Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Комп'ютерних наук і технологій
(повне найменування факультету)

Комп'ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА ВЕБСИСТЕМИ ДЛЯ ПОШУКУ ДРУЗІВ ЗА ІНТЕРЕСАМИ

Виконав(ла): студент(ка) 4 курсу,
групи КНТ-521

Спеціальності
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

КОВАЛЬ М.С.
(ПРІЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)
Рецензент СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра комп'ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
_____
«14» квітня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

КОВАЛЬ Марини Сергіївни
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка вебсистеми для пошуку друзів за інтересами

керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ С.Ю.,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01.06.2025 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, технології розробки клієнтської та серверної частини

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Визначення технологій розробки;
2) Розробка вимог до програмного забезпечення;
3) Моделювання програмного забезпечення;
4) Розробка системи;
5) Випробування комп'ютерної системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	СКРУПСЬКИЙ С.Ю.		
Нормоконтроль	ПОЛЬСЬКА О.В.		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	**Примітка**
1	Визначення технологій розробки	до 18.04.2025	
2	Визначення та аналіз вимог до програмного	до 22.04.2025	
	забезпечення		
3	Розробка специфікації вимог SRS	до 24.04.2025	
4	Розробка діаграми варіантів використання	до 28.04.2025	
5	Розробка діаграми послідовності	до 01.05.2025	
6	Розробка описів прецедентів	до 05.05.2025	
7	Розробка системи	до 12.05.2025	
8	Розробка користувацького інтерфейсу	до 20.05.2025	
9	Випробування комп’ютерної системи	до 22.05.2025	
10	Оформлення пояснювальної записки	до 25.05.2025	
11	Проходження нормоконтролю	до 01.06.2025	
12	Перевірка на наявність академічного плагіату	до 03.06.2025	
13	Проходження рецензування	до 10.06.2025	

Студент(ка) _____ Марина КОВАЛЬ
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Степан СКРУПСЬКИЙ
(підпис) (Ім’я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 107 с., 7 табл., 14 рис., 1 дод., 26 джерел.

ВЕБСИСТЕМА, ПОШУК ДРУЗІВ, КЛІЄНТ, СЕРВЕР, NEXT.JS, NEST.JS, TYPEORM, POSTGRESQL,

Об'єкт розробки – веб-система для пошуку друзів за інтересами.

Мета роботи – розробка веб-системи для пошуку друзів за інтересами для автоматизації процесу знайомства та спілкування людей зі схожими захопленнями та інтересами.

В ході розробки було проаналізовано технічне завдання та проведено дослідження існуючих рішень. Здійснено порівняльний аналіз архітектурних підходів, включаючи одношарову, клієнт-серверну, тришарову та мікросервісну архітектури. Проведено вибір оптимальних технологій для розробки клієнтської та серверної частини системи, для визначення найбільш підходящого рішення.

Проведено проєктування веб-системи, в результаті якого було сформовано базу даних, визначено алгоритм роботи системи та розроблено її архітектуру. Створено діаграми варіантів використання, діаграми послідовності та розгорнуті описи прецедентів для моделювання програмного забезпечення. Розроблено специфікацію вимог SRS з детальним описом функціональних та нефункціональних вимог до системи.

В результаті розроблена веб-система має можливість створення користувацьких профілів з інформацією про інтереси, пошуку користувачів за схожими захопленнями та надсилання заявок на дружбу. Клієнтська частина має сучасний та зручний інтерфейс з власним дизайном, що забезпечує комфортну взаємодію користувачів з системою.

ЗМІСТ

Вступ 6
1 Визначення технологій розробки 7
1.1 Вибір архітектури системи 7
1.2 Вибір технологій для клієнської та серверної частини 13
1.3 Вибір бази даних 21
1.4 Висновок щодо вибору технологій та бази даних 25
2 Розробка вимог та моделювання програмного забезпечення 26
2.1 Визначення та аналіз вимог до програмного забезпечення 26
2.2 Розробка специфікації вимог SRS 26
2.3 Діаграми варіантів використання 33
2.4 Діаграми послідовності 35
2.5 Розгорнуті описи прецедентів 39
3 Розробка системи 43
3.1 Розробка клієнтської частини вебсистеми 43
3.2 Розробка серверної частини вебсистеми 65
4 Випробування комп'ютерної системи 81
4.1 Випробування процесу реєстрації 81
4.2 Випробування створення профілю 83
4.3 Випробування системи пошуку 85
4.4 Випробування системи дружби 85
Висновки 87
Перелік джерел посилання 89
Додаток А 92

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у трансформації соціальних взаємодій, створюючи нові можливості для спілкування, співпраці та самовираження. Соціальні мережі, онлайн–платформи та веб–сервіси стали невід'ємною частиною життя мільйонів людей, дозволяючи їм обмінюватися інформацією, знаходити однодумців і будувати спільноти на основі спільних інтересів. У контексті глобалізації та цифровізації особливої актуальності набувають системи, які допомагають користувачам налагоджувати зв'язки не лише на локальному, а й на міжнародному рівні. Одним із перспективних напрямів є розробка веб–систем, які забезпечують пошук друзів за інтересами, сприяючи формуванню міцних і значущих соціальних зв'язків.

Розробка веб-системи для пошуку друзів за інтересами є відповіддю на сучасні виклики, пов'язані з потребою в персоналізованих і зручних інструментах для соціалізації. У світі, де люди все частіше відчувають потребу в якісному спілкуванні та підтримці, подібні платформи допомагають долати географічні, культурні та соціальні бар'єри. Така система дозволяє користувачам знаходити однодумців для спільних хобі, професійної діяльності, творчих проєктів чи просто дружнього спілкування, що сприяє підвищенню якості життя та розвитку міжособистісних стосунків. Актуальність теми дипломної роботи зумовлена не лише зростаючим попитом на інноваційні соціальні технології.

Дипломна робота передбачає аналіз потреб цільової аудиторії, дослідження існуючих рішень, визначення функціональних і нефункціональних вимог до системи, проєктування архітектури, розробку інтерфейсу користувача та перевірку працездатності прототипу. Основна мета роботи полягає у створенні зручної, безпечної та адаптивної веб–системи, яка відповідатиме сучасним стандартам веб–розробки, враховуватиме специфіку взаємодії користувачів і забезпечуватиме якісний користувацький досвід.

1 ВИЗНАЧЕННЯ ТЕХНОЛОГІЙ РОЗРОБКИ

1.1 Вибір архітектури системи

Вибір належної архітектури системи є одним з найбільш відповідальних етапів у процесі розробки програмного забезпечення, особливо для вебсистем. Цей вибір формує фундаментальну структуру проєкту, визначаючи, як різні компоненти взаємодіятимуть між собою. По суті, архітектура − це креслення, що скеровує роботу команди розробників та забезпечує злагоджену роботу всіх частин системи . Неправильно обрана архітектура може призвести до значних труднощів у майбутньому, зокрема обмежень у масштабованості, проблем з підтримкою та оновленнями, а також потенційних вразливостей у безпеці [1]. Навпаки, добре продумана архітектура закладає міцний фундамент для успіху, забезпечуючи ефективність, надійність та можливість адаптації до майбутніх змін і зростання навантаження.

Важливість архітектурного рішення важко переоцінити. Воно безпосередньо впливає на продуктивність вебсистеми, її здатність обробляти зростаючу кількість користувачів та даних (масштабованість), легкість внесення змін та виправлення помилок (підтримуваність), а також на загальний рівень безпеки [2]. Архітектура визначає поділ системи на логічні частини, їхні обов'язки та способи взаємодії, що робить систему більш зрозумілою та керованою. Крім того, вибір архітектури має враховувати такі чинники, як складність системи, очікуване навантаження, вимоги до безпеки та швидкість розробки.

Серед розмаїття архітектурних рішень, що застосовуються для побудови вебсистем, можна виділити такі поширені підходи:

– одношарова (монолітна);

– клієнт–серверна;

– тришарова (3–tier);

– мікросервісна.

