікацію і кластеризацію;
- сумісність з Entity Framework Core, що дозволяє інтегрувати базу даних у додаток із мінімальними витратами на адаптацію коду.

За даними офіційної документації Microsoft, Entity Framework Core надає повноцінну підтримку MySQL [34]. Це дає змогу використовувати повний спектр ORM-можливостей без порушення логіки взаємодії з базою.

Раціональна організація зберігання та обробки даних відіграє визначальну роль у створенні стабільного, масштабованого та функціонально повноцінного мобільного застосунку. Проведений аналіз варіантів зберігання показав, що саме реляційна модель, представлена MySQL, найкраще відповідає вимогам системи з високим ступенем структурованості даних, складною логікою зв’язків і критичністю до цілісності інформації. Інтеграція MySQL з Entity Framework Core забезпечить не лише ефективну взаємодію між застосунком і базою даних, а й значно спростить розробку та підтримку проєкту завдяки об’єктно-орієнтованому підходу, підтримці міграцій і централізованій конфігурації. Така архітектура гарантує гнучкість, безпеку та технологічну стійкість, що є ключовими передумовами для розвитку повнофункціонального сервісу соціального спрямування.

2.4 Тестування API при функціонуванні мобільного додатку

Ефективне тестування API відіграє ключову роль у забезпеченні стабільності й надійності мобільного додатку, оскільки саме серверна частина відповідає за обробку запитів, автентифікацію, керування даними користувачів та забезпечення цілісності комунікації між клієнтським застосунком і бекендом. Для мобільного додатку, що передбачає побудову динамічної взаємодії користувачів на основі їхніх інтересів, особливого значення набуває верифікація коректності REST-запитів, а також швидке виявлення помилок у логіці API.

З метою підвищення ефективності перевірки функціонування серверної частини мобільного додатку необхідне застосування інструменту Swagger, який одночасно виконує функції документації, інтерактивного середовища тестування й валідатора структурованих запитів.

Swagger забезпечує інтерактивну документацію, що дає змогу безпосередньо виконувати HTTP-запити через браузер. Такий підхід підвищує ефективність тестування, оскільки надає доступ до API без потреби в додаткових інструментах, дозволяє змінювати вхідні дані, спостерігати за результатами та діагностувати помилки в режимі реального часу [35]. Застосування OpenAPI-специфікації сприяє формалізованому опису API. Вся інформація щодо маршрутів, методів (GET, POST, PUT, DELETE), параметрів, типів даних і можливих помилок зберігається в структурованому вигляді. Це значно спрощує обслуговування системи, забезпечує узгодженість між розробниками та знижує ймовірність непорозумінь при реалізації нових функцій [36].

Інструмент Swagger UI надає можливість працювати з API у браузері, що є особливо зручним на етапі розробки та тестування. Через графічний інтерфейс можна виконувати запити, аналізувати відповіді, спостерігати за кодами статусів і валідувати дані. Така функціональність істотно підвищує швидкість виявлення логічних і структурних помилок у серверному коді. Важливим аргументом на користь Swagger слугує автоматичне оновлення документації при внесенні змін у структуру API. Це дозволяє підтримувати документацію в актуальному стані без додаткових витрат часу та ресурсів.

Платформа також демонструє високу сумісність із сучасними бекенд-фреймворками, такими як Express.js, NestJS, Django, Spring Boot тощо. Це відкриває можливості для швидкої інтеграції без необхідності створювати документацію вручну або застосовувати окремі бібліотеки для тестування запитів. Swagger може мати місце у тестуванні таких функціональних модулів мобільного застосунку:

- авторизація та реєстрація користувача;
- оновлення персонального профілю;
- керування переліком інтересів;
- формування персоналізованої стрічки контенту;
- комунікація в межах чатів;
- обробка змін у віртуальному портреті користувача.

Інтеграція Swagger UI розглядається як спосіб об’єднати документацію, тестування та перевірку узгодженості API в межах одного інструменту. Очікується, що такий підхід зменшить ризик логічних помилок під час подальшої інтеграції клієнтської частини, спростить супровід серверного функціоналу та забезпечить стабільність обміну даними на всіх етапах життєвого циклу застосунку.

Таким чином, використання Swagger у процесі тестування API мобільного застосунку обґрунтовується високим рівнем функціональності, сумісністю з популярними серверними технологіями та здатністю забезпечити інтегроване середовище для перевірки, валідації й документування REST-запитів. Поєднання OpenAPI-специфікації та Swagger UI сприяє формалізації архітектури серверної частини, забезпечує узгодженість дій розробників і зменшує ризики технічних і логічних помилок на всіх етапах розробки [35, 36]. Завдяки інтуїтивно зрозумілому графічному інтерфейсу, можливості тестування в реальному часі, а також автоматичному оновленню документації, Swagger виступає ефективним інструментом для підтримки якості та надійності API, що є важливим у контексті забезпечення безперервної взаємодії користувачів на основі спільних інтересів.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОЄКТУ СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ДОДАКТУ ДЛЯ ВЗАЄМОДІЇ КОРИСТУВАЧІВ НА ОСНОВІ СПІЛЬНИХ ІНТЕРЕСІВ

3.1 Структура проєкту серверної частини мобільного додатку

У проєкті застосовано класичну багаторівневу архітектуру на основі патерну Model-View-Controller (MVC), яка є стандартом для ASP.NET Core Web API і забезпечує чітке розділення відповідальностей між обробкою презентаційного рівня, бізнес-логікою та доступом до даних [26].

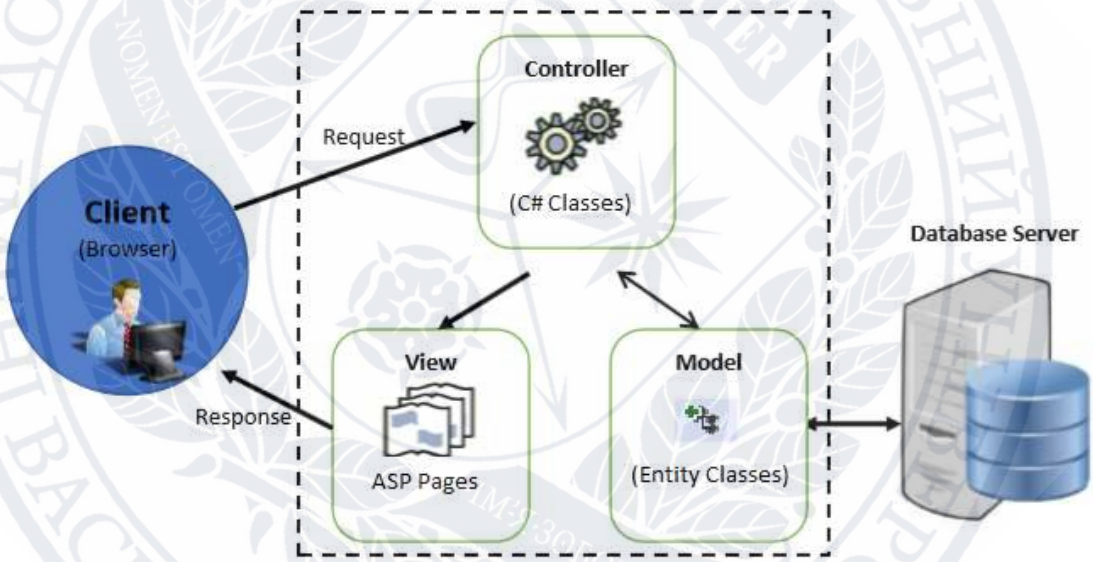

Рисунок 3.1 – Загальна MVC-архітектура ASP.NET Core Web API

Архітектура поділена на три ключові шари. Presentation-шар реалізовано в контролерах, що обробляють HTTP-запити, виконують початкову валідацію через атрибут [ApiController] і передають виклики до бізнес-логіки або безпосередньо до Entity Framework Core контексту [39]. Domain-шар складається з доменних моделей (Models), які відображають структуру таблиць бази даних і містять навігаційні властивості для встановлення зв’язків між сутностями [20]. Infrastructure-шар включає контекст SocialNetworkContext (розміщений у папці

Data) та сервіси (у папці Services), що інкапсулюють бізнес-логіку, наприклад, генерацію та валідацію JWT-токенів через TokenService [15].

Для забезпечення інверсії керування та підвищення тестованості всі компоненти реєструються в DI-контейнері ASP.NET Core у файлі Program.cs [27]. Контекст бази даних додається за допомогою AddDbContext, сервіси – через AddScoped або AddSingleton [27]. Інтерфейсна абстракція для ключових сервісів (наприклад, ITokenService та IMapper) дозволяє у майбутньому замінювати реалізації без зміни споживачів.

Зовнішні бібліотеки та інструменти:

- Entity Framework Core з провайдером Pomelo для MySQL. [37]
- JWT-Bearer для автентифікації та авторизації. [15]
- AutoMapper для проєкції моделей у DTO і зворотного перетворення. [24]
- BCrypt.Net-Next для хешування паролів. [38]
- Swagger (Swashbuckle) для автоматичної генерації документації API. [35]
- Вбудовані механізми ASP.NET Core для логування та валідації через атрибути моделей.

Уся серверна частина зосереджена в каталозі Server:

- Controllers/: основні Web API-контролери (PostsController, UsersController тощо).
- ClientControllers/: контролери для зовнішніх клієнтів (мобільних чи SPA-додатків).
- Data/: містить SocialNetworkContext.cs – точку входу EF Core.
- Models/: доменні моделі для всіх сутностей.
- Services/: класи бізнес-логіки (наприклад, TokenService).
- MappingProfile.cs: налаштування профілів AutoMapper.
- Program.cs та appsettings.json: конфігурація DI, middleware, рядків підключення до БД, ключів JWT і параметрів логування.

Контролери є точкою входу HTTP-запитів у серверну частину додатку: вони обробляють маршрути, виконують початкову валідацію вхідних даних і

передають виклики до бізнес-логіки або безпосередньо до контексту бази даних [26]. Усі контролери успадковані від ControllerBase та позначені атрибутом [ApiController], що автоматично запускає перевірку ModelState і формує уніфіковані HTTP-відповіді за стандартом REST.

UsersController у файлі Controllers/UsersController.cs. Основні ендпоінти:

- GET /api/users – отримати перелік користувачів із короткою або повною інформацією.
- GET /api/users/{id} – отримати деталі конкретного користувача.
- POST /api/users – створити нового користувача (доступно ролі Admin).
- DELETE /api/users/{id} – видалити користувача (доступно ролі Admin).
- GET /api/users/search – пошук за текстовим запитом.
- GET /api/users/random – набір випадкових користувачів.

Методи створення та видалення захищені атрибутом [Authorize(Roles = "Admin")], тоді як решта запитів потребує аутентифікації користувача [18].

PostsController у файлі Controllers/PostsController.cs. Основні ендпоінти:

- GET /api/posts/feed – персональна стрічка постів поточного користувача.
- GET /api/posts/{userId} – перелік постів обраного користувача.
- POST /api/posts – створити новий пост.
- GET /api/posts/{postId}/interactions – отримати взаємодії з постом.
- POST /api/posts/{postId}/interactions – додати взаємодію (наприклад, «лайк»).
- GET /api/posts/search – пошук за вмістом постів.
- GET /api/posts/random – випадкові пости.

Операції створення та взаємодій захищені [Authorize]. Для перетворення доменних моделей у DTO застосовується бібліотека AutoMapper, що налаштовується через профіль мапінгу.

MessagesController та ChatsController:

- MessagesController (Controllers/MessagesController.cs) забезпечує:
  - POST /api/messages/{chatId} – надсилання повідомлення.
  - PATCH /api/messages/{messageId} – редагування повідомлення.

- DELETE /api/messages/{messageId} – видалення повідомлення.

- ChatsController (Controllers/ChatsController.cs) відповідає за:
  - GET /api/chats – перелік чатів користувача.
  - GET /api/chats/{id} – деталі чату та його повідомлення.
  - POST /api/chats/private – створення приватного чату.
  - POST /api/chats/group – створення групового чату.

Обидва контролери захищені атрибутом [Authorize] і отримують доступ до бази даних через DI-впровадження SocialNetworkContext.

ClientControllers

- MeController (ClientControllers/MeController.cs): Ендпоінти для роботи з профілем поточного користувача, зокрема GET /api/me, PATCH /api/me/public, PATCH /api/me/private, DELETE /api/me, а також отримання взаємодій і сповіщень.
- AuthorizationController (ClientControllers/AuthorizationController.cs): Методи реєстрації та входу: POST /api/authorization/register, POST /api/authorization/login, POST /api/authorization/validate-token; незахищені ендпоінти мають [AllowAnonymous], захищені – [Authorize].

Загальні механізми

- Маршрутизація здійснюється через атрибут [Route("api/[controller]")].
- HTTP-методи позначені атрибутами [HttpGet], [HttpPost], [HttpPatch] та [HttpDelete].
- Впровадження залежностей: контролери отримують через конструктор SocialNetworkContext, сервіси (ITokenService, IMapper тощо) та інші залежності.
- Обробка помилок і логування реалізовані через глобальні middleware, налаштовані в Program.cs.

У проекті виділено два ключові шари для роботи з даними: доменні моделі й класи-DTO (Data Transfer Objects). Це дозволяє чітко розділяти внутрішню структуру БД та формат обміну даними з клієнтом, підвищуючи безпеку, продуктивність і підтримуваність коду.

Доменні моделі розташовані в каталозі Models/ і безпосередньо

відображають структуру таблиць бази даних. Кожна сутність містить первинний ключ, властивості даних та навігаційні властивості для встановлення зв’язків між таблицями. Наприклад, клас User включає ідентифікатор, електронну пошту, псевдонім, аватар, опис, повне ім’я, дати створення й оновлення, статус видалення, роль користувача та колекції навігаційних властивостей (пости, повідомлення, підписки тощо). Аналогічно, Post, Message, Chat, GroupChat, GroupMember, PrivateChat, Discussion, Tag, UserTagWeight, UserFollower, Notification, PostAttachment, MessageAttachment і ChatUpdatesSubscriber визначають всі інші необхідні сутності з відповідними зв’язками й обмеженнями на рівні навігаційних властивостей .

DTO-класи використовуються для безпечної та ефективної передачі даних між сервером і клієнтом. Вони містять лише необхідні поля, виключаючи чутливу інформацію (паролі, внутрішні ідентифікатори тощо). Приклад:

- UserPublic – містить ідентифікатор, псевдонім, аватар, опис, повне ім’я (без електронної пошти й ролі).
- UserFull – включає всі поля користувача, окрім пароля (електронна пошта, роль, дати тощо).
- PostDto/PostData – ідентифікатор, заголовок, текст, автор (UserPublic), дата оновлення, вкладення, кількість лайків.
- MessageDto – ідентифікатор, текст повідомлення, відправник (UserPublic), дата створення.

Використання DTO дозволяє зменшити обсяг передаваних даних і захистити внутрішню структуру моделі від клієнтських запитів .

Автоматичне перетворення доменних моделей у DTO і навпаки реалізовано за допомогою AutoMapper. Конфігурація профілів у класах, що успадковують Profile, визначає, які саме поля моделі слід включати або ігнорувати при проекції. Це забезпечує централізовану й зрозумілу логіку мапінгу без дублювання коду в контролерах або сервісах .

Для оптимізації запитів до бази даних застосовуються LINQ-проєкції: безпосереднє формування DTO у запиті дозволяє вибирати лише потрібні

стовпці та зменшувати обсяг переданих даних [40]. Наприклад, при отриманні списку користувачів запит може проєктувати поля прямо в UserPublic, мінімізуючи кількість зайвих даних і покращуючи продуктивність .

При роботі з колекціями (наприклад, повідомленнями в чаті) часто обмежують кількість елементів, що повертаються. Наприклад, вибірка останніх 10 повідомлень упорядковано за датою створення дозволяє зменшити навантаження на мережу й клієнт і підвищити швидкість відгуку.

У проєкті роль основного об’єкта доступу до бази даних виконує клас SocialNetworkContext, що наслідує DbContext з Entity Framework Core. Він був згенерований за допомогою інструменту Scaffold-DbContext, який виконує реверсний інжиніринг існуючої БД і створює відповідні C#-класи сутностей та їхню конфігурацію.

У складі SocialNetworkContext оголошено властивості DbSet<TEntity> для кожної таблиці бази даних: користувачів, постів, повідомлень, чатів, тегів, взаємодій, вкладень, сповіщень тощо (рис. 3.2). Кожен DbSet забезпечує доступ до відповідної колекції сутностей та реалізує CRUD-методи EF Core, що спрощує взаємодію з таблицями без написання SQL-запитів напряму.

Метод OnModelCreating(ModelBuilder modelBuilder) містить налаштування моделі, які відображають обмеження схеми бази даних [40]. За допомогою Fluent API налаштовано:

- зв’язки між сутностями (HasOne, WithMany, HasForeignKey),
- унікальні індекси (наприклад, обмеження на дублікати «учасник – груповий чат»),
- правила каскадного видалення відповідно до бізнес-логіки,
- імена таблиць і стовпців, типи даних та дефолтні значення полів.

Ці конфігурації точно відповідають схемі БД і були згенеровані автоматично.

Стрічка з’єднання зберігається в appsettings.json у секції ConnectionStrings [26]. При налаштуванні сервісу в Program.cs викликається AddDbContext<SocialNetworkContext>(options => options.UseMySql(...)), де вказано рядок з’єднання, параметри таймауту та повторні спроби підключення в

разі відмови. За умовчанням увімкнено відстеження змін (ChangeTracker), але для оптимізації операцій читання великих обсягів даних можна застосувати метод AsNoTracking() у LINQ-запитах [41].

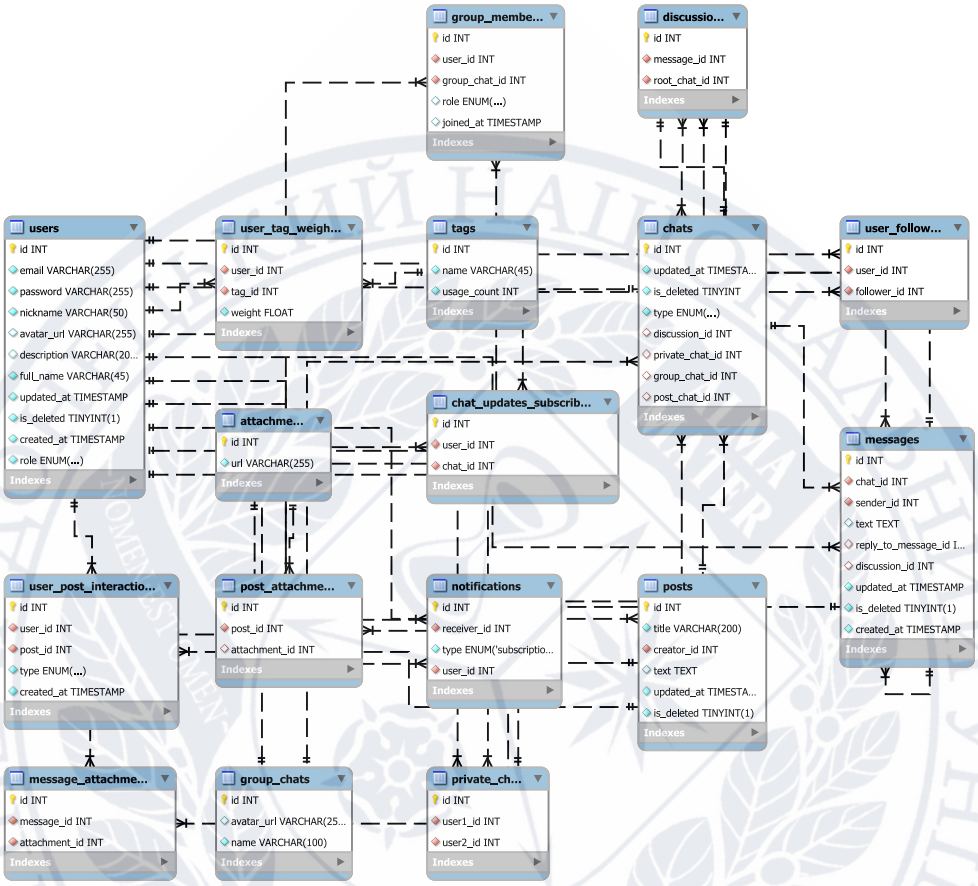

Рисунок 3.2 – ER-діаграма бази даних

На відміну від підходу з підтримкою EF Core Migrations, у цьому проєкті оновлення моделей здійснюється повторним викликом Scaffold-DbContext при змінах у схемі бази даних. Це означає, що всі класи сутностей та конфігурація контексту перегенеровуються вручну з використанням CLI-інструментів EF Core.

Entity Framework Core реалізує патерн Unit of Work через метод SaveChanges(), що агрегує всі внесені зміни в єдину транзакцію [34]. У разі необхідності складних або критичних операцій передбачено явне створення

транзакцій за допомогою Database.BeginTransaction(), що забезпечує атомарність групи запитів і можливість їх відкату.

3.2 Засоби взаємодії користувачів з додатком та безпека даних

Під час реєстрації в API користувачу необхідно надати набір обов'язкових полів, що гарантують коректне створення облікового запису та мінімальний рівень безпеки:

- Email. Використовується як унікальний ідентифікатор в системі та для подальшої ідентифікації під час входу. Перед збереженням перевіряється на наявність в базі, аби уникнути дублювання акаунтів [40].
- Password. Користувацький пароль хешується за допомогою бібліотеки BCrypt.Net-Next перед записом у базу, що забезпечує стійкість до атак у разі компрометації даних [38].
- Nickname. Унікальний псевдонім для відображення в інтерфейсі й пошуку користувача. Перевіряється на унікальність аналогічно до email.
- FullName. Ім'я та прізвище, необхідні для персоналізації облікового запису та відображення у профілі.

Крім цього, підтримуються необов'язкові поля для покращення UX:

- AvatarUrl – посилання на зображення профілю.
- Interests – перелік тегів за інтересами.
- NotInterested – перелік тегів, що виключаються зі стрічки.

Механізми валідації реалізовані на двох рівнях:

1. Атрибути моделей ([Required], [EmailAddress]) та автоматична перевірка ModelState завдяки [ApiController] [39].

2. Додаткова бізнес-логіка в контролері для перевірки унікальності email та псевдоніму.

Для входу в систему користувачі надають свій email та пароль, які перевіряються проти збережених у базі даних записів. Усі контролери, що обробляють приватні операції (створення контенту, перегляд особистих даних тощо), захищені атрибутом [Authorize], а методи реєстрації та входу –

[AllowAnonymous] [42].

- Вхідні дані.

Користувач вводить свій email та пароль. Пароль зберігається в БД у хешованому вигляді за алгоритмом BCrypt, тому під час входу відбувається порівняння введеного пароля з хешем – без зворотного дешифрування [14].

- Логіка обробки запиту.

1. Контролер [HttpPost] /api/authorization/login приймає об’єкт з полями Email і Password.

2. Сервіс аутентифікації (TokenService) виконує пошук користувача за email, перевіряє відповідність пароля та у разі успіху генерує JWT–токен [5].

3. У випадку невірних даних повертається помилка 401 Unauthorized з повідомленням “Invalid email or password” .

- Вхід здійснюється виключно за email і паролем. Це рішення спрощує початкову розробку та UX, але знижує рівень захисту від стороннього доступу та компрометації акаунтів.

У відповідь на успішну реєстрацію або вхід система видає клієнту JWT-токен (JSON Web Token), який використовується для автентифікації під час подальших запитів до API. Генерація токена відбувається за допомогою стандартного механізму JwtSecurityTokenHandler у сервісі TokenService [29].

Поля, включені в payload токена (рис. 3.3).

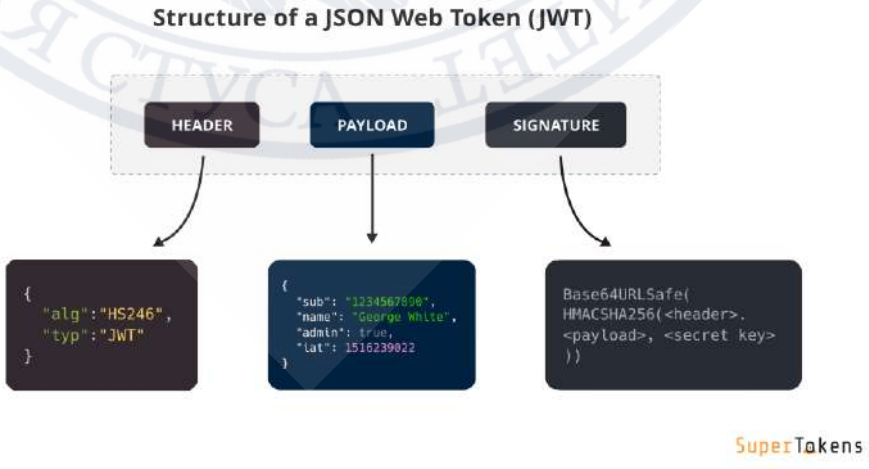

Рисунок 3.3 – Структура токена

Токен містить набір claims – структурованих тверджень про користувача [1, 7]:

- ClaimTypes.NameIdentifier – унікальний ID користувача (User.Id).
- ClaimTypes.Name – псевдонім користувача (nickname), який відображається в UI.
- ClaimTypes.Email – email, використовується для ідентифікації та може бути зручним для внутрішніх перевірок.
- exp – дата закінчення дії токена, додається автоматично при генерації.
- ClaimTypes.Role – використовується для розмежування прав (user/admin).

Обґрунтування: Такий набір claims дозволяє серверу обробляти запити без повторного звернення до бази даних, зберігаючи мінімально необхідну інформацію в токені.

Access-токен проти refresh-токена.

На даному етапі реалізовано лише access-токени (JWT), які клієнт отримує після входу в систему. Refresh-токени не використовуються:

- Access-токен додається до кожного захищеного запиту в заголовку Authorization: Bearer {token} [7].
- Після завершення дії токена користувач має знову пройти аутентифікацію [8].

Обґрунтування: Відмова від refresh-токенів на етапі MVP дозволяє спростити архітектуру, скоротити кількість компонентів та уникнути складностей з їх зберіганням і відкликанням. Проте в майбутньому доцільно реалізувати механізм оновлення access-токенів через refresh-токени, що підвищить безпеку та зручність для користувача.

Строк дії токена.

- Стандартний строк дії: 7 днів – задається як Expires = DateTime.UtcNow.AddDays(7).
- Автоматичне оновлення: не реалізоване. Після завершення строку дії токена користувач має увійти повторно.

Коментар: Довготривалі токени зручні для користувачів, але можуть створити ризики у випадку викрадення токена. З цієї причини рекомендується у продакшені скоротити строк дії access-токена (наприклад, до 15 хвилин) і впровадити refresh-токени.