1.1.1 Одношарова архітектура

Одношарова (монолітна) архітектура є одним із традиційних підходів, де всі функціональні компоненти вебдодатку, включаючи інтерфейс користувача (UI), бізнес–логіку та рівень доступу до бази даних, об'єднані в єдиний блок. Всі ці частини тісно взаємопов'язані, поділяють спільну кодову базу та, як правило, розгортаються як єдиний виконуваний файл або процес [3]. Ця структура може бути привабливою для початкових етапів розробки або для невеликих проєктів завдяки своїй простоті розгортання та відносно легкій розробці порівняно з більш розподіленими архітектурами . У початкових фазах розробки моноліт дозволяє швидко створити працездатний прототип або мінімально життєздатний продукт . Тестування в монолітній системі може бути менш складним, оскільки всі компоненти знаходяться в одному середовищі . Також вважається, що монолітна архітектура може забезпечити дещо вищу продуктивність для невеликих застосунків завдяки відсутності накладних витрат на мережеву взаємодію між компонентами . Безпека в моноліті може бути централізованою, що спрощує її реалізацію на початковому етапі.

Однак, зі зростанням функціоналу та кількості користувачів, моноліт стає громіздким, що ускладнює його масштабування та подальшу підтримку. Розмір кодової бази зростає, що робить її важчою для розуміння новими членами команди та збільшує ризик внесення помилок. Навіть незначна зміна в одному модулі може вимагати перезбирання та повторного розгортання всього застосунку, збільшуючи ризик внесення помилок та тривалість цього процесу [4]. Масштабування моноліту часто означає масштабування всього застосунку шляхом додавання нових екземплярів, що може бути неефективним з точки зору використання ресурсів, якщо збільшення навантаження стосується лише окремих функцій або модулів. Крім того, через тісні зв'язки між компонентами, збій в одній частині монолітного застосунку може призвести до повної непрацездатності всієї системи. Використання єдиного технологічного стеку для всього застосунку в монолітній архітектурі також може обмежувати гнучкість та можливість інтеграції нових технологій або фреймворків, які могли б бути оптимальнішими

для окремих частин системи.

1.1.2 Клієнт–серверна архітектура

Клієнт–серверна архітектура є фундаментальною та найбільш поширеною моделлю для побудови переважної більшості вебзастосунків, що визначає спосіб взаємодії між двома основними компонентами системи: клієнтом та сервером. Цей архітектурний стиль базується на принципі розподілу ролей, де клієнт ініціює запити на отримання послуг чи ресурсів, а сервер ці запити обробляє та надає відповіді [5].

Клієнтська сторона – це зазвичай програмне забезпечення, яке працює на пристрої кінцевого користувача, найчастіше у веббраузері, але це також можуть бути мобільні застосунки чи інші типи клієнтів. Основна функція клієнта полягає у забезпеченні інтерфейсу користувача, зборі даних введених користувачем, формуванні запитів до сервера та відображенні отриманих від нього даних. Клієнт відповідає за взаємодію з користувачем та презентацію інформації. Частина логіки застосунку, пов'язана з візуалізацією та інтерактивністю інтерфейсу, також може виконуватися на клієнті, особливо в сучасних односторінкових застосунках (SPA).

Серверна сторона – це потужніший компонент системи, що працює на віддалених серверах і відповідає за централізоване зберігання та обробку даних, виконання основної бізнес–логіки та управління ресурсами. Коли сервер отримує запит від клієнта, він його аналізує, виконує необхідні операції (наприклад, запити до бази даних, обчислення, перевірку прав доступу) та формує відповідь, яка надсилається назад клієнту. Сервер відіграє ключову роль у забезпеченні безпеки, цілісності даних та доступності сервісів для багатьох клієнтів одночасно.

Взаємодія між клієнтом і сервером у вебсередовищі зазвичай відбувається за допомогою стандартних протоколів, таких як HTTP (Hypertext Transfer Protocol) та його безпечна версія HTTPS. Клієнт надсилає HTTP–запити (GET, POST, PUT, DELETE тощо), а сервер відповідає HTTP–відповідями, що містять статус операції та, за необхідності, запитувані дані.

Цей архітектурний підхід має низку переваг. Він забезпечує чітке

розділення відповідальності, що спрощує розробку, тестування та підтримку кожної частини системи окремо. Завдяки централізації даних та основної логіки на сервері полегшується управління ними, резервне копіювання, забезпечення безпеки та узгодженості даних. Хоча масштабування може бути складнішим, ніж у мікросервісах, клієнт–серверна архітектура дозволяє масштабувати серверну частину, додаючи нові сервери для обробки зростаючого навантаження. Крім того, така модель забезпечує високу доступність, дозволяючи багатьом різним клієнтам з різних пристроїв та місць отримувати доступ до централізованих ресурсів та сервісів, які надає сервер [6].

Незважаючи на переваги, існують і потенційні недоліки, такі як залежність від стабільності мережевого з'єднання між клієнтом та сервером, а також потенційна можливість перевантаження сервера, якщо він не може впоратися з одночасними запитами від великої кількості клієнтів.

Клієнт–серверна архітектура є базовою моделлю, на якій будуються більш складні архітектурні патерни вебдодатків, такі як тришарова архітектура, де серверна частина далі структурується на логічні рівні, або мікросервісна архітектура, де функції сервера розподіляються між багатьма незалежними сервісами.

1.1.3 Тришарова архітектура

Тришарова (3–tier) архітектура є розвиненою формою клієнт–серверної моделі і є одним із класичних та широко вживаних патернів для побудови складних вебзастосунків. Вона передбачає логічне та, часто, фізичне розділення системи на три взаємодіючі рівні (шари): рівень представлення, рівень застосунку (або бізнес–логіки) та рівень даних [7]. Це розділення дозволяє чітко визначити відповідальність кожного компонента та сприяє кращій організованості коду.

Рівень представлення (Presentation Tier) є найвищим рівнем архітектури, з яким безпосередньо взаємодіє користувач. Його головне завдання – відображення інформації користувачеві та збір введених ним даних. Цей рівень включає користувацький інтерфейс, який у випадку вебсистеми реалізується за допомогою таких технологій, як HTML, CSS та JavaScript, що виконуються у веббраузері.

Рівень представлення відповідає за візуалізацію даних, валідацію вхідних даних на стороні клієнта та надсилання запитів до наступного рівня – рівня застосунку. Важливою особливістю є те, що цей рівень не містить значної бізнес–логіки та не взаємодіє напряму з рівнем даних, що підвищує його гнучкість та незалежність від змін у нижчих рівнях.

Рівень застосунку (Application Tier) або рівень бізнес–логіки (Business Logic Tier) розташовується між рівнем представлення та рівнем даних. Це "серце" застосунку, де зосереджена основна логіка його роботи. Цей рівень приймає запити від рівня представлення, обробляє їх відповідно до бізнес–правил, виконує необхідні обчислення, приймає рішення та координує взаємодію з рівнем даних. Рівень застосунку слугує посередником, який обробляє запити користувача, викликає відповідні функції для виконання завдань та готує дані для відображення. Реалізація цього рівня часто включає використання серверних мов програмування та відповідних фреймворків.

Рівень даних (Data Tier) або рівень доступу до даних (Data Access Layer) є найнижчим рівнем тришарової архітектури і відповідає за зберігання, управління та доступ до даних. Цей рівень зазвичай складається з системи управління базами даних (СУБД), яка може бути реляційною або нереляційною. Рівень даних отримує запити на збереження, оновлення, видалення або вибірку даних виключно від рівня застосунку; прямий доступ до рівня даних з рівня представлення заборонений. Така ізоляція забезпечує цілісність та безпеку даних, а також дозволяє змінювати тип сховища даних без суттєвих змін у логіці застосунку.

Взаємодія між рівнями відбувається лінійно: рівень представлення надсилає запити до рівня застосунку, який, за необхідності, взаємодіє з рівнем даних. Відповіді рухаються у зворотному напрямку: від рівня даних до рівня застосунку, а потім до рівня представлення. Таке чітке розділення та визначена послідовність взаємодії надають тришаровій архітектурі значних переваг, включаючи покращену підтримуваність (зміни в одному рівні мінімально впливають на інші), підвищену масштабованість (кожен рівень може масштабуватися незалежно

залежно від навантаження) та кращу організованість проєкту, що полегшує паралельну розробку різними командами [8].