Після отримання JWT-токена користувач має додавати його до кожного запиту, що вимагає автентифікації. Це дозволяє ідентифікувати користувача без потреби у повторному вході або зберіганні сесій на сервері. Система автоматично перевіряє дійсність токена за допомогою вбудованого механізму авторизації в ASP.NET Core.

JWT-токен надсилається в кожному HTTP-запиті до захищених ресурсів у вигляді заголовка з назвою Authorization, значення якого починається зі слова Bearer, після чого йде сам токен. Наприклад: «Authorization: Bearer <jwt_token>». Цей заголовок автоматично розпізнається middleware системи аутентифікації, яка витягує дані з токена і створює об’єкт користувача, доступний у контролерах через контекст запиту.

У проєкті використовуються атрибути [Authorize] та [AllowAnonymous] для керування доступом:

- [Authorize] застосовується до всіх дій, які стосуються персональних даних користувача або змінюють стан системи. Це, зокрема, створення, редагування та видалення постів, повідомлень, профілю, а також отримання власного профілю, перегляд сповіщень або керування підписками.
- [AllowAnonymous] використовується для дій, які мають бути доступними без входу в систему. До них належать: реєстрація нового користувача, вхід (login), валідація токена, а також (у разі дозволу за політикою проєкту) перегляд публічних профілів або пошук користувачів.

Таке чітке розмежування дозволяє захистити приватні ресурси, не обмежуючи доступ до загальнодоступного функціоналу.

Якщо токен недійсний – наприклад, був підроблений, змінений або просто втратив чинність (протермінований) – сервер повертає статус відповіді 401 Unauthorized. У тілі відповіді зазвичай присутнє повідомлення, що пояснює

причину – наприклад, "Token expired" або "Invalid signature". Така поведінка дозволяє клієнтському додатку правильно реагувати, зокрема перенаправляти користувача на сторінку входу.

Уся ця логіка обробляється автоматично стандартним middleware у складі ASP.NET Core, за умови правильного налаштування параметрів перевірки JWT-токенів.

У головному конфігураційному файлі додатку (звичайно це Program.cs) налаштовано компоненти для автентифікації та авторизації. Зокрема, сервіс автентифікації підключається через метод, який додає підтримку JWT-токенів до системи обробки запитів. Крім того, додається сервіс авторизації, який дозволяє використовувати атрибути [Authorize] у контролерах.

Для логування спроб доступу використовується вбудована система логування ASP.NET Core. Вона дозволяє фіксувати неуспішні запити, помилки токенів, спроби доступу до заборонених ресурсів тощо. За потреби, можна реалізувати власний middleware, який буде зберігати повну інформацію про спроби доступу: час, IP-адресу, тип запиту і результат авторизації. Це особливо важливо для виявлення підозрілої активності або ведення аудиту.

Під час розробки веб-API важливо забезпечити не лише функціональність, а й безпеку обробки та зберігання даних користувачів. У поточній реалізації проєкту передбачено кілька базових заходів захисту, які відповідають рекомендованим практикам для ASP.NET Core та сучасної веб-архітектури.

Реалізовані заходи захисту:

1. Хешування паролів:

- Паролі користувачів не зберігаються у відкритому вигляді. Замість цього вони хешуються за допомогою алгоритму BCrypt, який стійкий до атак перебором і має вбудований механізм соління.
- Під час входу введений пароль порівнюється з хешованим значенням, що зберігається в базі даних, за допомогою функції перевірки.

2. Використання JWT-токенів:

- Після реєстрації або входу користувач отримує JWT-токен, який не

зберігається на сервері, а передається клієнту. Це дозволяє реалізувати безсерверну автентифікацію, де вся необхідна інформація для ідентифікації користувача міститься у самому токені.

- Завдяки цьому система не потребує збереження сесій або стану на сервері, що зменшує потенційні вектори атак.

3. Налаштування CORS:

- У конфігурації проєкту передбачена підтримка політик CORS (Cross-Origin Resource Sharing), які обмежують, з яких саме доменів дозволено надсилати запити до API [43].
- У режимі розробки дозволено всі домени, що спрощує тестування. У продакшені слід обмежити доступ до перевірених фронтенд-доменів для уникнення XSS-ризиків.

4. HTTPS для всіх запитів:

- Проєкт налаштовано на обробку запитів виключно через HTTPS, що гарантує шифрування всіх даних, які передаються між клієнтом і сервером [44].
- Це захищає від атак типу «man-in-the-middle», перехоплення облікових даних і витоку конфіденційної інформації.

5. Валідація введених даних

- Під час реєстрації перевіряється унікальність електронної пошти та псевдоніма (nickname).
- Крім того, перевіряється, щоб поле пароля не було порожнім і відповідало мінімальним вимогам безпеки (це можна додатково посилити через атрибути валідації).

6. Авторизація доступу:

- До всіх ендпоїнтів, що оперують конфіденційною інформацією (наприклад, зміна профілю, створення постів, повідомлень), застосовано атрибут [Authorize].
- Відкриті дії (реєстрація, логін, перегляд публічної інформації) позначаються через [AllowAnonymous], щоб зберегти баланс між захищеністю й

доступністю.

7. Обробка невалідних токенів:

- Якщо запит надійшов із простроченим або некоректним токеном, сервер автоматично повертає код 401 Unauthorized.
- Це дозволяє клієнту адекватно реагувати – наприклад, автоматично перенаправити користувача на сторінку авторизації.

Забезпечення захисту персональних даних користувачів та захищеного доступу до API – критично важливий аспект будь-якого вебзастосунку, особливо соціальної мережі. У проєкті реалізовано низку базових, але ефективних заходів безпеки, що відповідають сучасним практикам.

Паролі користувачів ніколи не зберігаються у відкритому вигляді. Замість цього вони хешуються з використанням стійкого алгоритму BCrypt, який вважається промисловим стандартом для зберігання паролів. Під час реєстрації пароль хешується перед записом у базу даних. Під час входу в систему введений пароль порівнюється з хешем у базі шляхом валідації відповідності.

Система автентифікації базується на JWT (JSON Web Tokens). Користувач отримує токен після успішної реєстрації або входу. Токен містить основні дані (ідентифікатор, псевдонім, email, час дії) та використовується для доступу до захищених частин API. JWT не зберігаються у базі даних – вони зберігаються лише на стороні клієнта, що зменшує ризик компрометації при витоку серверних даних.

У конфігурації API налаштовано CORS (Cross-Origin Resource Sharing) для регулювання, з яких доменів дозволено надсилати запити до API. У режимі розробки дозволено запити з усіх джерел для зручності тестування. У продакшн-режимі рекомендується обмежити доступ лише до дозволених доменів.

Використання захищеного протоколу HTTPS забезпечується стандартними засобами платформи ASP.NET Core. Усі запити передаються лише через зашифроване з’єднання, що запобігає перехопленню чутливих даних (наприклад, паролів або токенів).

При реєстрації та оновленні профілю перевіряється:

- Унікальність email і псевдоніма.
- Формат email.
- Наявність пароля та інших обов’язкових полів.

Контролери та методи, що працюють із приватною інформацією або змінюють стан системи, захищені атрибутом [Authorize]. Для публічного доступу (реєстрація, логін, перегляд публічних профілів) використовується [AllowAnonymous]. Ці атрибути забезпечують розмежування доступу до API на основі наявності та валідності токена.

Якщо токен є недійсним або простроченим, система автоматично повертає HTTP-відповідь з кодом 401 Unauthorized. Це реалізовано за допомогою стандартного middleware авторизації. У таких випадках клієнт має повторно пройти автентифікацію.

3.3 Функціональні частини комунікації користувачів в додатку

У поточній системі передбачено два режими спілкування: приватні чати між двома користувачами та групові чати з довільною кількістю учасників. Приватні розмови зберігаються в окремій сутності PrivateChat, де кожен запис асоціюється виключно з двома користувачами, а групові – через модель GroupChat з додатковою таблицею GroupMembers для списку учасників (рис. 3.4). Така архітектура дозволяє легко вводити нові можливості, наприклад, ролі учасників чи модерацію групових діалогів, без необхідності масштабних змін основної структури даних.

Ініціалізація діалогу здійснюється тільки через спеціальні HTTP-запити до API, а не автоматично при першій надсилці повідомлення. Щоб створити приватний чат, клієнт надсилає POST-запит на кінцеву точку /api/chats/private; для створення групової розмови – POST-запит на /api/chats/group [22]. Перед створенням нового приватного діалогу сервер перевіряє, чи вже існує чат між тими ж самими двома користувачами . Якщо так, замість створення дубліката повертається існуючий запис, що гарантує ідемпотентність операцій створення і запобігає розпорошенню історії повідомлень [45].

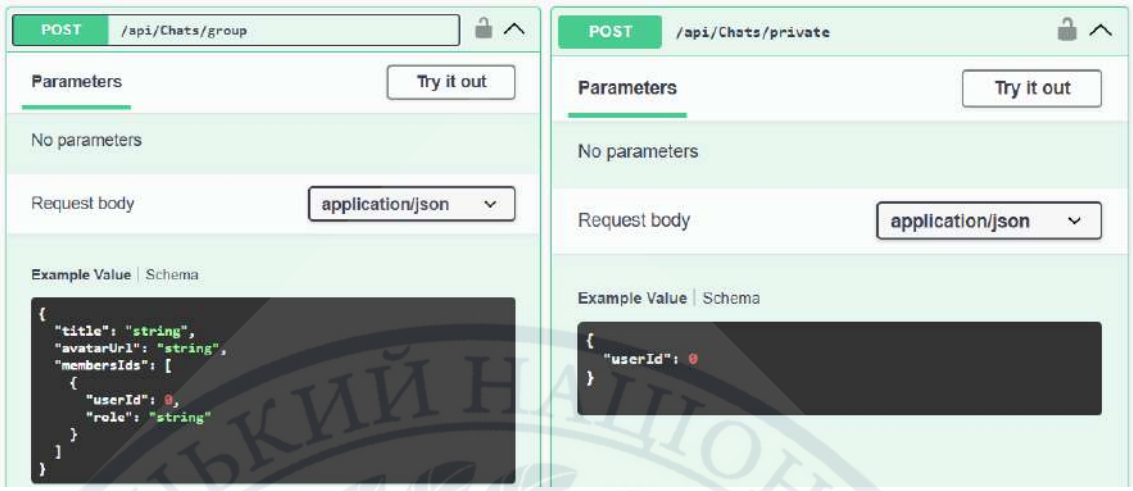

Рисунок 3.4 – Ендпоінти для створення чатів

Загальна модель Chat слугує контейнером для всіх типів розмов і містить стандартні поля: унікальний ідентифікатор, текстове позначення типу чату (наприклад, «приватний», «груповий», «обговорення посту» тощо), мітку часу останньої активності для правильної сортування діалогів і прапорець «м’якого» видалення [17]. Зв’язок із деталізованими даними кожного формату забезпечується через низку зовнішніх ключів, які вказують на відповідні таблиці PrivateChat, GroupChat чи інші розширювані сутності. Така реалізація відповідає принципу єдиного інтерфейсу та дозволяє спростити роботу із загальними операціями над чатами навіть при розширенні їхнього функціоналу.

Видалення діалогів виконується шляхом позначення запису прапорцем «IsDeleted», що дозволяє прибрати чат зі списку активних, не втрачаючи при цьому історію повідомлень у базі. Це рішення поєднує зручність користувача та можливість проведення аудиту чи відновлення розмови в майбутньому. Хоча окрема функція архівації наразі не реалізована, поле «IsDeleted» створює основу для подальшого додавання таких можливостей без змін у базовій структурі [17].

У поточній реалізації кожне повідомлення є окремим записом у таблиці Message і містить текстовий вміст, ідентифікатор відправника, мітку часу створення, а також посилання на чат через поле ChatId. Для підтримки відповіді на конкретне повідомлення передбачено поле ReplyToMessageId, яке вказує на

батьківське повідомлення [19]. Таке рішення дозволяє будувати вкладені дискусії в межах одного чату та зберігати історію в довільному порядку відтворення, а також відповідає загальним рекомендаціям щодо моделювання гілок комунікації в чатах.

Процес відправки повідомлення реалізовано через окремий REST-ендпоїнт, куди клієнт надсилає текст ідентифікатору чату. Перед збереженням система перевіряє, чи має користувач доступ до цього чату, а також чи не позначений чат як видалений. Відповідно до принципів ідемпотентності POST-запитів, повторна відправка того самого повідомлення не повинна створювати дублікати, тому у разі потреби можна ввести опціональне поле-клієнтський ідентифікатор повідомлення, що допоможе уникнути повторних записів при мережевих помилках [45].

Щоб завантажити історію листування, клієнт звертається до ендпоїнта із GET-запитом, вказуючи ідентифікатор чату та параметри пагінації. Використовуються курсорна пагінація з полями limit (максимальна кількість повідомлень за запитом) та cursor (ідентифікатор останнього отриманого елемента). Такий підхід є більш надійним за класичне “offset-limit” і запобігає пропуску або дублюванню повідомлень при високій активності чату [46]. Для кожного запиту система повертає масив повідомлень, відсортованих за часом створення, а також новий курсор для наступної ітерації.

Незважаючи на те, що відмітка “прочитано/непрочитано” наразі не зберігається в базі, у майбутніх версіях передбачено введення окремої сутності MessageReadStatus, яка фіксуватиме ідентифікатор повідомлення, ідентифікатор користувача та час прочитання [3]. Також розглядається можливість використання SignalR для отримання нових повідомлень у реальному часі без необхідності постійного опитування серверу, що значно підвищить відчуття “живої” розмови й відповідатиме сучасним очікуванням користувачів мобільних чат-додатків.

Модель посту сконструйована навколо основних полів: унікального ідентифікатора, заголовка, текстового вмісту, ідентифікатора автора (CreatorId),

мітки часу останнього оновлення та прапорця «м’якого» видалення. Для зберігання медіавмісту використано окрему сутність вкладень (PostAttachments), яка посилається на таблицю Attachments, де кожен запис містить URL файлу. Такий підхід відповідає рекомендаціям із проєктування REST-ресурсів: окремі, чітко визначені колекції для вкладених ресурсів дозволяють масштабувати схему й підтримувати різні типи файлів (зображення, відео тощо) без зміни самої моделі посту. Створення нового запису здійснюється через єдиний кінцевий пункт POST /api/posts. Клієнт передає заголовок, текст і перелік URL вкладень [39]. Система перевіряє, що щонайменше текст або вкладення не порожні, після чого створює основний запис у таблиці Post і пов’язує його з відповідними записами у Attachments. Цей підхід дозволяє зберегти єдину відповідальність за валідацію даних: контроль обов’язковості полів відбувається на рівні одного маршруту, що спрощує обслуговування та тестування.

Користувацька стрічка формується залежно від вказаного режиму (home, friends, following), причому за замовчуванням повертаються всі пости, крім власних. Параметр limit керує кількістю записів за запитом, а замість класичної «offset–limit» використовують курсорну пагінацію з маркером останнього отриманого елемента. Cursor-based пагінація запобігає пропуску або дублюванню даних під час високої активності й відповідає сучасним best practices для масштабованих API [46].

Коментарі реалізовано через ту саму механіку, що й повідомлення в чатах: кожен коментар – це запис у таблиці Message, зв’язаний із «постовим» чатом. Для підтримки вкладених коментарів застосовується поле ReplyToMessageId, яке вказує на батьківський запис, що дозволяє створювати гілки обговорення. Видалення коментарів виконується через soft-delete (IsDeleted), а редагування – через окремий PATCH-запит, доступний тільки автору. Такий підхід не дублює код: замість окремої сутності Comment використовується уніфікована архітектура повідомлень, що спрощує підтримку та розширення.

Реакції на пости зберігаються в таблиці UserPostInteraction і підтримують типи like, dislike, save, not_interested тощо [19]. Перед додаванням нової

взаємодії система видаляє попередні записи користувача для даного посту, щоб запобігти дублюванню реакцій. Така архітектура відповідає патерну «one-to-many with overwrite», коли остання дія користувача є актуальною, а база завжди містить лише поточний статус його взаємодії [21].

3.4 Тестування процесів функціонування серверної частини мобільного додатку

Для перевірки роботи серверної частини використовується інтерактивна документація Swagger UI, що дозволяє одразу виконувати GET, POST, PATCH і DELETE запити без потреби в окремих інструментах для тестування [35] (рис. 3.5). Основний фокус покладено на перевірку ключових сценаріїв: реєстрації користувача, аутентифікації, створення та отримання постів, надсилання й отримання повідомлень у чатах та запитів на отримання профілю. Хоча unit-тести та Postman наразі не застосовуються, Swagger UI виконує роль і документації, і засобу мануального тестування, забезпечуючи виконання всіх CRUD-операцій з живими даними [7].

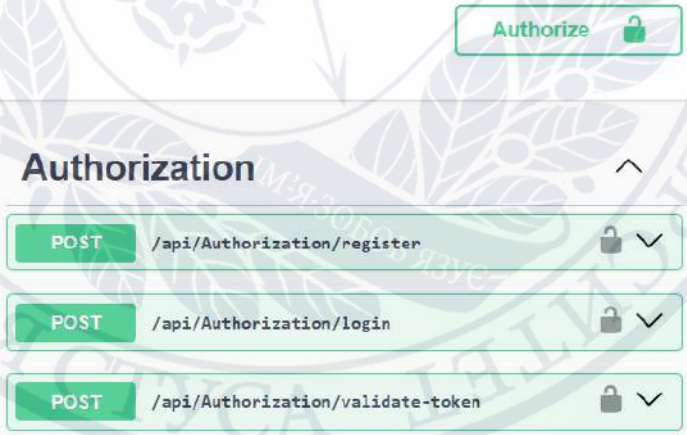

Рисунок 3.5 – Інтерфейс SwaggerUI

Після відправки запиту через Swagger UI здійснюється повторний GET-запит до відповідного ендпоїнту для підтвердження, що дані були записані коректно [47]. При цьому особлива увага приділяється зв’язкам між моделями: перевіряється наявність вкладень у постах, асоціації повідомлень із чатами та

коректне зберігання тегів інтересів для користувачів. Додатково тестується унікальність полів (email, nickname) та обробка помилок бази даних – порушення унікальних індексів повертає коректний HTTP-код із детальним повідомленням.

Документація генерується автоматично за допомогою Swashbuckle при кожному запуску проєкту й відображає всі доступні контролери та моделі, що спрощує інтеграцію фронтенд-розробників та проведення ручного тестування. Інтерактивний інтерфейс дозволяє не лише переглядати параметри запитів, а й одразу бачити зразки відповідей і помилок. Завдяки автоматичному оновленню документації внесені зміни до контролерів і моделей одразу відображаються в UI без додаткових налаштувань. Усі захищені ендпоїнти мають атрибут [Authorize], що контролюється через JWT. Тестування передбачає надсилання запитів із валідним, невалідним та простроченим токеном: очікувані відповіді 200 OK при валідному токені та 401 Unauthorized при помилковому чи відсутньому [39]. Сценарії з різними ролями (наприклад, адмін проти звичайного користувача) наразі не охоплюються, але механізм авторизації готовий до розширення.

Неправильні або неповні запити повертають відповіді з HTTP-статусами 400 Bad Request (некоректні дані), 404 Not Found (відсутній ресурс) або 401/403 (проблеми авторизації). Це відповідає загальним рекомендаціям щодо використання стандартних статусних кодів для REST API [17]. Для аудиту та діагностики помилок застосовується вбудована система логування ASP.NET Core, що дозволяє фіксувати всі ненормативні ситуації й переглядати їх у консолі або файлах логів [27]. Навантажувальне та тривалостійке тестування наразі не проводилося; усі запити виконуються вручну через Swagger UI. Однак для майбутніх перевірок продуктивності передбачено використання інструментів для load-тестування (наприклад, k6), що дозволить оцінити поведінку системи при великій кількості паралельних підключень та виявити потенційні витоки ресурсів.

ВИСНОВКИ

У межах даної бакалаврської роботи було розроблено серверну частину мобільного додатку, що забезпечує соціальну взаємодію користувачів на основі спільних інтересів. Архітектура системи побудована на сучасних принципах REST API з використанням ASP.NET Core та Entity Framework Core, що забезпечує масштабованість, гнучкість і зручність у подальшому розширенні функціональності. За результатами дослідження можна зробити наступні висновки:

1. Проведений аналіз особливостей функціонування сучасних соціальних платформ, основними функціями яких є реалізація взаємодії від особистого спілкування до професійного нетворкінгу, реклами та просування контенту. Визначено основні фактори, які вплинули на появу та розивток соціальних мереж, а також їх реалізацію в умовах цифрового простору, зокрема у вигляді онлайн-платформ.

2. Розроблено функціональні вимоги до мобільного додатку, зокрема його серверної частини. Ними визначено: реєстрацію користувача з забезпеченням перевірки унікальності ключових ідентифікаційних атрибутів, збереження паролів у вигляді криптографічних хешів, автентифікація користувачів реалізовона за допомогою компактного відкритого стандарту, впровадження механізмів контролю доступу до конкретних ресурсів.

3. Розроблена серверна частина мобільного додатку має трьохшарову архітектуру. Presentation-шар реалізовано в контролерах, що обробляють HTTP-запити, виконують початкову валідацію через атрибут [ApiController] і передають виклики до бізнес-логіки або безпосередньо до Entity Framework Core контексту. Domain-шар складається з доменних моделей (Models), які відображають структуру таблиць бази даних і містять навігаційні властивості для встановлення зв’язків між сутностями. Infrastructure-шар включає контекст SocialNetworkContext (розміщений у папці Data) та сервіси (у папці Services), що інкапсулюють бізнес-логіку.

4. Було реалізовано повноцінну систему реєстрації, аутентифікації та авторизації користувачів із використанням JWT, що дозволило забезпечити захист персональних даних та контроль доступу до обмежених ресурсів згідно з сучасними стандартами безпеки.

5. Забезпечено механізми безпечної взаємодії та обробки запитів. Особливу увагу приділено організації взаємодії між користувачами через пости, коментарі та реакції. Створено функціонал для додавання, перегляду, soft delete постів із підтримкою медіавкладень. Реакції на пости (лайки, дизлайки, збереження) зберігаються у вигляді окремої таблиці, що підвищує ефективність обробки запитів та спрощує масштабування.

6. Реалізовано механізм буферизації взаємодій на клієнті. Всі дії користувача з постами зберігаються локально у мобільному додатку та синхронізуються із сервером у форматі батч-запитів під час зміни вкладок. Такий підхід підвищує зручність користування додатком, зменшує кількість запитів до серверу та дозволяє працювати з додатком у режимі з нестабільним з’єднанням.

7. Проведено тестування основного функціоналу через Swagger UI. Це дозволило перевірити коректність роботи контролерів, цілісність зв’язків між моделями, обробку типових помилок та відповідність API поставленим вимогам.

В результаті роботи було створено гнучку, масштабовану серверну інфраструктуру, яка може бути використана як основа для повноцінного соціального мобільного додатку. Розроблена система придатна до інтеграції з іншими сервісами, зручна для супроводу та подальшого розвитку.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Як почався розвиток соціальних мереж. Treba. URL: https://treba-solutions.com/yak-pochavsya-rozvytok-soczialnyh-merezh/ (дата звернення: 04.05.2025).

2. Соціальні мережі: поняття, історія, виникнення. URL: https://zounb.zp.ua/resourse/zaporizkyy-kray/zaporizhzhya-bibliotechne/fahova-osvita/socialni-merezhi-piv (дата звернення: 04.05.2025).

3. Орап М.О., Лебединець О.С. Надмірне залучення до соціальних мереж: теоретичний аналіз. Science and Education a New Dimension: науковий журнал, 2020. С. 61–64.

4. Каніболоцька М.С. Теоретичні засади дослідження соціальних мереж у контексті соціально-психологічного супроводу освітніх інновацій. Наука і освіта: науково-практичний журнал, 2016. С. 99–105.

5. 7 Engaging Social Media App Features That Keep Users Active. URL: https://getstream.io/blog/social-media-app-features/ (дата звернення: 05.05.2025).

6. Top 15 Features of Social Media to Keep Your App Users Engaged. URL: https://coredevsltd.com/articles/features-of-social-media/ (дата звернення: 05.05.2025).

7. 10 Top Features of Social Media Apps. URL: https://www.koombea.com/blog/10-top-features-of-social-media-apps/ (дата звернення: 05.05.2025).

8. 7 Engaging Social Media App Features That Keep Users Active. URL: https://getstream.io/blog/social-media-app-features/ (дата звернення: 06.05.2025).

9. Introducing Threads: A New Way to Share With Text. URL: https://indglobal.in/top-7-game-changing-features-every-social-media-apps-needs-today/ (дата звернення: 06.05.2025).

10. Introducing Threads: A New Way to Share With Text. URL: https://about.fb.com/news/2023/07/introducing-threads-new-app-text-sharing/ (дата звернення: 06.05.2025).

11. Racism, misogyny, lies: how did X become so full of hatred? And is it ethical to keep using it? URL: https://www.theguardian.com/technology/article/2024/sep/05/racism-misogyny-lies-how-did-x-become-so-full-of-hatred-and-is-it-ethical-to-keep-using-it (дата звернення: 06.05.2025).

12. Sommerville, I. Software Engineering. 10th ed., Pearson, 2015. С. 339–373.

13. Provos, N., & Mazières, D. A Future-Adaptable Password Scheme. Proceedings of the 1999 USENIX Annual Technical Conference.

14. Password Storage - OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

15. Jones, M., Bradley, J., & Sakimura, N. RFC 7519: JSON Web Token (JWT). Internet Engineering Task Force, 2015.

16. RFC 7519: JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519

17. Saltzer, J. H., & Schroeder, M. D. The Protection of Information in Computer Systems. Proceedings of the IEEE, 63(9), 1975, С. 1278–1308. DOI: 10.1109/PROC.1975.9939.

18. Authorization - OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

19. The Beginner's Guide to Database Relationships | Luzmo. URL: https://www.luzmo.com/blog/database-relationships

20. Entity Framework Core Overview | Microsoft Learn. URL: https://learn.microsoft.com/en-us/ef/core/

21. Richardson, L., & Ruby, S. RESTful Web Services. O'Reilly Media, 2007.

22. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures.

23. Мартін, Р. К. Чиста архітектура. С. 109–137.