1.1.4 Мікросервісна архітектура

Мікросервісна архітектура є сучасним архітектурним стилем, що набув значної популярності, особливо для великих та складних вебсистем, і полягає в декомпозиції монолітного застосунку на набір невеликих, автономних сервісів. Кожен такий сервіс реалізує певну, чітко визначену бізнес–функцію або можливість і може бути розроблений, протестований, розгорнутий та масштабований незалежно від інших сервісів. Сервіси в мікросервісній архітектурі є слабо зв'язаними і взаємодіють один з одним, як правило, через легкі механізми комунікації, такі як HTTP API (REST або gRPC) або системи обміну повідомленнями (наприклад, Kafka, RabbitMQ).

Цей архітектурний стиль характеризується кількома ключовими аспектами. Перш за все, це автономність кожного мікросервісу: він є самостійним, часто має власну кодову базу та сховище даних, і його внутрішня реалізація прихована від інших сервісів. По–друге, сервіси часто організуються навколо конкретних бізнес–можливостей або доменів, що відповідають функціональним частинам системи (наприклад, сервіс користувачів, каталог товарів, обробка замовлень). Важливою особливістю є незалежне розгортання, що дозволяє оновлювати або масштабувати окремі сервіси без впливу на решту системи, прискорюючи процеси доставки та впровадження нових функцій. Також мікросервіси підтримують технологічну різноманітність, надаючи командам гнучкість у виборі найбільш відповідних технологій та мов програмування для кожного конкретного сервісу [9].

Мікросервісна архітектура надає значні переваги, особливо для великих та динамічних проєктів. Вона забезпечує покращену масштабованість, дозволяючи масштабувати лише ті сервіси, які потребують додаткових ресурсів, що робить використання інфраструктури більш ефективним. Також підвищується стійкість до збоїв, оскільки відмова одного сервісу, як правило, не призводить до каскадного збою всієї системи завдяки ізоляції відмов. Мікросервіси сприяють

більшій гнучкості та прискоренню процесів розробки, оскільки невеликі, автономні команди можуть працювати над різними частинами системи паралельно, незалежно одна від одної. Для великих систем мікросервіси можуть спростити підтримуваність, оскільки кожен окремий сервіс має керований розмір та складність .

Однак, перехід до мікросервісної архітектури пов'язаний із певними викликами. Спостерігається збільшення загальної складності системи, що вимагає значних зусиль для управління численними незалежними компонентами, їх моніторингу, розгортання та оркестрації. Комунікація між сервісами відбувається по мережі, що може спричиняти затримки та вимагає впровадження складних механізмів для забезпечення надійності у розподіленому середовищі. Виникають складнощі із забезпеченням узгодженості даних, оскільки кожен сервіс може мати власне сховище даних, що потребує реалізації розподілених транзакцій або інших патернів для підтримки консистентності. Зрештою, керування мікросервісною інфраструктурою може призвести до збільшення операційних накладних витрат та вимагати високої експертизи в області DevOps [10].

Мікросервісна архітектура є оптимальним вибором для проєктів з високими вимогами до масштабованості, стійкості та швидкості розробки, особливо коли над системою працюють декілька незалежних команд. Вона дозволяє ефективно управляти складністю великих систем шляхом їх декомпозиції на більш керовані частини.

1.2 Вибір технологій для клієнської та серверної частини

Проєктування вебзастосунків неможливе без обґрунтованого вибору технологічного стеку, який охоплює як клієнтську, так і серверну частину. Від того, наскільки вдало обрані інструменти для кожного з рівнів, залежить не лише швидкість і зручність розробки, але й ефективність функціонування системи в

цілому, її масштабованість, безпечність і стабільність у продуктивному середовищі.

Клієнтська частина, або frontend, реалізує взаємодію з користувачем через графічний інтерфейс, відповідає за відображення даних, реакцію на події та передачу запитів до серверу. Основою клієнтської розробки залишаються HTML, CSS та JavaScript – універсальні технології, що визначають структуру, зовнішній вигляд і динаміку сторінки відповідно. Однак через зростання складності користувацьких інтерфейсів усе частіше застосовуються спеціалізовані бібліотеки та фреймворки, зокрема React, Angular і Vue.js. Окрему нішу займає Next.js – фреймворк, побудований на основі React, що поєднує серверний рендеринг із клієнтською динамікою, забезпечуючи швидке завантаження сторінок і покращене SEO.

Серверна частина, або backend, реалізує основну логіку застосунку, обробляє запити з клієнтського боку, керує доступом до баз даних і забезпечує виконання обчислювальних операцій. Серед популярних технологій для цього рівня можна виокремити Node.js, який дає змогу виконувати JavaScript на сервері, що є зручним у разі використання єдиної мови на обох кінцях застосунку. У PHP–екосистемі провідне місце займає Laravel, який реалізує архітектурний шаблон MVC і має зручні засоби для маршрутизації. У великих корпоративних системах часто використовується Spring Boot (Java), що забезпечує гнучку конфігурацію та широкі можливості інтеграції. У проєктах на Ruby зазвичай використовується Ruby on Rails, відомий своєю філософією швидкої розробки з мінімальною конфігурацією.

Таким чином, правильна комбінація технологій фронтенду та бекенду формує єдиний технологічний ланцюг, який визначає ефективність, надійність і довговічність вебзастосунку в реальному середовищі експлуатації.

1.2.1 React

React – це високопродуктивна JavaScript–бібліотека з відкритим вихідним кодом, розроблена фахівцями Meta (раніше Facebook), що спеціалізується на побудові сучасних та динамічних користувацьких інтерфейсів вебзастосунків.

Центральною ідеєю React є компонентно–орієнтований підхід, який дозволяє розробникам розділяти складний інтерфейс на невеликі, незалежні та повторно використовувані елементи – компоненти. Кожен компонент інкапсулює власну логіку та візуальне представлення, що значно спрощує процес розробки, структурування коду та подальшу підтримку великих проєктів [11].

Однією з ключових технологій, що забезпечують ефективність React, є використання Віртуального DOM. Замість прямої та потенційно ресурсомісткої маніпуляції з реальним DOM браузера, React створює його віртуальне представлення у пам'яті. При зміні стану застосунку оновлюється спочатку Віртуальний DOM, а потім React обчислює мінімальні зміни, необхідні для синхронізації реального DOM, що оптимізує процес візуалізації та підвищує швидкість роботи застосунків, особливо при частих оновленнях даних. Також у розробці на React активно використовується синтаксичне розширення JSX, що дозволяє інтегрувати HTML–подібний синтаксис безпосередньо у JavaScript–код, роблячи код компонентів більш читабельним та наочним.

Завдяки своїм архітектурним особливостям та механізмам роботи, React пропонує низку переваг, включаючи високу продуктивність інтерфейсу, можливість ефективного повторного використання розроблених компонентів та суттєве спрощення логіки оновлення користувацького інтерфейсу. Наявність великої та активної спільноти розробників та розгалуженої екосистеми додаткових інструментів та бібліотек також сприяє його популярності та полегшує вирішення типових завдань [12]. Ці якості роблять React вдалим вибором для розробки фронтенду вебсистеми пошуку друзів за інтересами, де важлива швидкість реакції інтерфейсу, динамічне відображення інформації та можливість створення багатофункціонального користувацького середовища.

1.2.2 Angular

Angular являє собою комплексний відкритий фреймворк для розробки односторінкових клієнтських застосунків (SPA), створений та підтримуваний командою інженерів Google. На відміну від бібліотек, що надають інструменти для вирішення окремих задач, Angular є повноцінною платформою, яка охоплює

значний спектр аспектів фронтенд–розробки, від структуризації коду до управління станом та взаємодії з сервером. Основним інструментом для написання коду в Angular є TypeScript – типованізований надмножина JavaScript, що підвищує надійність застосунків та спрощує їхню підтримку, особливо в контексті великих проєктів. Фреймворк будується на принципах компонентної архітектури, де інтерфейс користувача декомпозиціюється на незалежні та повторно використовувані компоненти, кожен з яких містить власний шаблон, стилі та логіку [13].

Ключові можливості Angular включають ефективні механізми зв'язування даних, зокрема двостороннє зв'язування, що автоматизує синхронізацію між моделлю даних та представленням в інтерфейсі користувача, спрощуючи розробку інтерактивних елементів. Важливою складовою є система впровадження залежностей (Dependency Injection), яка сприяє створенню модульного та легкотестованого коду, підвищуючи гнучкість та підтримуваність застосунку. Angular надає розвинений набір інструментів командного рядка (Angular CLI), що полегшує створення проєктів, компонентів та автоматизацію типових завдань розробки [14].

Серед переваг використання Angular виділяють його комплексність та наявність "з коробки" рішень для багатьох стандартних завдань, що може прискорити розробку порівняно з підходами, що вимагають інтеграції численних сторонніх бібліотек. Сувора структура фреймворку та використання TypeScript сприяють підтримці високої якості коду та полегшують командну роботу над великими та довгостроковими проєктами. Однак, Angular може мати вищу криву навчання для нових розробників через його багатогранність та використання специфічних концепцій. Завдяки своїй надійності та масштабованості, Angular часто обирають для побудови великих корпоративних застосунків.

1.2.3 Vue

Vue визначається як прогресивний JavaScript – фреймворк з відкритим вихідним кодом, що використовується для ефективної побудови користувацьких інтерфейсів вебзастосунків . Він розроблений з філософією поступового

впровадження, що надає розробникам гнучкість у його використанні: від інтеграції невеликих інтерактивних компонентів на існуючі сторінки до розробки повноцінних та складних односторінкових застосунків (SPA) [4.1]. Основною структурною одиницею у Vue є компонент, який інкапсулює власну логіку, візуальне представлення та стилі, дозволяючи будувати інтерфейси шляхом комбінування цих незалежних блоків .

Ключовими особливостями Vue, що сприяють його популярності, є високопродуктивна реактивна система та використання Віртуального DOM. Реактивна система дозволяє Vue автоматично відстежувати зміни у стані застосунку та синхронізувати їх з інтерфейсом користувача найбільш ефективним способом. Застосування Віртуального DOM, схожого на підхід в інших сучасних бібліотеках, оптимізує процес оновлення DOM, забезпечуючи швидку реакцію інтерфейсу на зміни даних [15]. Vue також відомий своєю простотою та зрозумілим синтаксисом, що полегшує процес вивчення та розробки.

Серед переваг Vue.js часто відзначають його зручність для вивчення та гнучкість, що дозволяє використовувати фреймворк у проєктах різного масштабу та інтегрувати його з іншими бібліотеками чи існуючими проєктами. Фреймворк забезпечує високу продуктивність та має досить компактний розмір, що позитивно впливає на швидкість завантаження застосунків. Хоча Vue може мати меншу екосистему та спільноту порівняно з найбільшими фреймворками, ці аспекти активно розвиваються [16]. Здатність швидко створювати інтерактивні інтерфейси та його гнучкість роблять Vue.js привабливим варіантом для реалізації фронтенду вебсистеми пошуку друзів за інтересами.

1.2.4 Next

Next.js є відкритим React–фреймворком, призначеним для побудовивисокопродуктивних вебзастосунків, оптимізованих для середовища промислової експлуатації. На відміну від використання чистої бібліотеки React лише для рендерингу користувацького інтерфейсу, Next.js надає додатковий функціонал та структуру, необхідні для розробки повноцінних вебдодатків. Фреймворк автоматично забезпечує такі важливі можливості, як пре–рендеринг

сторінок, що може здійснюватися у двох основних формах: Server–Side Rendering (SSR), де HTML генерується на сервері при кожному запиті, та Static Site Generation (SSG), де статичний HTML генерується під час збірки проєкту. Також Next.js реалізує файлову систему маршрутизації, що спрощує організацію URL–адрес та навігацію між сторінками [17].

Використання методів пре–рендерингу, таких як SSR та SSG, є однією з ключових переваг Next.js, оскільки це значно покращує продуктивність застосунку та його доступність для пошукових систем (SEO), забезпечуючи швидке завантаження контенту та можливість його індексації пошуковими роботами. Фреймворк також включає автоматичний розподіл коду (automatic code splitting), що гарантує завантаження лише того JavaScript–коду, який необхідний для поточної сторінки, зменшуючи початковий час завантаження. Крім того, Next.js підтримує створення API маршрутів безпосередньо в проєкті, що дозволяє реалізувати прості серверні функції або створити бекенд для застосунку в межах єдиної кодової бази [18].

Next.js сприяє підвищенню ефективності розробки завдяки своїм вбудованим можливостям, оптимізаціям та чіткій структурі. Він є універсальним, дозволяючи використовувати гібридний підхід, поєднуючи SSR, SSG та клієнтський рендеринг залежно від потреб конкретної сторінки. Фреймворк добре інтегрується з іншими бібліотеками та інструментами екосистеми React і підходить для розробки різноманітних типів вебзастосунків, від статичних сайтів до складних динамічних порталів. Ці властивості роблять Next.js потужним інструментом для створення фронтенду вебсистеми пошуку друзів за інтересами, забезпечуючи високу продуктивність, покращене SEO та зручність розробки.

1.2.5 Node

Node.js є відкритим середовищем виконання JavaScript, що дозволяє виконувати JS–код на стороні сервера, використовуючи високопродуктивний рушій V8. Його ключова особливість – подіїно–орієнтована архітектура з неблокуючим вводом/виводом, що забезпечує ефективну обробку великої кількості одночасних з'єднань, що є важливим для вебзастосунків з активною

взаємодією. Незважаючи на те, що Node.js сам по собі надає базові можливості для створення серверів, для побудови складних та масштабованих застосунків доцільно використовувати структурні фреймворки. Саме тут вступає в дію NestJS – прогресивний фреймворк для Node.js, побудований на основі TypeScript, який надає чітку структуру та архітектурні патерни, необхідні для розробки ефективних та масштабованих серверних застосунків [19].

NestJS пропонує архітектуру, натхненну Angular, що базується на концепціях модулів, контролерів та провайдерів і активно використовує впровадження залежностей (Dependency Injection), що сприяє високій модульності, легкості тестування та підтримки коду. Використання TypeScript підвищує якість та надійність розробки. Поєднання Node.js та NestJS надає значні переваги для бекенд–розробки: Node.js забезпечує високу продуктивність для I/O–інтенсивних завдань, тоді як NestJS гарантує структурованість, масштабованість та легкість підтримки коду, роблячи процес розробки більш передбачуваним та ефективним. Ця комбінація є потужним інструментом для створення надійних та розширюваних серверних рішень.

1.2.6 PHP

PHP є широко вживаною скриптовою мовою програмування з відкритим вихідним кодом, яка особливо адаптована для веброзробки. Вона виконується на стороні сервера і слугує для генерації динамічного вебконтенту, взаємодії з базами даних та реалізації основної логіки вебзастосунків. Завдяки своїй популярності та тривалій історії, PHP має велику спільноту та розгалужену екосистему інструментів та бібліотек. Незважаючи на можливість написання вебдодатків на чистому PHP, для розробки складніших та масштабованіших систем доцільно використовувати структурні фреймворки.

Одним із найбільш популярних фреймворків для PHP є Laravel, який надає надійну структуру та набір інструментів для прискорення та спрощення процесу розробки вебзастосунків. Laravel реалізує архітектурний шаблон Model–View–Controller (MVC), що сприяє чіткому розділенню відповідальності та покращує організацію коду [20]. Фреймворк включає численні вбудовані функції, такі як

об'єктно–реляційне відображення (ORM) Eloquent для зручної роботи з базами даних, систему маршрутизації, шаблонізатор Blade, а також засоби для аутентифікації, авторизації та кешування. Використання Laravel з PHP дозволяє значно підвищити швидкість розробки, покращити підтримуваність та масштабованість застосунків, спираючись на перевірені архітектурні патерни та велику кількість готових рішень.

1.2.7 Java

Java є потужною, об'єктно–орієнтованою мовою програмування загального призначення, яка тривалий час залишається одним із провідних інструментів для розробки надійних та масштабованих серверних і корпоративних застосунків. Завдяки своїй незалежності від платформи (принцип "Write Once, Run Anywhere"), високій продуктивності віртуальної машини (JVM) та розвиненій екосистемі, Java широко використовується для побудови складних бекенд–систем. Однак, розробка на чистій Java або з використанням базового Spring Framework може вимагати значного обсягу конфігурації та шаблонного коду.