24. Getting Started Guide - AutoMapper documentation. URL: https://docs.automapper.org/en/stable/Getting-started.html#what-is-automapper

25. Learn C# - free tutorials, courses, videos, and more | .NET - Microsoft. URL: https://learn.microsoft.com/en-us/dotnet/csharp/

26. Overview of ASP.NET Core | Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-9.0

27. ASP.NET Core Fundamentals | Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0&tabs=windows

28. BCrypt Algorithm - Topcoder. URL: https://www.topcoder.com/thrive/articles/bcrypt-algorithm

29. System.IdentityModel.Tokens.Jwt Namespace | Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/api/system.identitymodel.tokens.jwt?view=msal-web-dotnet-latest&viewFallbackFrom=azure-dotnet4

30. A Complete Guide to the Different Types of DBMS - DbVisualizer. URL: https://www.dbvis.com/thetable/a-complete-guide-to-the-different-types-of-dbms/

31. NoSQL Databases Advantages and Disadvantages | Dataversity.net. URL: https://www.dataversity.net/nosql-databases-advantages-and-disadvantages/

32. SQL vs NoSQL: Bridging the Gap with Hybrid Databases - TiDB. URL: https://www.pingcap.com/article/sql-vs-nosql-bridging-the-gap-with-hybrid-databases/

33. MySQL: Understanding What It Is and How It's Used | Oracle Україна. URL: https://www.oracle.com/ua/mysql/what-is-mysql/

34. Database Providers - EF Core | Microsoft Learn. URL: https://learn.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

35. Swagger UI | Swagger. URL: https://swagger.io/tools/swagger-ui/

36. OpenAPI Specification | Swagger. URL: https://swagger.io/specification/

37. Pomelo Foundation. Pomelo.EntityFrameworkCore.MySql. URL: https://github.com/PomeloFoundation/Pomelo.EntityFrameworkCore.MySql (дата звернення: 14.05.2025).

38. BCrypt.Net Next Documentation. URL: https://github.com/BcryptNet/bcrypt.net (дата звернення: 14.05.2025).

39. Microsoft. [ApiController] attribute. URL: https://learn.microsoft.com/aspnet/core/web-api/?view=aspnetcore-9.0#apicontroller-attribute (дата звернення: 14.05.2025).

40. Microsoft. Configuring EF Core model with Fluent API. URL: https://learn.microsoft.com/ef/core/modeling/ (дата звернення: 14.05.2025).

41. Microsoft. Query Types–AsNoTracking. URL: https://learn.microsoft.com/ef/core/querying/tracking#no-tracking-queries (дата звернення: 14.05.2025).

42. Microsoft. Simple authorization in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/security/authorization/simple?view=aspnetcore-9.0 (дата звернення: 14.05.2025).

43. Microsoft. Enable Cross-Origin Requests (CORS) in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/security/cors?view=aspnetcore-9.0 (дата звернення: 14.05.2025).

44. Microsoft. Enforce HTTPS in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-9.0 (дата звернення: 14.05.2025).

45. Fielding, R. T. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, RFC 7231, Section 4.2.2 “Safe Methods and Idempotent Methods”.

46. Stripe API Reference. Pagination. URL: https://stripe.com/docs/api/pagination (дата звернення: 14.05.2025).

47. Microsoft. Getting started with Swashbuckle and Swagger in ASP.NET Core. URL: https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-9.0 (дата звернення: 14.05.2025).

ДОДАТКИ

ДОДАТОК А

Program.cs

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Server.Services;
using Server.Data;
using System.Text;
using System.Net;

namespace Server
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);


builder.Services.AddAutoMapper(typeof(Program).Assembly);
builder.Services.AddHealthChecks();

            builder.Services.AddAuthorization(options =>
            {
                options.AddPolicy("AdministratorPolicy", policy =>
                    policy.RequireRole("administrator"));

                options.AddPolicy("UserPolicy", policy =>
                    policy.RequireRole("user"));

                options.AddPolicy("RegisteredPolicy", policy =>
                    policy.RequireRole("user", "administrator"));
            });

            var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
            builder.Services
                .AddDbContext<SocialNetworkContext>(options =>
                options.UseMySql(connectionString,
ServerVersion.AutoDetect(connectionString)));

            builder.Services
                .AddControllers();
            builder.Services.AddEndpointsApiExplorer();
            builder.Services.AddSwaggerGen(c =>
            {
```

```
                c.SwaggerDoc("v1", new() { Title = "SocialNetwork
API", Version = "v1" });

                c.AddSecurityDefinition("Bearer", new
Microsoft.OpenApi.Models.OpenApiSecurityScheme
                {
                    Name = "Authorization",
                    Type =
Microsoft.OpenApi.Models.SecuritySchemeType.Http,
                    Scheme = "bearer",
                    BearerFormat = "JWT",
                    In =
Microsoft.OpenApi.Models.ParameterLocation.Header,
                    Description = "Enter 'Bearer' [space] and then
your token in the text input below.\n\nExample: 'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...'"
                });

                c.AddSecurityRequirement(new
Microsoft.OpenApi.Models.OpenApiSecurityRequirement
    {
        {
            new Microsoft.OpenApi.Models.OpenApiSecurityScheme
            {
                Reference = new
Microsoft.OpenApi.Models.OpenApiReference
                {
                    Type =
Microsoft.OpenApi.Models.ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            new string[] {}
        }
    });
            });


            var key =
Encoding.ASCII.GetBytes(builder.Configuration["Jwt:Key"]);

            var tokenService = new
TokenService(builder.Configuration);
            builder.Services.AddScoped(provider => tokenService);
```

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                options.RequireHttpsMetadata = false;
                options.SaveToken = true;
                options.TokenValidationParameters = tokenService.GetTokenValidationParameters();
            });

            builder.Services.AddCors(options =>
            {
                options.AddDefaultPolicy(builder =>
                {
                    builder.AllowAnyOrigin()
                           .AllowAnyMethod()
                           .AllowAnyHeader();
                });
            });

            var app = builder.Build();

            if (app.Environment.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseSwagger();
                app.UseSwaggerUI();
            }

            app.UseStaticFiles();
            app.UseHttpsRedirection();
            app.UseCors();
            app.UseAuthentication();
            app.UseAuthorization();
            app.MapControllers();
            app.MapHealthChecks("/health");
            app.Run();
        }
    }
}
```

MappingProfile.cs

```
using AutoMapper;
using Server.Controllers;
using Server.Models;
```

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        CreateMap<User, UserPublic>()
            .ForMember(dest => dest.Id, opt => opt.MapFrom(src =>
src.Id))
            .ForMember(dest => dest.Nickname, opt =>
opt.MapFrom(src => src.Nickname))
            .ForMember(dest => dest.AvatarUrl, opt =>
opt.MapFrom(src => src.AvatarUrl))
            .ForMember(dest => dest.Description, opt =>
opt.MapFrom(src => src.Description))
            .ForMember(dest => dest.FullName, opt =>
opt.MapFrom(src => src.FullName));
        CreateMap<User, UserFull>()
            .ForMember(dest => dest.Id, opt => opt.MapFrom(src =>
src.Id))
            .ForMember(dest => dest.Nickname, opt =>
opt.MapFrom(src => src.Nickname))
            .ForMember(dest => dest.AvatarUrl, opt =>
opt.MapFrom(src => src.AvatarUrl))
            .ForMember(dest => dest.Description, opt =>
opt.MapFrom(src => src.Description))
            .ForMember(dest => dest.FullName, opt =>
opt.MapFrom(src => src.FullName))
            .ForMember(dest => dest.UpdatedAt, opt =>
opt.MapFrom(src => src.UpdatedAt))
            .ForMember(dest => dest.CreatedAt, opt =>
opt.MapFrom(src => src.CreatedAt))
            .ForMember(dest => dest.IsDeleted, opt =>
opt.MapFrom(src => src.IsDeleted))
            .ForMember(dest => dest.Role, opt => opt.MapFrom(src
=> src.Role))
            .ForMember(dest => dest.Email, opt => opt.MapFrom(src
=> src.Email));
    }
}
```

AuthorizationController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Data;
using Server.Models;
using Server.Services;
```

```
namespace Server.ClientControllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class AuthorizationController : ControllerBase
    {
        private readonly SocialNetworkContext _context;
        private readonly TokenService _tokenService;

        public AuthorizationController(SocialNetworkContext
context, TokenService tokenService)
        {
            _context = context;
            _tokenService = tokenService;
        }

        [HttpPost("register")]
        public async Task<IActionResult> Register([FromBody]
RegistrationRequest request)
        {
            var existingUser = await
_context.Users.FirstOrDefaultAsync(u => u.Email == request.Email);
            if (existingUser != null)
            {
                return BadRequest("Email already in use.");
            }

            var existingNickname = await
_context.Users.FirstOrDefaultAsync(u => u.Nickname ==
request.Nickname);
            if (existingNickname != null)
            {
                return BadRequest("Nickname already in use.");
            }

            if (string.IsNullOrWhiteSpace(request.Password))
            {
                return BadRequest("Password cannot be empty.");
            }

            if (string.IsNullOrWhiteSpace(request.Nickname))
            {
                return BadRequest("Nickname cannot be empty.");
            }

            var newUser = new User
```

```
            {
                Email = request.Email,
                Password =
BCrypt.Net.BCrypt.HashPassword(request.Password),
                Nickname = request.Nickname,
                AvatarUrl = request.Avatar,
                FullName = request.FullName,
            };

            _context.Users.Add(newUser);
            await _context.SaveChangesAsync();

            if (request.Interests != null &&
request.Interests.Count > 0)
            {
                var existingTags = await _context.Tags
                    .Where(t =>
request.Interests.Contains(t.Name))
                    .ToListAsync();

                var newTags = request.Interests
                    .Where(interest => !existingTags.Any(t =>
t.Name == interest))
                    .Select(interest => new Tag { Name = interest
})
                    .ToList();

                if (newTags.Any())
                {
                    await _context.Tags.AddRangeAsync(newTags);
                    await _context.SaveChangesAsync();
                    existingTags.AddRange(newTags);
                }

                var userTags = existingTags.Select(tag => new
UserTagWeight
                {
                    UserId = newUser.Id,
                    TagId = tag.Id,
                    Weight = 1
                }).ToList();

                if (userTags.Any())
                {
                    await
_context.UserTagWeights.AddRangeAsync(userTags);
                    await _context.SaveChangesAsync();
```

```
                }
            }

            if (request.NotInterested != null &&
request.NotInterested.Count > 0)
            {
                var existingTags = await _context.Tags
                    .Where(t =>
request.NotInterested.Contains(t.Name))
                    .ToListAsync();

                var newTags = request.NotInterested
                    .Where(interest => !existingTags.Any(t =>
t.Name == interest))
                    .Select(interest => new Tag { Name = interest
})
                    .ToList();

                if (newTags.Any())
                {
                    await _context.Tags.AddRangeAsync(newTags);
                    await _context.SaveChangesAsync();
                    existingTags.AddRange(newTags);
                }

                var userTags = existingTags.Select(tag => new
UserTagWeight
                {
                    UserId = newUser.Id,
                    TagId = tag.Id,
                    Weight = -1
                }).ToList();

                if (userTags.Any())
                {
                    await
_context.UserTagWeights.AddRangeAsync(userTags);
                    await _context.SaveChangesAsync();
                }
            }

            var token = _tokenService.GenerateJwtToken(newUser);

            return Ok(new { token });
        }

        [HttpPost("login")]
```

```
        public async Task<IActionResult> Login([FromBody]
LoginRequest request)
        {
            User? user;

            if (request.Login.Contains('@'))
            {
                user = await _context.Users.FirstOrDefaultAsync(u
=> u.Email == request.Login);
            }
            else
            {
                user = await _context.Users.FirstOrDefaultAsync(u
=> u.Nickname == request.Login);
            }

            if (user == null)
            {
                return NotFound("User not found.");
            }

            if (!BCrypt.Net.BCrypt.Verify(request.Password,
user.Password))
            {
                return Unauthorized("Invalid password.");
            }

            var token = _tokenService.GenerateJwtToken(user);
            return Ok(new { token });
        }

        [HttpPost("validate-token")]
        public IActionResult ValidateToken([FromBody]
TokenValidationRequest request)
        {
            try
            {
                var principal =
_tokenService.ValidateJwtToken(request.Token);
                if (principal == null)
                {
                    return Unauthorized("Invalid token.");
                }

                return Ok(new { Message = "Token is valid." });
            }
            catch (Exception ex)
```

```
            {
                return BadRequest(new { Message = "Token
validation failed.", Error = ex.Message });
            }
        }

        public class TokenValidationRequest
        {
            public string Token { get; set; } = null!;
        }

        public struct RegistrationRequest
        {
            public string Email { get; set; }
            public string Password { get; set; }
            public string Nickname { get; set; }
            public string FullName { get; set; }
            public string Avatar { get; set; }
            public List<string> Interests { get; set; }
            public List<string> NotInterested { get; set; }
        }

        public struct LoginRequest
        {
            public string Login { get; set; }
            public string Password { get; set; }
        }
    }
}
```

MeController.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Controllers;
using Server.Data;
using Server.Models;
using System.Security.Claims;

namespace Server.ClientControllers;

[Authorize]
[ApiController]
[Route("api/[controller]")]
public class MeController : ControllerBase
```

```
{
    private readonly SocialNetworkContext _context;
    private readonly IMapper _mapper;

    public MeController(SocialNetworkContext context, IMapper
mapper)
    {
        _mapper = mapper;
        _context = context;
    }

    [HttpGet]
    public async Task<ActionResult<UserFull>> GetUser()
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        var user = await _context.Users
            .Include(u => u.Posts)
            .FirstOrDefaultAsync(u => u.Id == userId);
        if (user == null)
        {
            return NotFound("User not found.");
        }
        var userFull = _mapper.Map<UserFull>(user);
        return Ok(userFull);
    }