Для спрощення та прискорення розробки на базі Spring Framework було створено фреймворк Spring Boot. Spring Boot надає "думку" (opinionated) платформу, що суттєво зменшує необхідність у ручній конфігурації завдяки механізмам автоконфігурації та наявності "стартових" залежностей, які автоматично підключають необхідні бібліотеки. Він дозволяє створювати самостійні (standalone) застосунки з вбудованими серверами (наприклад, Tomcat або Jetty), що спрощує процес розгортання, оскільки не вимагає встановлення зовнішнього вебсервера [21]. Використання Spring Boot у поєднанні з Java забезпечує швидкий старт проєкту, ефективне управління залежностями та можливість легко інтегрувати компоненти з широкої екосистеми Spring, що робить цю комбінацію особливо придатною для розробки корпоративних рішень та мікросервісів.

1.2.7 Ruby

Ruby – це динамічна, об'єктно–орієнтована мова програмування з відкритим вихідним кодом, яка здобула визнання завдяки своїй простоті синтаксису та

фокусу на продуктивності розробника. Вона підтримує множинні парадигми програмування і часто використовується для написання серверної логіки вебзастосунків. Філософія Ruby спрямована на створення коду, який легко читається та пишеться, що сприяє більш приємному та ефективному процесу розробки. Однак, для створення повноцінних вебдодатків на Ruby, як правило, використовуються спеціалізовані фреймворки, які надають необхідну структуру та готові компоненти.

Ruby on Rails (або просто Rails) є високопродуктивним вебфреймворком, написаним мовою Ruby, який значно прискорює процес розробки вебзастосунків. Rails базується на принципах Convention over Configuration (згоди замість конфігурації) та Don't Repeat Yourself (не повторюйся), що зменшує обсяг шаблонного коду та стандартизує структуру проєктів [22]. Фреймворк реалізує архітектурний шаблон Model–View–Controller (MVC) і надає багатий набір вбудованих компонентів, таких як ActiveRecord для роботи з базами даних, систему маршрутизації та шаблонізатор Blade. Використання Ruby on Rails з Ruby дозволяє розробникам швидко створювати функціональні веббеккенди, дотримуючись кращих практик, і користуватися перевагами великої екосистеми "гемів" (бібліотек) та активної спільноти.

1.3 Вибір бази даних

Зберігання й обробка даних є фундаментальними складовими будь–якого вебзастосунку. Для ефективної організації цієї частини системи використовуються системи керування базами даних, вибір яких залежить від характеру проєкту, обсягу інформації, що обробляється, та архітектурних вимог. У найзагальнішому поділі виділяють два основні типи: реляційні та нереляційні бази даних. Реляційні СКБД працюють на основі структурованих таблиць і забезпечують строгі схеми даних, підтримку транзакцій і складні SQL–запити, що

робить їх доцільними для застосунків зі стабільною структурою інформації. Натомість нереляційні рішення (NoSQL) краще підходять для систем із великою кількістю неструктурованих або змінних даних, оскільки пропонують більш гнучкі моделі зберігання та зручне масштабування.

У сучасному веброзробленні широке застосування мають такі бази даних, як PostgreSQL – потужна реляційна система з відкритим кодом, що підтримує складні типи даних і розширення; MongoDB – документоорієнтоване NoSQL–рішення, оптимізоване для гнучкої роботи з JSON–подібними структурами; Firebase, яка надає можливість синхронізованого оновлення даних у реальному часі, особливо корисна у динамічних клієнт–серверних взаємодіях; та SQLite – вбудована легка реляційна база, яка не потребує серверного оточення й часто використовується для невеликих або автономних вебсистем.

Таким чином, ефективне управління даними в межах вебзастосунку залежить від правильного вибору СКБД, який повинен відповідати як функціональним потребам системи, так і обмеженням середовища її розгортання.

1.3.1 PostgreSQL

PostgreSQL є потужною відкритою об'єктно–реляційною системою управління базами даних (ОРСУБД), яка широко використовується для зберігання та управління структурованими даними у вебзастосунках різного масштабу. Вона відома своєю надійністю, цілісністю даних та суворим дотриманням стандартів SQL. PostgreSQL підтримує повноцінні транзакції з властивостями ACID (Atomicity, Consistency, Isolation, Durability), що гарантує надійність операцій з даними навіть у разі збоїв системи. Крім традиційних реляційних можливостей, система також надає підтримку для складних типів даних, включаючи масиви, JSON/JSONB та інші.

Однією з ключових переваг PostgreSQL є його розширюваність, що дозволяє розробникам створювати власні функції, типи даних та оператори, адаптуючи базу даних під специфічні потреби проєкту. Система пропонує розвинені механізми індексації та ефективне управління паралельним доступом за допомогою Multi–Version Concurrency Control (MVCC), що забезпечує високу

продуктивність при одночасній роботі багатьох користувачів [23]. Будучи відкритим програмним забезпеченням, PostgreSQL не вимагає ліцензійних відрахувань, що робить його економічно вигідним рішенням, а наявність великої та активної спільноти гарантує постійний розвиток та підтримку.

1.3.2 MongoDB

MongoDB – це популярна відкрита документо–орієнтована база даних класу NoSQL, яка відрізняється від традиційних реляційних СУБД гнучкішим підходом до зберігання даних. Замість таблиць з фіксованою схемою, MongoDB зберігає дані у форматі, подібному до JSON, який називається BSON (Binary JSON). Це дозволяє зберігати документи зі змінним набором полів у межах однієї колекції, що суттєво спрощує роботу з даними, структура яких може змінюватися або відрізнятися для різних сутностей. Така динамічна схема надає розробникам значну гнучкість та прискорює ітерації під час розробки, особливо на ранніх етапах проєкту.

Ключовими перевагами MongoDB є її висока масштабованість та доступність. Система підтримує горизонтальне масштабування (sharding), дозволяючи розподіляти дані між багатьма серверами для обробки великих обсягів інформації та високого навантаження. Механізми реплікації (replica sets) забезпечують високу доступність та стійкість до збоїв шляхом зберігання копій даних на різних серверах. MongoDB добре підходить для застосунків, що вимагають швидкої розробки, обробки напівструктурованих даних, а також легкої масштабованості, хоча при роботі зі складними реляційними зв'язками або строгими транзакційними вимогами традиційні реляційні бази можуть мати переваги [24].

1.3.3 Firebase

Firebase – це комплексна платформа для розробки мобільних та вебзастосунків, розроблена компанією Google, яка функціонує як Backend–as–a–Service (BaaS). Вона надає розробникам широкий спектр керованих сервісів, усуваючи необхідність у створенні та підтримці значної частини серверної інфраструктури самостійно. До ключових сервісів Firebase належать NoSQL бази

даних (Realtime Database та Cloud Firestore), засоби для аутентифікації користувачів, хостинг статичних сайтів, хмарні функції (cервepless обчислення), а також інструменти для зберігання файлів та аналітики. Такий підхід дозволяє розробникам зосередитися переважно на логіці клієнтської частини застосунку.

Використання Firebase значно прискорює процес розробки завдяки наявності готових до використання компонентів та автоматизації багатьох бекенд–завдань [25]. Платформа спрощує реалізацію таких функцій, як аутентифікація користувачів та синхронізація даних у реальному часі, що особливо цінно для інтерактивних застосунків. Firebase розроблений для автоматичного масштабування, дозволяючи застосункам обробляти зростаюче навантаження без додаткових зусиль з боку розробників щодо управління інфраструктурою. Однак, при виборі Firebase слід враховувати потенційний "vendor lock–in", оскільки інтеграція з платформою Google може ускладнити міграцію на інші бекенд–рішення в майбутньому.

1.3.4 SQLite

SQLite є унікальною легковаговою реляційною системою управління базами даних, реалізованою як бібліотека на мові C. Її фундаментальна відмінність від більшості інших СУБД полягає у серверній архітектурі та нульовій конфігурації: SQLite не потребує окремого серверного процесу, а натомість зберігає всю базу даних в єдиному файлі на диску, з яким взаємодіє безпосередньо застосунок, що її використовує. Така архітектура робить SQLite надзвичайно простою у розгортанні та використанні, усуваючи необхідність в адмініструванні бази даних. SQLite підтримує значну частину стандарту SQL та є ACID–сумісною, що гарантує надійність транзакцій.

Ключовими перевагами SQLite, що випливають з її дизайну, є її портативність (база даних – це просто файл, який легко копіювати та переміщувати), мінімальні системні вимоги та придатність для вбудованих систем, мобільних та десктопних застосунків, а також для невеликих вебсайтів або прототипів. Вона ідеально підходить для випадків, коли база даних потрібна

локально в рамках одного застосунку і не вимагає високопродуктивної обробки численних одночасних записів з багатьох мережевих клієнтів [26].

1.4 Висновок щодо вибору технологій та бази даних

У процесі розробки веб–застосунку для пошуку друзів за інтересами було визначено ключові технічні пріоритети: швидка реалізація мінімально життєздатного продукту (MVP), стабільна робота з даними, зручність масштабування та підтримки у майбутньому. Враховуючи ці вимоги, було прийнято рішення використовувати сучасний стек JavaScript/TypeScript–технологій із чітким розподілом клієнтської та серверної логіки.

Клієнтська частина реалізована з використанням Next.js – фреймворку, що базується на React і підтримує як клієнтський, так і серверний рендеринг. Такий підхід забезпечує високу продуктивність інтерфейсу, кращу SEO–оптимізацію та зручну організацію маршрутизації. Завдяки можливості використання API–роутів та модульної структури, Next.js сприяє швидкому розгортанню функціоналу та гнучкому масштабуванню.

Серверна частина побудована на базі Nest.js – сучасного фреймворку для Node.js, який забезпечує чисту архітектуру, підтримку TypeScript, вбудовану систему валідації, логування та маршрутизацію REST–запитів. Завдяки модульному підходу Nest.js дозволяє ефективно організувати бізнес–логіку проєкту, що значно спрощує підтримку та тестування.

Як основну систему керування базами даних обрано PostgreSQL – потужну реляційну СУБД з відкритим вихідним кодом, яка підтримує транзакції, складні SQL–запити, зберігання JSON–даних та повнотекстовий пошук. У поєднанні з TypeORM – об’єктно–реляційним мапером для TypeScript – це рішення дозволяє зручно працювати з моделями даних, реалізовувати міграції та забезпечувати цілісність структури БД протягом усього життєвого циклу застосунку.

2 РОЗРОБКА ВИМОГ ТА МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Визначення та аналіз вимог до програмного забезпечення

На цьому етапі першочерговим завданням є встановлення необхідного рівня деталізації вимог та способу їх представлення.

Вимоги користувачів, які повинні бути реалізовані у веб-додатку для пошуку людей за інтересами:

– інтуїтивний інтерфейс для створення профілю, додавання інформації про свої інтереси та пошуку користувачів;

– гнучкі налаштування критеріїв пошуку (інтереси, географічне розташування, вік);

– система рекомендацій на основі спільних інтересів і взаємних уподобань;

– система верифікації для підтвердження справжності профілів.

Для обробки, зберігання та оптимізації персональних даних використовуються сучасні алгоритми шифрування та компресії для забезпечення високої швидкості доступу та захисту даних.

2.2 Розробка специфікації вимог SRS

2.2.1 Введення (Розділ 1 SRS)

2.2.1.1 Призначення (Підрозділ 1.1 SRS)

Цей документ визначає функціональні та нефункціональні вимоги до вебсистеми "MateUp", який призначений для пошуку людей за спільними інтересами. Застосунок надає можливість користувачам створювати профілі, шукати людей за фільтрами. Ця специфікація є основою для розробників, тестувальників і менеджерів проекту, щоб забезпечити узгодженість і якість

кінцевого продукту.

2.2.1.2 Область дії (Підрозділ 1.2 SRS)

Продукт забезпечує платформу для взаємодії між користувачами, які мають спільні інтереси. Можливості програми включають:

– створення персонального профілю;

– використання розширеного пошуку за інтересами, віком, місцезнаходженням та іншими критеріями;

– надсилання запитів на дружбу.

Ця вебсистема орієнтована на звичайних користувачів, які прагнуть знаходити друзів або колег за спільними захопленнями.

2.2.1.3 Визначення, акроніми та скорочення (Підрозділ 1.3 SRS)

Визначення:

Профіль користувача – персональна сторінка, де користувач вказує основну інформацію, уподобання та інтереси.

Матчинг – механізм, що знаходить людей із подібними інтересами.

Акроніми:

UI – User Interface, графічний інтерфейс користувача.

API – Application Programming Interface, програмний інтерфейс застосунків.

HTTPS – Hypertext Transfer Protocol Secure, протокол безпечного передавання даних.

DBMS – Database Management System, система керування базами даних.

2.2.1.4 Публікації (Підрозділ 1.4 SRS)

Закон України "Закон України Про захист персональних даних" – регулює відносини , пов'язані із захистом персональних даних під час їх обробки.

Стандарт ISO/IEC 26514:2022 – містить вимоги до проектування та розробки документації користувача програмного забезпечення як частини процесів життєвого циклу, визначає процес документування з точки зору розробника документації, визначає структуру, зміст і формат документації користувача, а також надає інформативні вказівки щодо стилю документації користувача, не залежить від програмних засобів, які можна використовувати для

створення документації, і застосовується як до друкованої документації, так і до документації на екрані.

Стандарт ISO/IEC 27001:2022 – встановлює вимоги до створення, впровадження, підтримки та постійного поліпшення системи менеджменту інформаційної безпеки в контексті організації. Він також включає в себе вимоги до оцінки і обробки ризиків інформаційної безпеки з урахуванням потреб організації.

2.2.1.5 Короткий огляд (Підрозділ 1.5 SRS)

Специфікація вимог програмного забезпечення складається з трьох основних розділів, кожен з яких розподілено на відповідні підрозділи, пронумеровані у логічній послідовності.

У першому розділі розглядаються основні аспекти призначення ПЗ, базові вимоги до системи, а також включено перелік необхідних визначень, скорочень, акронімів і посилання на використані нормативні документи.

Другий розділ містить загальний опис програмного продукту, який охоплює такі аспекти, як контекст використання, функціональність, існуючі обмеження, припущення та залежності, які можуть впливати на розробку чи експлуатацію продукту.

Третій розділ деталізує вимоги до системи, включаючи такі важливі аспекти, як її надійність, безпека, взаємодія з іншими системами, а також вимоги до інтерфейсу користувача та інших технічних характеристик.

2.2.2 Загальний опис (Розділ 2 SRS)

2.2.2.1 Контекст програмного продукту (Підрозділ 2.1 SRS)

Система є доступною через браузери на будь-якому пристрої. Вона забезпечує адаптивний інтерфейс, що зручно відображається як на мобільних пристроях, так і на великих екранах.

2.2.2.2 Функції програмного продукту (Підрозділ 2.2 SRS)

Система надає такі функції:

– створення та редагування профілів користувачів;

– додавання інтересів із запропонованого списку;

– пошук інших користувачів із подібними інтересами.

2.2.2.3 Характеристики користувача (Підрозділ 2.3 SRS)

Цільовою аудиторією є люди віком від 18 років, які мають базові навички користування інтернетом і зацікавлені у встановленні нових знайомств.

2.2.2.4 Обмеження (Підрозділ 2.4 SRS)

Для забезпечення коректної роботи вебсистеми важливо враховувати технічні параметри пристрою. Оскільки програма працює безпосередньо у веббраузері, вона є невибагливою до ресурсів комп'ютера чи мобільного пристрою та не потребує встановлення додаткового програмного забезпечення. Проте для уникнення затримок під час завантаження сторінок, відображення інтерактивних елементів або роботи з великими обсягами інформації, рекомендовано дотримуватися певних системних вимог.

З метою зручності системні вимоги умовно поділено на мінімальні та оптимальні, що наведено у таблицях 2.1 та 2.2 відповідно.

Таблиця 2.1 - Мінімальні вимоги апаратної частини

Компонент	**Характеристика**
Процесор (CPU)	2 ядра, 1.6 ГГц
Оперативна пам'ять (RAM)	2 ГБ
Дисплей	Будь-який екран з роздільною здатністю від 1024×768
Веб-браузер	Сучасний браузер (Chrome, Firefox, Edge, Safari - останні 2 версії)
Доступ до Інтернету	Стабільне з'єднання зі швидкістю від 10 Мбіт/с
Накопичувач	Не потребує встановлення (браузерна версія)

Таблиця 2.1 містить мінімальні характеристики, за яких система буде функціонувати стабільно, проте з певними обмеженнями. Зокрема, на пристроях з такими параметрами можлива триваліша обробка запитів, сповільнене відображення інтерактивних компонентів або часткове обмеження роботи зі складними інтерактивними елементами. Ці вимоги підходять для ознайомлення зі змістом сайту, роботи з простими формами чи виконання базових операцій.