    [HttpDelete]
    public async Task<IActionResult> DeleteUser()
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        var user = await _context.Users
            .Include(u => u.Posts)
            .FirstOrDefaultAsync(u => u.Id == userId);
        if (user == null)
        {
            return NotFound("User not found.");
        }
        user.IsDeleted = true;
        await _context.SaveChangesAsync();
        return Ok("User deleted successfully.");
    }

    [HttpPatch("public")]
    public async Task<IActionResult> UpdateUserPublic([FromBody]
UpdateUserPublicRequest request)
```

```
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        var user = await _context.Users
            .Include(u => u.Posts)
            .FirstOrDefaultAsync(u => u.Id == userId);

        if (request.AvatarUrl != null)
        {
            user!.AvatarUrl = request.AvatarUrl;
        }
        if (request.Description != null)
        {
            user!.Description = request.Description;
        }
        if (request.FullName != null)
        {
            user!.FullName = request.FullName;
        }

        await _context.SaveChangesAsync();
        return Ok("User updated successfully.");
    }

    [HttpPatch("private")]
    public async Task<IActionResult> UpdateUserPrivate([FromBody]
UpdateUserPrivateRequest request)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        var user = await _context.Users
            .Include(u => u.Posts)
            .FirstOrDefaultAsync(u => u.Id == userId);
        if (request.Nickname != null)
        {
            var existingNickname = await _context.Users
                .FirstOrDefaultAsync(u => u.Nickname ==
request.Nickname && u.Id != userId);
            if (existingNickname != null)
            {
                return BadRequest("Nickname already in use.");
            }
            user!.Nickname = request.Nickname;
        }
        if (request.Email != null)
        {
            var existingEmail = await _context.Users
```

```
                .FirstOrDefaultAsync(u => u.Email == request.Email
&& u.Id != userId);
            if (existingEmail != null)
            {
                return BadRequest("Email already in use.");
            }
            user!.Email = request.Email;
        }
        if (request.Password != null)
        {
            user!.Password =
BCrypt.Net.BCrypt.HashPassword(request.Password);
        }
        await _context.SaveChangesAsync();
        return Ok("User updated successfully.");
    }

    [HttpGet("interactions")]
    public async Task<ActionResult<IEnumerable<PostData>>>
GetInteractionsPosts(string type = "create")
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        List<Post> posts = new List<Post>();
        if (type == "create")
        {
            posts = await _context.Posts
            .Include(p => p.Creator)
            .Include(p => p.UserPostInteractions)
            .Include(p => p.PostAttachments)
                .ThenInclude(pa => pa.Attachment)
            .Where(p => p.CreatorId == userId)
            .OrderByDescending(p => p.UpdatedAt)
            .ToListAsync();
        }
        else
        {
            posts = await _context.UserPostInteractions
                .Include(p => p.Post)
                .Include(p => p.Post.Creator)
                .Include(p => p.Post.UserPostInteractions)
                .Where(i => i.UserId == userId && i.Type == type)
                .Select(i => i.Post)
                .OrderByDescending(i => i.UserPostInteractions
                    .Where(upi => upi.UserId == userId && upi.Type
== type)
                    .Select(upi => upi.CreatedAt)
```

```
                    .FirstOrDefault())
                .ToListAsync();
        }

        var postsData = posts.Select(p => new PostData
        {
            Id = p.Id,
            Title = p.Title,
            Text = p.Text ?? string.Empty,
            Attachments = p.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
            ChatId = p.Chat?.Id ?? 0,
            Creator = _mapper.Map<UserPublic>(p.Creator),
            Interactions = p.UserPostInteractions.Select(upi =>
upi.Type).ToList()
        });

        return Ok(postsData);
    }

    [HttpGet("relations")]
    public async Task<ActionResult<IEnumerable<UserPublic>>>
GetRelations(string type = "friends")
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        if(userId == 0)
        {
            return BadRequest("User not found.");
        }
        List<User> relations = new List<User>();

        if (type == "friends")
        {
            relations =
                await _context.UserFollowers
                    .Include(uf => uf.User)
                    .Include(uf => uf.Follower)
                    .Where(uf => uf.User.Id == userId &&
uf.Follower.Id != userId)
                    .Select(uf => uf.Follower)
                    .Intersect(
                        _context.UserFollowers
                            .Include(uf => uf.User)
                            .Include(uf => uf.Follower)
                            .Where(uf => uf.Follower.Id == userId
&& uf.User.Id != userId)
```

```
                            .Select(uf => uf.User)
                    )
                    .ToListAsync();
        }
        else if (type == "followers")
        {
            var followingIds = await _context.UserFollowers
                .Where(uf => uf.Follower.Id == userId)
                .Select(uf => uf.User.Id)
                .ToListAsync();

            relations = await _context.UserFollowers
                .Include(uf => uf.Follower)
                .Where(uf => uf.User.Id == userId &&
!followingIds.Contains(uf.Follower.Id))
                .Select(uf => uf.Follower)
                .ToListAsync();
        }
        else if (type == "following")
        {
            var followersIds = await _context.UserFollowers
                .Where(uf => uf.User.Id == userId)
                .Select(uf => uf.Follower.Id)
                .ToListAsync();
            relations = await _context.UserFollowers
                .Include(uf => uf.User)
                .Where(uf => uf.Follower.Id == userId &&
!followersIds.Contains(uf.User.Id))
                .Select(uf => uf.User)
                .ToListAsync();
        }
        else
        {
            return BadRequest("Invalid type. Valid types are:
friends, followers, following.");
        }

        var usersPublic =
_mapper.Map<List<UserPublic>>(relations);

        return Ok(usersPublic);
    }

    [HttpGet("notifications")]
    public async Task<ActionResult<IEnumerable<Notification>>>
GetNotifications()
    {
```

```
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
        if (userId == 0)
        {
            return BadRequest("User not found.");
        }
        var notifications = await _context.Notifications
            .Include(n => n.User)
            .Where(n => n.Receiver.Id == userId)
            .Select(n => new NotificationData
            {
                Type = n.Type,
                User = _mapper.Map<UserPublic>(n.User)
            })
            .ToListAsync();
        return Ok(notifications);
    }

    public class NotificationData
    {
        public string Type { get; set; } = null!;
        public UserPublic User { get; set; } = null!;
    }


    public class UpdateUserPublicRequest
    {
        public string? AvatarUrl { get; set; }
        public string? Description { get; set; }
        public string? FullName { get; set; }
    }

    public class UpdateUserPrivateRequest
    {
        public string? Email { get; set; }
        public string? Password { get; set; }
        public string? Nickname { get; set; }
    }

    public class CreatePostRequest
    {
        public string Title { get; set; } = null!;
        public string? Text { get; set; }
        public List<string>? Attachments { get; set; }
    }
}
```

ChatsController.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Controllers;
using Server.Data;
using Server.Models;
using System.Security.Claims;

[ApiController]
[Route("api/[controller]")]
[Authorize]
public class ChatsController : ControllerBase
{
    private readonly SocialNetworkContext _context;
    private readonly IMapper _mapper;
    public ChatsController(SocialNetworkContext context, IMapper
mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<ChatPreview>>>
GetAllChats()
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");

        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }

        var groupChats = await _context.GroupMembers
            .Where(gm => gm.UserId == userId)
            .Select(gm => new ChatPreview
            {
                Id = gm.GroupChat.Chat.Id,
                Type = "group",
                AvatarUrl = gm.GroupChat.AvatarUrl,
                Title = gm.GroupChat.Name,
                UpdatedAt = gm.GroupChat.Chat.UpdatedAt
```

```
            })
            .OrderByDescending(c => c.UpdatedAt)
            .ToListAsync();
        var privateChats = await _context.PrivateChats
            .Where(pc => pc.User1Id == userId || pc.User2Id ==
userId)
            .Select(pc => new ChatPreview
            {
                Id = pc.Chat.Id,
                Type = "private",
                AvatarUrl = pc.User1Id == userId ?
pc.User2.AvatarUrl : pc.User1.AvatarUrl,
                Title = pc.User1Id == userId ? pc.User2.FullName :
pc.User1.FullName,
                UpdatedAt = pc.Chat.UpdatedAt
            })
            .OrderByDescending(c => c.UpdatedAt)
            .ToListAsync();
        var allChats = groupChats.Concat(privateChats)
            .OrderByDescending(c => c.UpdatedAt)
            .ToList();
        return Ok(allChats);
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetChat(int id, DateTime?
cursor, int limit = 10)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");

        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }

        var chat = await _context.Chats
    .Include(c => c.GroupChat)
    .Include(c => c.PrivateChat)
        .ThenInclude(pc => pc.User1)
    .Include(c => c.PrivateChat)
        .ThenInclude(pc => pc.User2)
    .FirstOrDefaultAsync(c => c.Id == id);

        if (chat == null)
        {
```

```
            return NotFound("Chat not found.");
        }

        if (chat.IsDeleted == 1)
        {
            return BadRequest("Chat is deleted.");
        }

        var messages = await _context.Messages
            .Where(m => m.ChatId == id && !m.IsDeleted &&
(!cursor.HasValue || m.CreatedAt >= cursor))
            .OrderBy(m => m.UpdatedAt)
            .Take(limit)
            .Select(m => new MessageData
            {
                Id = m.Id,
                Text = m.Text,
                CreatedAt = m.CreatedAt,
                Attachments = _context.MessageAttachments
                    .Where(a => a.MessageId == m.Id)
                    .Select(a => a.Attachment.Url)
                    .ToList(),
                ReplyToMessage = _context.Messages
                    .Where(r => r.Id == m.ReplyToMessageId)
                    .Select(r => new ReplyPreview
                    {
                        Id = r.Id,
                        Text = r.Text,
                        Attachment = _context.MessageAttachments
                            .Where(a => a.MessageId == r.Id)
                            .Select(a => a.Attachment.Url)
                            .FirstOrDefault()
                    })
                    .FirstOrDefault(),
                Sender = _mapper.Map<UserPublic>(m.Sender),
                DiscussionId = m.DiscussionId
            })
            .ToListAsync();
        var info = new ChatInfo { Id = id};
        object additional = null!;
        if (chat.GroupChat != null)
        {
            info.Type = "group";
            additional = new
            {
                Title = chat.GroupChat.Name,
                AvatarUrl = chat.GroupChat.AvatarUrl,
```

```
                MembersIds = await _context.GroupMembers
                    .Where(gm => gm.GroupChatId ==
chat.GroupChat.Id)
                    .Select(gm => gm.UserId)
                    .ToListAsync()
            };
        }
        else if (chat.PrivateChat != null)
        {
            info.Type = "private";
            additional = new
            {
                User =
_mapper.Map<UserPublic>(chat.PrivateChat.User1Id == userId ?
chat.PrivateChat.User2 : chat.PrivateChat.User1)
            };
        }
        else
        {
            return NotFound("Chat not found.");
        }

        if (cursor.HasValue)
        {
            return Ok(new { info, messages });
        }
        else
        {
            return Ok(new { info, additional, messages });
        }
    }

    [HttpPost("group")]
    public async Task<IActionResult> CreateGroupChat([FromBody]
CreateGroupChatRequest request)
    {
        if
(!int.TryParse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value,
out var userId) || userId == 0)
        {
            return Unauthorized("User not found.");
        }

        if (string.IsNullOrWhiteSpace(request.Title))
        {
            return BadRequest("Chat title is required.");
        }
```

```
        if (request.MembersIds == null ||
!request.MembersIds.Any())
        {
            return BadRequest("At least one member is required.");
        }

        var groupChat = new Chat
        {
            Type = "group",
            UpdatedAt = DateTime.UtcNow,
            GroupChat = new GroupChat
            {
                Name = request.Title,
                AvatarUrl = request.AvatarUrl
            }
        };

        using var transaction = await
_context.Database.BeginTransactionAsync();
        try
        {
            _context.Chats.Add(groupChat);
            await _context.SaveChangesAsync();

            var memberIds = request.MembersIds.Select(m =>
m.UserId).ToList();
            var existingUsers = await _context.Users
                .Where(u => memberIds.Contains(u.Id))
                .ToListAsync();

            foreach (var member in request.MembersIds)
            {
                var existingUser = existingUsers.FirstOrDefault(u
=> u.Id == member.UserId);
                if (existingUser == null)
                {
                    return BadRequest($"User with id
{member.UserId} not found.");
                }

                var groupMember = new GroupMember
                {
                    UserId = member.UserId,
                    GroupChatId = groupChat.GroupChat.Id,
                    Role = member.Role
                };
```

```
                _context.GroupMembers.Add(groupMember);
            }

            await _context.SaveChangesAsync();
            await transaction.CommitAsync();

            return Ok(new
            {
                Message = $"New group chat created successfully
with name {request.Title}",
                ChatId = groupChat.Id,
                GroupChat = new
                {
                    groupChat.GroupChat.Name,
                    groupChat.GroupChat.AvatarUrl
                }
            });
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
    [HttpPost("private")]
    public async Task<IActionResult> CreatePrivateChat([FromBody]
CreatePrivateChatRequest request)
    {
        if
(!int.TryParse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value,
out var userId) || userId == 0)
        {
            return Unauthorized("User not found.");
        }

        if (request.UserId == userId)
        {
            return BadRequest("You cannot create a private chat
with yourself.");
        }

        var otherUser = await _context.Users.FirstOrDefaultAsync(u
=> u.Id == request.UserId);
        if (otherUser == null)
        {
            return NotFound($"User with id {request.UserId} not
found.");
```

```
        }

        var existingChat = await _context.PrivateChats
            .Include(pc => pc.Chat)
            .FirstOrDefaultAsync(pc =>
                (pc.User1Id == userId && pc.User2Id ==
request.UserId) ||
                (pc.User1Id == request.UserId && pc.User2Id ==
userId));

        if (existingChat != null)
        {
            return Ok(new
            {
                Message = "Private chat already exists.",
                ChatId = existingChat.Chat!.Id
            });
        }

        var privateChat = new Chat
        {
            Type = "private",
            UpdatedAt = DateTime.UtcNow,
            PrivateChat = new PrivateChat
            {
                User1Id = userId,
                User2Id = request.UserId
            }
        };

        _context.Chats.Add(privateChat);
        await _context.SaveChangesAsync();

        var updatedChat = await _context.Chats
            .Include(c => c.PrivateChat)
            .ThenInclude(pc => pc.User1)
            .Include(c => c.PrivateChat!.User2)
            .FirstOrDefaultAsync(c => c.Id == privateChat.Id);

        return Ok(new
        {
            Message = "Private chat created successfully.",
            ChatId = privateChat.Id,
        });

    }
}
```

```
public class ChatInfo
{
    public int Id { get; set; }
    public string Type { get; set; } = null!;
}

public struct ChatPreview
{
    public int Id { get; set; }
    public string Type { get; set; }
    public string AvatarUrl { get; set; }
    public string Title { get; set; }
    public DateTime UpdatedAt { get; set; }
}

public struct CreateGroupChatRequest
{
    public string Title { get; set; }
    public string? AvatarUrl { get; set; }
    public List<GroupMemberData> MembersIds { get; set; }
}

public struct CreatePrivateChatRequest
{
    public int UserId { get; set; }
}

public struct GroupMemberData
{
    public int UserId { get; set; }
    public string Role { get; set; }
}

public class ChatData
{
    public int Id { get; set; }
}

public class GroupChatData : ChatData
{
    public string Title { get; set; } = null!;
    public string? AvatarUrl { get; set; }
    public List<int> MembersIds { get; set; } = null!;
}

public class PrivateChatData : ChatData
```

```
{
    public UserPublic User { get; set; } = null!;
}

internal struct GetChatResponse
{
    public ChatData? Chat { get; set; }
    public List<MessageData> Messages { get; set; }
}

public class MessageData
{
    public int Id { get; set; }
    public string? Text { get; set; }
    public DateTime CreatedAt { get; set; }
    public List<string>? Attachments { get; set; }
    public ReplyPreview? ReplyToMessage { get; set; }
    public UserPublic Sender { get; set; } = null!;
    public int? DiscussionId { get; set; }
}

public class ReplyPreview
{
    public int Id { get; set; }
    public string? Text { get; set; }
    public string? Attachment { get; set; }
}
```

MessagesController.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Data;
using Server.Models;
using System.Security.Claims;

[Route("api/[controller]")]
[Authorize]
public class MessagesController : ControllerBase
{
    private readonly SocialNetworkContext _context;
    private readonly IMapper _mapper;
    public MessagesController(SocialNetworkContext context,
IMapper mapper)
    {
```

```
        _context = context;
        _mapper = mapper;
    }

    [HttpPatch("{messageId}")]
    public async Task<IActionResult> UpdateMessage(int messageId,
[FromBody] UpdateMessageRequest request)
    {
        var message = await
_context.Messages.FindAsync(messageId);

        if (message == null)
        {
            return NotFound("Message not found.");
        }

        var corresndingAttachments = await
_context.MessageAttachments
            .Where(a => a.MessageId == messageId)
            .ToListAsync();

        if (string.IsNullOrWhiteSpace(request.Text))
        {
            if (corresndingAttachments.Count == 0)
            {
                return BadRequest("Message text cannot be
empty.");
            }
        }

        message.Text = request.Text;
        message.UpdatedAt = DateTime.UtcNow;

        await _context.SaveChangesAsync();
        return Ok("Message updated successfully.");
    }

    [HttpDelete("{messageId}")]
    public async Task<IActionResult> DeleteMessage(int messageId)
    {
        var message = await
_context.Messages.FindAsync(messageId);
        if (message == null)
        {
            return NotFound("Message not found.");
        }
        message.IsDeleted = true;
```

```
        await _context.SaveChangesAsync();
        return Ok("Message deleted successfully.");
    }

    [HttpPost("{chatId}")]
    public async Task<IActionResult> SendMessage(int chatId,
[FromBody] SendMessageRequest request)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");

        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }

        Console.WriteLine(request.Text);
        Console.WriteLine(request.Attachments);

        var chat = await _context.Chats
            .FirstOrDefaultAsync(c => c.Id == chatId);

        if (chat == null)
        {
            return NotFound("Chat not found.");
        }

        var message = new Message
        {
            ChatId = chat.Id,
            SenderId = userId,
            Text = request.Text,
            UpdatedAt = DateTime.UtcNow,

        };

        _context.Messages.Add(message);
        await _context.SaveChangesAsync();

        if (request.Attachments != null &&
request.Attachments.Count > 0)
        {
            foreach (var attachmentUrl in request.Attachments)
            {
                var existingAttachment = await
_context.Attachments
```

```
                    .FirstOrDefaultAsync(a => a.Url ==
attachmentUrl);

                Attachment attachment;

                if (existingAttachment != null)
                {
                    attachment = existingAttachment;
                }
                else
                {
                    attachment = new Attachment
                    {
                        Url = attachmentUrl
                    };

                    _context.Attachments.Add(attachment);
                    await _context.SaveChangesAsync();
                }

                var messageAttachment = new MessageAttachment
                {
                    MessageId = message.Id,
                    AttachmentId = attachment.Id
                };


_context.MessageAttachments.Add(messageAttachment);
            }

            await _context.SaveChangesAsync();
        }

        return Ok("Message sent successfully.");
    }
}

public class SendMessageRequest
{
    public string Text { get; set; }
    public List<string> Attachments { get; set; } = new
List<string>();
}

public class UpdateMessageRequest
{
    public string Text { get; set; } = null!;
```

```
}
```

PostsController.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Data;
using Server.Models;
using System.Security.Claims;
using System.Text.RegularExpressions;
namespace Server.Controllers;

[ApiController]
[Route("api/[controller]")]
public class PostsController : ControllerBase
{
    private readonly SocialNetworkContext _context;
    private readonly IMapper _mapper;
    public PostsController(SocialNetworkContext context, IMapper
mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    [HttpGet("feed")]
    public async Task<ActionResult<IEnumerable<PostData>>>
GetFeed(string category = "home", int limit = 10)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");

        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }

        IQueryable<Post> postsQuery = _context.Posts
            .Include(p => p.Creator)
            .Include(p => p.PostAttachments)
                .ThenInclude(pa => pa.Attachment)
            .Include(p => p.UserPostInteractions);

        if (category == "friends")
```

```
        {
            var friendsIds = await _context.UserFollowers
                .Where(uf => uf.FollowerId == userId)
                .Select(uf => uf.UserId)
                .Intersect(
                    _context.UserFollowers
                        .Where(uf => uf.UserId == userId)
                        .Select(uf => uf.FollowerId)
                )
                .ToListAsync();

            postsQuery = postsQuery.Where(p =>
friendsIds.Contains(p.CreatorId));
        }
        else if (category == "following")
        {
            var followedIds = await _context.UserFollowers
                .Where(uf => uf.FollowerId == userId)
                .Select(uf => uf.UserId)
                .ToListAsync();

            postsQuery = postsQuery.Where(p =>
followedIds.Contains(p.CreatorId));
        }
        else if (category == "home")
        {
            postsQuery = postsQuery.Where(p => p.CreatorId !=
userId);
        }
        else
        {
            return BadRequest("Invalid category. Valid categories
are: home, friends, followed.");
        }

        var posts = await postsQuery
            .OrderBy(p => Guid.NewGuid())
            .Take(limit)
            .ToListAsync();

        var feed = posts.Select(p => new PostData
        {
            Id = p.Id,
            Title = p.Title,
            Text = p.Text ?? string.Empty,
            Attachments = p.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
```

```
            ChatId = p.Chat?.Id ?? 0,
            Creator = _mapper.Map<UserPublic>(p.Creator),
            Interactions = p.UserPostInteractions.Select(upi =>
upi.Type).ToList()
        });

        return Ok(feed);
    }

    [HttpGet("{postId}")]
    public async Task<ActionResult<PostData>> GetPost(int postId)
    {
        var post = await _context.Posts
            .Include(p => p.Creator)
            .Include(p => p.PostAttachments)
                .ThenInclude(pa => pa.Attachment)
            .Include(p => p.UserPostInteractions)
            .Where(p => p.Id == postId)
            .FirstOrDefaultAsync();

        var postsData = new PostData
        {
            Id = post.Id,
            Title = post.Title,
            Text = post.Text ?? string.Empty,
            Attachments = post.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
            ChatId = post.Chat?.Id ?? 0,
            Creator = _mapper.Map<UserPublic>(post.Creator),
            Interactions =
            post.UserPostInteractions
                .Where(upi => upi.UserId ==
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0"))
                .Select(upi => upi.Type)
                .ToList()
        };

        return Ok(postsData);
    }

    [Authorize]
    [HttpPost]
    public async Task<IActionResult> CreatePost([FromBody]
ClientControllers.MeController.CreatePostRequest request)
    {
```

85

```
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);

        var user = await _context.Users
            .Include(u => u.Posts)
            .FirstOrDefaultAsync(u => u.Id == userId);

        if (user == null)
        {
            return NotFound("User not found.");
        }
        if (string.IsNullOrWhiteSpace(request.Text) &&
(request.Attachments == null || !request.Attachments.Any()))
        {
            return BadRequest("You must provide either text or
attachments.");
        }

        var newPost = new Post
        {
            CreatorId = userId,
            Title = request.Title,
            Text = request.Text,
        };

        _context.Posts.Add(newPost);
        await _context.SaveChangesAsync();

        if (request.Attachments != null &&
request.Attachments.Any())
        {
            var attachments = new List();
            foreach (var item in request.Attachments)
            {
                string url = item;
                if (item.StartsWith("data:image"))
                {
                    var match = Regex.Match(item,
@"data:image/(?.+?)base64,(?.+)");
                    if (match.Success)
                    {
                        var ext = match.Groups["type"].Value;
                        var base64Data =
match.Groups["data"].Value;
                        var bytes =
Convert.FromBase64String(base64Data);
```

```
                        using var sha256 =
System.Security.Cryptography.SHA256.Create();
                        var hash = sha256.ComputeHash(bytes);
                        var hashString =
BitConverter.ToString(hash).Replace("-", "").ToLowerInvariant();

                        var fileName = $"{hashString}.{ext}";
                        var filePath = Path.Combine("wwwroot",
"uploads", fileName);


Directory.CreateDirectory(Path.GetDirectoryName(filePath)!);

                        if (!System.IO.File.Exists(filePath))
                        {
                            await
System.IO.File.WriteAllBytesAsync(filePath, bytes);
                        }
                        url =
$"https://localhost:7232/uploads/{fileName}";
                    }
                }

                var existingAttachment = await
_context.Attachments.FirstOrDefaultAsync(a => a.Url == url);
                Attachment attachment;
                if (existingAttachment != null)
                {
                    attachment = existingAttachment;
                }
                else
                {
                    attachment = new Attachment { Url = url };
                    _context.Attachments.Add(attachment);
                    await _context.SaveChangesAsync();
                }
                attachments.Add(attachment);
            }

            var postAttachments = attachments.Select(attachment =>
new PostAttachment
            {
                Attachment = attachment,
                PostId = newPost.Id
            }).ToList();

            _context.PostAttachments.AddRange(postAttachments);
```

```
            await _context.SaveChangesAsync();
        }

        return Ok(new { message = "Post created successfully." });
    }



    [HttpGet("{postId}/interactions")]
    public async Task<ActionResult<IEnumerable<Interaction>>>
GetPostInteractions(int postId)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");
        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }
        var postInteractions = await _context.UserPostInteractions
            .Where(i => i.Post.Id == postId && i.User.Id ==
userId)
            .ToListAsync();

        var interactions = postInteractions.Select(i =>
i.Type).ToList();

        return Ok(interactions);
    }

    [HttpPost("{postId}/interactions")]
    public async Task<IActionResult> AddInteractions(int postId,
[FromBody] List<string> interactions)
    {
        var userId =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value ??
"0");
        if (userId == 0)
        {
            return Unauthorized("User not found.");
        }

        var post = await _context.Posts
            .Include(p => p.UserPostInteractions)
            .FirstOrDefaultAsync(p => p.Id == postId);

        if (post == null)
```

```
        {
            return NotFound("Post not found.");
        }

        var existingInteractions = await
_context.UserPostInteractions
            .Where(i => i.PostId == postId && i.UserId == userId)
            .ToListAsync();

        if (existingInteractions.Any())
        {

_context.UserPostInteractions.RemoveRange(existingInteractions);
        }

        var newInteractions = interactions.Select(i => new
UserPostInteraction
        {
            UserId = userId,
            PostId = postId,
            Type = i
        }).ToList();

        await
_context.UserPostInteractions.AddRangeAsync(newInteractions);

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            return Conflict("Data was modified or deleted by
another process.");
        }

        return Ok(new{ message = "Interaction added
successfully."});
    }

    [HttpGet("search")]
    public async Task<ActionResult<IEnumerable<PostData>>>
SearchPosts(string query, int? limit=10)
    {
        if(query == null || query.Trim().Length < 3)
        {
```

```
            return BadRequest("Query must be at least 3 characters
long.");
        }

        if (limit <= 0)
        {
            return BadRequest("Limit must be greater than 0.");
        }

        var posts = await _context.Posts
            .Include(p => p.Creator)
            .Include(p => p.PostAttachments)
                .ThenInclude(pa => pa.Attachment)
            .Where(p => (p.Title != null &&
p.Title.Contains(query)) || (p.Text != null &&
p.Text.Contains(query)))
            .OrderBy(p => p.Id)
            .Take(limit ?? 1)
            .ToListAsync();

        var postsData = posts.Select(p => new PostData
        {
            Id = p.Id,
            Title = p.Title,
            Text = p.Text ?? string.Empty,
            Attachments = p.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
            ChatId = p.Chat?.Id ?? 0,
            Creator = _mapper.Map<UserPublic>(p.Creator),
            Interactions = p.UserPostInteractions.Select(upi =>
upi.Type).ToList()
        });

        return Ok(postsData);
    }
    [HttpGet("random")]
    public async Task<ActionResult<IEnumerable<PostData>>>
GetRandomPosts(int? limit = 10)
    {
        if (limit <= 0)
        {
            return BadRequest("Limit must be greater than 0.");
        }
        var posts = await _context.Posts
            .Include(p => p.Creator)
            .Include(p => p.PostAttachments)
                .ThenInclude(pa => pa.Attachment)
```

```
            .OrderBy(p => Guid.NewGuid())
            .Take(limit ?? 1)
            .ToListAsync();
        var postsData = posts.Select(p => new PostData
        {
            Id = p.Id,
            Title = p.Title,
            Text = p.Text ?? string.Empty,
            Attachments = p.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
            ChatId = p.Chat?.Id ?? 0,
            Creator = _mapper.Map<UserPublic>(p.Creator),
            Interactions = p.UserPostInteractions.Select(upi =>
upi.Type).ToList()
        });
        return Ok(postsData);
    }
}

public class PostData
{
    public int Id { get; set; }
    public string Title { get; set; } = null!;
    public string Text { get; set; } = null!;
    public List<string> Attachments { get; set; } = null!;
    public int ChatId { get; set; }
    public UserPublic Creator { get; set; } = null!;
    public List<string> Interactions { get; set; } = null!;
}

public class Interaction
{
    public int Id { get; set; }
    public string Type { get; set; } = null!;
}
```