Таблиця 2.2 - Оптимальні вимоги апаратної частини

Компонент	**Характеристика**
Процесор (CPU)	4 ядра, 2.0 ГГц і вище
Оперативна пам’ять (RAM)	4 ГБ і більше
Дисплей	Екран з роздільною здатністю Full HD (1920×1080)
Веб-браузер	Оновлений браузер останньої версії (Chrome, Firefox, Edge, Safari)версії)
Доступ до Інтернету	Надійне з’єднання від 25 Мбіт/с
Накопичувач	Не потребує встановлення (браузерна версія)

Таблиця 2.2 подає оптимальні параметри, які дозволяють максимально ефективно використовувати всі можливості вебзастосунку. Використання пристроїв із такими характеристиками забезпечує плавну анімацію, швидке завантаження сторінок, миттєву обробку запитів користувача, повну підтримку адаптивного інтерфейсу та безперебійну роботу навіть у багатозадачному режимі. Це особливо важливо під час роботи з великим обсягом даних, інтегрованими мультимедійними компонентами або при використанні системи на мобільних пристроях із високою роздільною здатністю екрана.

2.2.2.5 Допущення і залежності (Підрозділ 2.5 SRS)

Фактори, які можуть вплинути на вимоги:

– додавання нових функцій, пов'язаних із штучним інтелектом, таких як рекомендаційні системи для покращення пошуку або аналіз інтересів користувачів;

– необхідність оптимізації веб-додатку для роботи на нових пристроях або у веб-браузерах із специфічними технічними вимогами;

– вплив нових норм і правил захисту персональних даних на зберігання, обробку та передачу інформації про користувачів;

– оновлення стандартів щодо онлайн-безпеки та боротьби зі зловживаннями в інтернеті;

– оновлення технологій, які можуть впливати на продуктивність або енергоефективність веб-додатку.

2.2.2.6 Розподілення вимог (Підрозділ 2.6 SRS)

Вимоги, відкладені до майбутніх версій:

– інтеграція відеозв’язку;

– механізм оцінки користувачів;

– підтримка мультимовного інтерфейсу;

– опція створення бізнес-акаунтів для пошуку партнерів.

2.2.3 Специфічні вимоги (Розділ 3 SRS)

2.2.3.1 Функції інтерфейсу користувача

Після запуску вебдодатку користувач одразу потрапляє на стартову сторінку, де пропонується вибрати одну з двох опцій: «Зареєструватися» або «Увійти». Після реєстрації нового користувача відкривається вікно профілю в яких користувач заповнює інформацію про себе. Коли новий користувач заповнив всю інформацію профілю або користувач натиснув кнопку «Увійти» та після цього виконав авторизацію в відповідному вікні, відкривається головна сторінка вебсистеми.

Подальша робота з додатком відбувається на головній сторінці, де користувач має можливість виконати наступні функції:

– вікно рекомендованих користувачів: призначене для перегляду короткої інформації про користувачів, надсилання запиту дружби, початку чату відкриття профілю та перегляду детальної інформації;

– поле пошуку: дозволяє швидко знайти людей за ім’ям країною або містом;

– фільтри: для сортування користувачів за інтересами, віком;

– кнопка для швидкого доступу до налаштувань профілю: користувач може змінити налаштування або інформацію свого профілю;

– розділ друзів: користувач може переглядати або шукати своїх друзів або приймати або відхиляти запити на дружбу від інших людей;

– меню навігації: дозволяє швидко перейти до інших розділів додатку.

2.2.3.2 Атрибути якості програмної системи

2.2.3.2.1 Надійність

Вебсистема має бути спроектована з урахуванням високої продуктивності та масштабованості, особливо коли йдеться про обробку значних обсягів даних під час пошуку та фільтрації інформації за інтересами користувачів. Це дозволить забезпечити безперебійну та швидку роботу додатка, навіть за умови одночасного перебування великої кількості користувачів та при роботі з обширними масивами даних.

Крім того, важливо передбачити механізми швидкого відновлення роботи системи у випадку непередбачених збоїв або помилок. Користувач повинен мати можливість легко продовжити свою взаємодію з системою, наприклад, завдяки збереженню останніх пошукових запитів або історії взаємодії з іншими користувачами та групами. Це гарантуватиме, що прогрес не буде втрачено, навіть якщо доведеться перезавантажити сторінку або весь сеанс, забезпечуючи безшовний користувацький досвід.

2.2.3.2.2 Захист

Відповідно до українського законодавства, зокрема Закону України "Про захист персональних даних", наш веб-додаток зобов'язаний гарантувати повну конфіденційність та безпеку особистої інформації користувачів. Усі дані, зібрані під час реєстрації та подальшого використання системи, зберігатимуться виключно для внутрішніх цілей і ні за яких обставин не будуть передані третім особам без прямої згоди користувача, за винятком випадків, чітко визначених законом.

Для забезпечення такого рівня захисту, програмне забезпечення буде відповідати найсуворішим вимогам щодо захисту персональних даних. Це включає обов'язкове використання надійних методів шифрування для всієї конфіденційної інформації, а також впровадження комплексних заходів для запобігання будь-якому несанкціонованому доступу до цих даних. Ми прагнемо створити середовище, де кожен користувач почуватиметься впевнено щодо збереження своєї приватності.

2.2.3.2.3 Зручність супроводження

Вебсистема спроектована на основі сучасних веб-архітектурних шаблонів, що гарантує її легку підтримку та ефективне масштабування. Завдяки модульній структурі ми можемо безпечно та безперешкодно інтегрувати новий функціонал, а також вносити зміни та оновлення, не порушуючи стабільну роботу основних компонентів.

Такий підхід дозволяє нам швидко адаптувати додаток до нових технологій та мінливих вимог ринку, ефективно реагуючи на запити користувачів. При цьому ми підтримуємо високу стабільність та продуктивність веб-додатку, забезпечуючи безперервний та надійний користувацький досвід.

2.3 Діаграми варіантів використання

Діаграми прецедентів в UML слугують для візуалізації функціональних аспектів системи та уточнення її вимог. Ці діаграми ілюструють ключові можливості системи та окреслюють її сферу застосування. Також вони показують, як система взаємодіє із зовнішніми суб'єктами, відомими як актори. Прецеденти та актори на цих схемах описують, які завдання виконує система та яким чином її експлуатують, проте не заглиблюються в деталі її внутрішньої реалізації. Окрім того, діаграми прецедентів є ефективним засобом комунікації між командою розробників та усіма зацікавленими особами, забезпечуючи прозоре розуміння системних вимог. Вони дають змогу виявити можливі недоліки ще на початкових стадіях розробки. Більше того, такі діаграми сприяють визначенню пріоритетності функціоналу, що потребує першочергової реалізації. Це має особливе значення для масштабних проєктів, де критично важливо мати чітке уявлення про обсяг та межі створюваної системи.

Діаграми варіантів використання містять наступні елементи:

– випадок використання – визначає функцію, яку виконує система для

досягнення цілей користувача;

– актор – це роль користувача, який взаємодіє із системою; актором може бути людина, організація, пристрій або інша зовнішня система.

Діаграма варіантів використання вебсистеми для пошуку людей за інтересами зображена на рисунку 2.1.

В межах веб-додатку для пошуку друзів за інтересами було визначено вісім основних прецедентів, що на діаграмі позначені відповідними зв’язками включення. Стислий опис кожного прецедента:

– реєстрація користувача – описує процес створення нового облікового запису, що включає введення персональних даних, електронної пошти та пароля;

– авторизація користувача – описує процес входу до системи за допомогою раніше зареєстрованих облікових даних;

– створення профілю – описує можливість заповнення персонального профілю користувача з додаванням інформації про інтереси, фотографії та іншої важливої інформації;

– пошук людей за інтересами – описує функціонал для пошуку користувачів з подібними інтересами за допомогою ключових слів, фільтрів або рекомендованих результатів;

– надсилання запиту на дружбу – описує можливість ініціювання дружніх зв’язків між користувачами через відправлення запиту;

– відповідь на запит дружби – описує функціонал для прийняття або відхилення отриманого запиту на дружбу, з можливістю додавання користувача до списку друзів або надання відмови.