TagsController.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Server.Data;

[ApiController]
[Route("api/[controller]")]
public class TagsController : ControllerBase
{
```

```
    private readonly SocialNetworkContext _context;
    private readonly IMapper _mapper;
    public TagsController(SocialNetworkContext context, IMapper
mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<string>>>
GetTags(string filter, int limit)
    {
        var tags = await _context.Tags
            .Where(t => t.Name.Contains(filter))
            .OrderByDescending(t => t.UsageCount)
            .Take(limit)
            .Select(t => t.Name)
            .ToListAsync();
        return Ok(tags);
    }
}
```

ДОДАТОК Б

UsersCntroller.cs

```
using AutoMapper;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Server.Data;
using Server.Models;
using System.Security.Claims;

namespace Server.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class UsersController : ControllerBase
    {
        private readonly SocialNetworkContext _context;
        private readonly IMapper _mapper;

        public    UsersController(SocialNetworkContext    context,
IMapper mapper)
        {
            _context = context;
            _mapper = mapper;
        }

        [HttpGet]
        public  async  Task<ActionResult<IEnumerable<UserPublic>>>
GetUsers(bool full)
        {
            if (full)
            {
                var userPublicFull = await _context.Users
                    .Select(user => new UserPublicFull
                    {
                        Id = user.Id,
                        Nickname = user.Nickname,
                        AvatarUrl = user.AvatarUrl ?? string.Empty,
                        Description    =    user.Description    ??
string.Empty,
                        FullName = user.FullName,
                        UpdatedAt = user.UpdatedAt,
                        CreatedAt = user.CreatedAt,
                        IsDeleted = user.IsDeleted,
                        Role = user.Role
```

```
                    })
                    .ToListAsync();

                return Ok(userPublicFull);
            }
            else
            {
                var usersPublic = await _context.Users
                    .Select(user => new UserPublic
                    {
                        Id = user.Id,
                        Nickname = user.Nickname,
                        AvatarUrl = user.AvatarUrl ?? string.Empty,
                        Description    =    user.Description    ??
string.Empty,
                        FullName = user.FullName
                    })
                    .ToListAsync();

                return Ok(usersPublic);
            }
        }

        [HttpGet("{id}")]
        public async Task<ActionResult<UserPublicFull>> GetUser(int
id)
        {
            var user = await _context.Users
                .Select(user => new UserPublicFull
                {
                    Id = user.Id,
                    Nickname = user.Nickname,
                    AvatarUrl = user.AvatarUrl ?? string.Empty,
                    Description = user.Description ?? string.Empty,
                    FullName = user.FullName,
                    UpdatedAt = user.UpdatedAt,
                    CreatedAt = user.CreatedAt,
                    IsDeleted = user.IsDeleted,
                    Role = user.Role
                })
                .FirstOrDefaultAsync(user => user.Id == id);
            if (user == null)
            {
                return NotFound();
            }
            return Ok(user);
```

```
        }

        [HttpGet("{userId}/posts")]
        public async Task<ActionResult<IEnumerable<PostData>>>
GetPosts(int userId)
        {
            var posts = await _context.Posts
                .Include(p => p.Creator)
                .Include(p => p.PostAttachments)
                    .ThenInclude(pa => pa.Attachment)
                .Where(p => p.CreatorId == userId)
                .ToListAsync();

            var postsData = posts.Select(p => new PostData
            {
                Id = p.Id,
                Title = p.Title,
                Text = p.Text ?? string.Empty,
                Attachments = p.PostAttachments.Select(a =>
a.Attachment.Url).ToList(),
                ChatId = p.Chat?.Id ?? 0,
                Creator = _mapper.Map<UserPublic>(p.Creator),
                Interactions = p.UserPostInteractions.Select(upi =>
upi.Type).ToList()
            });

            return Ok(postsData);
        }

        [Authorize(Policy = "AdministratorPolicy")]
        [HttpPost]
        public async Task<IActionResult> CreateUser(UserFull user)
        {
            if (await _context.Users.AnyAsync(u => u.Email ==
user.Email))
            {
                return BadRequest("A user with this email already
exists.");
            }
            else if (await _context.Users.AnyAsync(u => u.Nickname
== user.Nickname))
            {
                return BadRequest("A user with this nickname
already exists.");
            }

            var newUser = new User
```

```
            {
                Nickname = user.Nickname,
                AvatarUrl = user.AvatarUrl,
                Description = user.Description,
                FullName = user.FullName,
                Email = user.Email,
                Password                                        =
BCrypt.Net.BCrypt.HashPassword(user.Password),
                CreatedAt = DateTime.UtcNow,
                UpdatedAt = DateTime.UtcNow,
                IsDeleted = false,
                Role = user.Role
            };

            _context.Users.Add(newUser);
            await _context.SaveChangesAsync();
            return Ok();
        }

        [Authorize(Policy = "AdministratorPolicy")]
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteUser(int id)
        {
            var user = await _context.Users.FindAsync(id);
            if (user == null)
            {
                return NotFound();
            }
            user.IsDeleted = true;
            await _context.SaveChangesAsync();
            return NoContent();
        }

        [HttpGet("search")]
        public  async  Task<ActionResult<IEnumerable<UserPublic>>>
SearchUsers(string query, int? limit=10)
        {
            if (query == null || query.Trim().Length < 3)
            {
                return  BadRequest("Query  must  be  at  least  3
characters long.");
            }
            if (limit <= 0)
            {
                return BadRequest("Limit must be greater than 0.");
            }
```

```
            Console.WriteLine("Hello");

            var users = await _context.Users
                .Where(user => user.Nickname.Contains(query) ||
user.FullName.Contains(query) || user.Description.Contains(query))
                .OrderBy(user => user.Id)
                .Select(user => new UserPublic
                {
                    Id = user.Id,
                    Nickname = user.Nickname,
                    AvatarUrl = user.AvatarUrl ?? string.Empty,
                    Description = user.Description ?? string.Empty,
                    FullName = user.FullName
                })
                .Take(limit ?? 1)
                .ToListAsync();

            return Ok(users);
        }

        [HttpGet("random")]
        public async Task<ActionResult<IEnumerable<UserPublic>>>
GetRandomUsers(int limit = 10)
        {
            if (limit <= 0)
            {
                return BadRequest("Count must be greater than 0.");
            }
            var randomUsers = await _context.Users
                .OrderBy(u => Guid.NewGuid())
                .Take(limit)
                .Select(user => new UserPublic
                {
                    Id = user.Id,
                    Nickname = user.Nickname,
                    AvatarUrl = user.AvatarUrl ?? string.Empty,
                    Description = user.Description ?? string.Empty,
                    FullName = user.FullName
                })
                .ToListAsync();
            return Ok(randomUsers);
        }
        [Authorize]
        [HttpPatch("{id}/follow")]
        public async Task<IActionResult> FollowUser(int id,
[FromBody] bool state)
```

```
        {
            var                         userId                          =
int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);
            if (userId == id)
            {
                return BadRequest("You cannot follow yourself.");
            }

            var user = await _context.Users.FindAsync(userId);
            if (user == null)
            {
                return NotFound("User not found.");
            }
            var targetUser = await _context.Users.FindAsync(id);
            if (targetUser == null)
            {
                return NotFound("Target user not found.");
            }

            var existingFollower = await _context.UserFollowers
                .FirstOrDefaultAsync(f => f.FollowerId == userId &&
f.UserId == id);

            if (state)
            {
                if (existingFollower == null)
                {
                    var follower = new UserFollower
                    {
                        FollowerId = userId,
                        UserId = id
                    };
                    _context.UserFollowers.Add(follower);
                    await _context.SaveChangesAsync();
                    return Ok("Subscribed successfully");
                }
                else
                {
                    return Ok("Already subscribed");
                }
            }
            else
            {
                if (existingFollower != null)
                {
```

```
_context.UserFollowers.Remove(existingFollower);
                    await _context.SaveChangesAsync();
                    return Ok("Unsubscribed successfully");
                }
                else
                {
                    return Ok("Not subscribed");
                }
            }
        }
    }
    public class UserPublicFull : UserPublic
    {
        public DateTime UpdatedAt { get; set; }
        public DateTime CreatedAt { get; set; }
        public bool IsDeleted { get; set; }
        public string Role { get; set; } = null!;
        public List<Relation> Relations { get; set; } = new
List<Relation>();
    }

    public class UserPublic
    {
        public int Id { get; set; }
        public string Nickname { get; set; } = null!;
        public string AvatarUrl { get; set; } = null!;
        public string? Description { get; set; } = null!;
        public string FullName { get; set; } = null!;
    }

    public class UserFull : UserPublicFull
    {
        public string Password { get; set; } = null!;
        public string Email { get; set; } = null!;
    }

    public class Relation
    {
        public string Type { get; set; } = null!;
        public int Count { get; set; }
    }
}
```

TokenService.cs

```
using Microsoft.IdentityModel.Tokens;
using Server.Models;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

namespace Server.Services
{
    public class TokenService
    {
        private readonly IConfiguration _configuration;
        private readonly TokenValidationParameters
_tokenValidationParameters;

        public TokenService(IConfiguration configuration)
        {
            _configuration = configuration;

            var key =
Encoding.ASCII.GetBytes(_configuration["Jwt:Key"]);
            _tokenValidationParameters = new
TokenValidationParameters
            {
                ValidateIssuer = true,
                ValidateAudience = true,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                ValidIssuer = _configuration["Jwt:Issuer"],
                ValidAudience = _configuration["Jwt:Audience"],
                IssuerSigningKey = new SymmetricSecurityKey(key)
            };
        }

        public string GenerateJwtToken(User user)
        {

            var tokenHandler = new JwtSecurityTokenHandler();
            var key =
Encoding.ASCII.GetBytes(_configuration["Jwt:Key"]);

            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.NameIdentifier,
user.Id.ToString()),
                new Claim(ClaimTypes.Name, user.Nickname),
                new Claim(ClaimTypes.Email, user.Email)
            };
```

```
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(claims),
                Expires = DateTime.UtcNow.AddDays(7),
                SigningCredentials = new SigningCredentials(new
SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature),
                Issuer = _configuration["Jwt:Issuer"],
                Audience = _configuration["Jwt:Audience"]
            };

            var token = tokenHandler.CreateToken(tokenDescriptor);
            return tokenHandler.WriteToken(token);
        }

        public                          TokenValidationParameters
GetTokenValidationParameters()
        {
            return _tokenValidationParameters;
        }

        internal ClaimsPrincipal ValidateJwtToken(string token)
        {
            var tokenHandler = new JwtSecurityTokenHandler();
            var   principal   =   tokenHandler.ValidateToken(token,
_tokenValidationParameters, out SecurityToken validatedToken);
            if (validatedToken is JwtSecurityToken jwtToken)
            {
                if
(!jwtToken.Header.Alg.Equals(SecurityAlgorithms.HmacSha256,
StringComparison.InvariantCultureIgnoreCase))
                    throw   new   SecurityTokenException("Invalid
token");
            }
            return principal;
        }
    }
}
```

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»; що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації; що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів; що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)