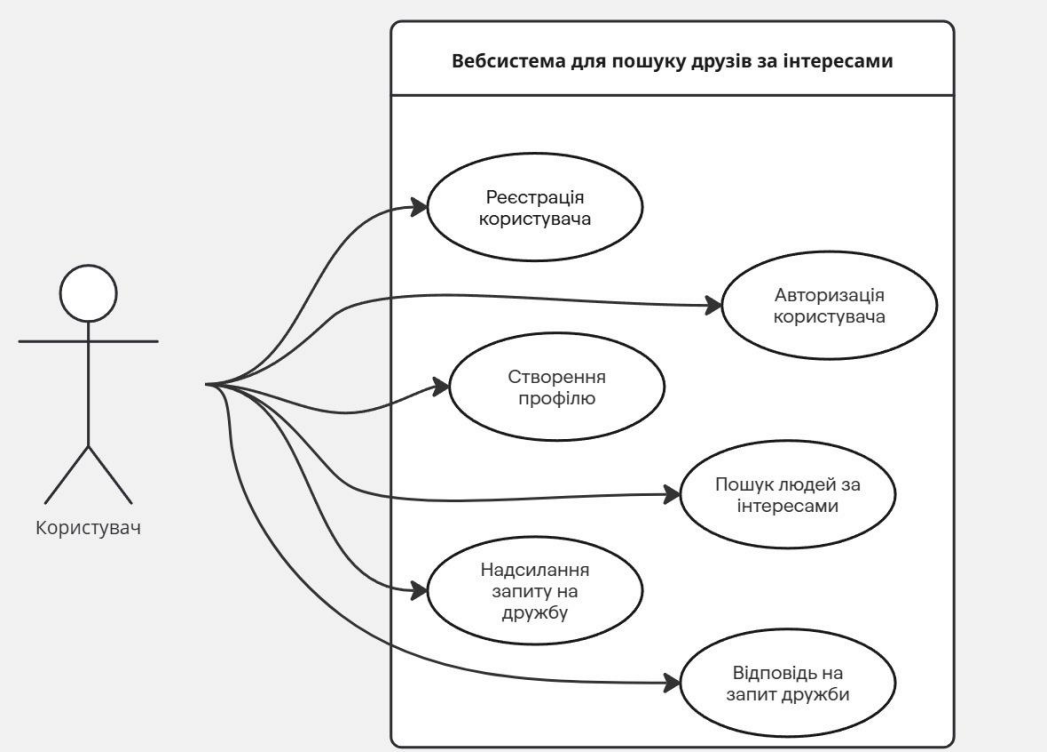


Рисунок 2.1 – Діаграма варіантів використання веб-додатку для пошуку людей за інтересами

2.4 Діаграми послідовності

Діаграма послідовності в контексті вебсистеми для пошуку друзів за інтересами слугує потужним інструментом для візуалізації динамічної поведінки системи. Вона детально ілюструє, як різні об'єкти системи співпрацюють між собою в часі, щоб досягти конкретних результатів, наприклад, знайти користувачів, об'єднаних спільними захопленнями.

Цінність діаграми послідовності полягає в її здатності перетворювати абстрактні функціональні вимоги, зафіксовані у варіантах використання, на конкретні сценарії взаємодії. Це дає змогу розробникам глибоко зануритися в логіку додатку, чітко визначити кроки, що виконуються користувачем, і паралельно описати внутрішні процеси системи.

Основне призначення діаграми послідовності для такої системи – це чітке визначення послідовності дій, які необхідні для реалізації цільових функцій, таких як реєстрація, авторизація або пошук. Діаграма акцентує увагу не тільки на змісті інформації, що передається між об'єктами, але й на її порядку, що створює структуроване уявлення про взаємодію системних компонентів.

Візуально діаграма послідовності організована за двома вимірами. Вертикальний вимір відображає часову вісь, показуючи повідомлення зверху вниз у міру їхнього виникнення. Горизонтальний вимір представляє екземпляри об'єктів (наприклад, користувач, база даних, сервер), які беруть участь у взаємодії, розташовуючи їх зліва направо в порядку надсилання та отримання повідомлень. Таке двовимірне представлення дозволяє легко відстежувати потік виконання та взаємозв'язки між елементами системи. У даному проекті є лише один актор та програма, з якою він взаємодіє.

У даному проекті є лише один актор та програма, з якою він взаємодіє. Таким чином, діаграми послідовності мають простий вигляд (див. рис. 2.2–2.6).

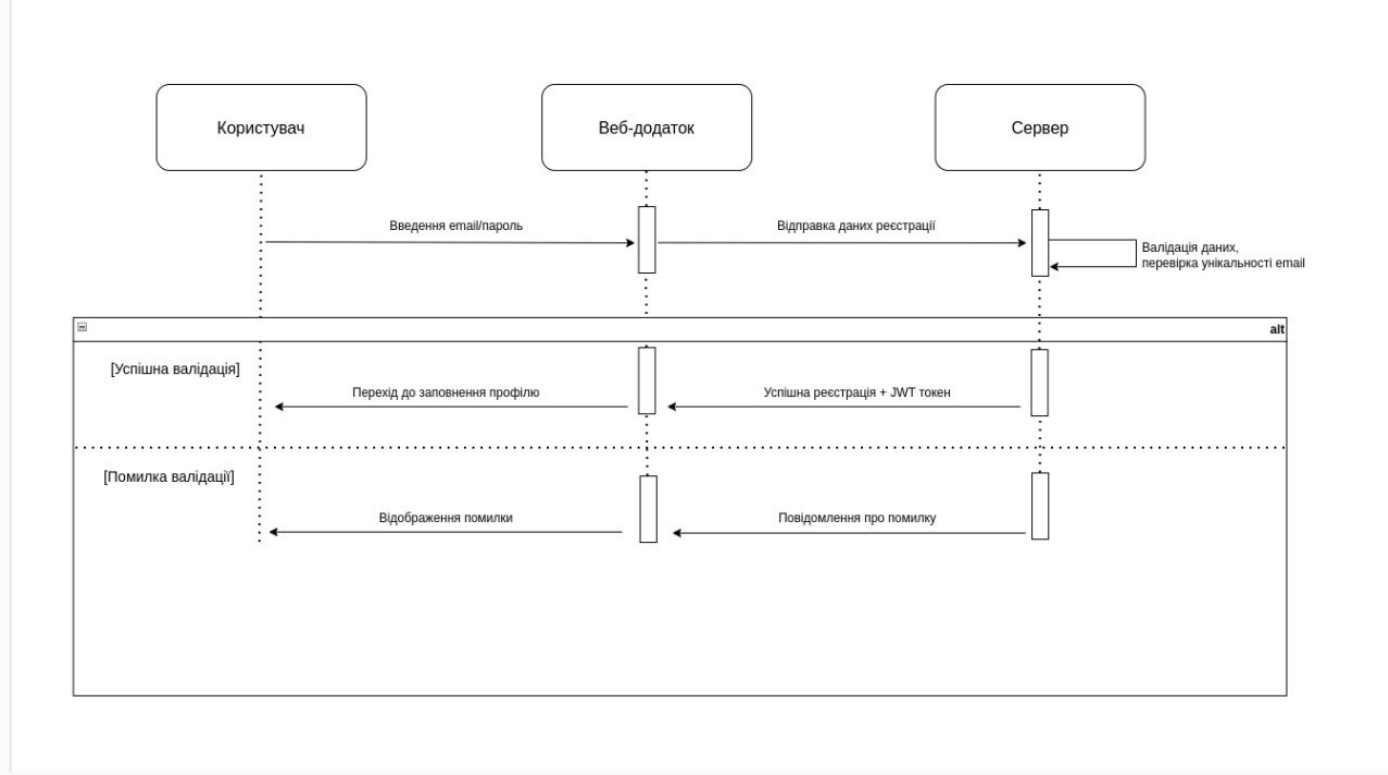


Рисунок 2.2 – Діаграма послідовності для прецеденту «Реєстрація користувача»

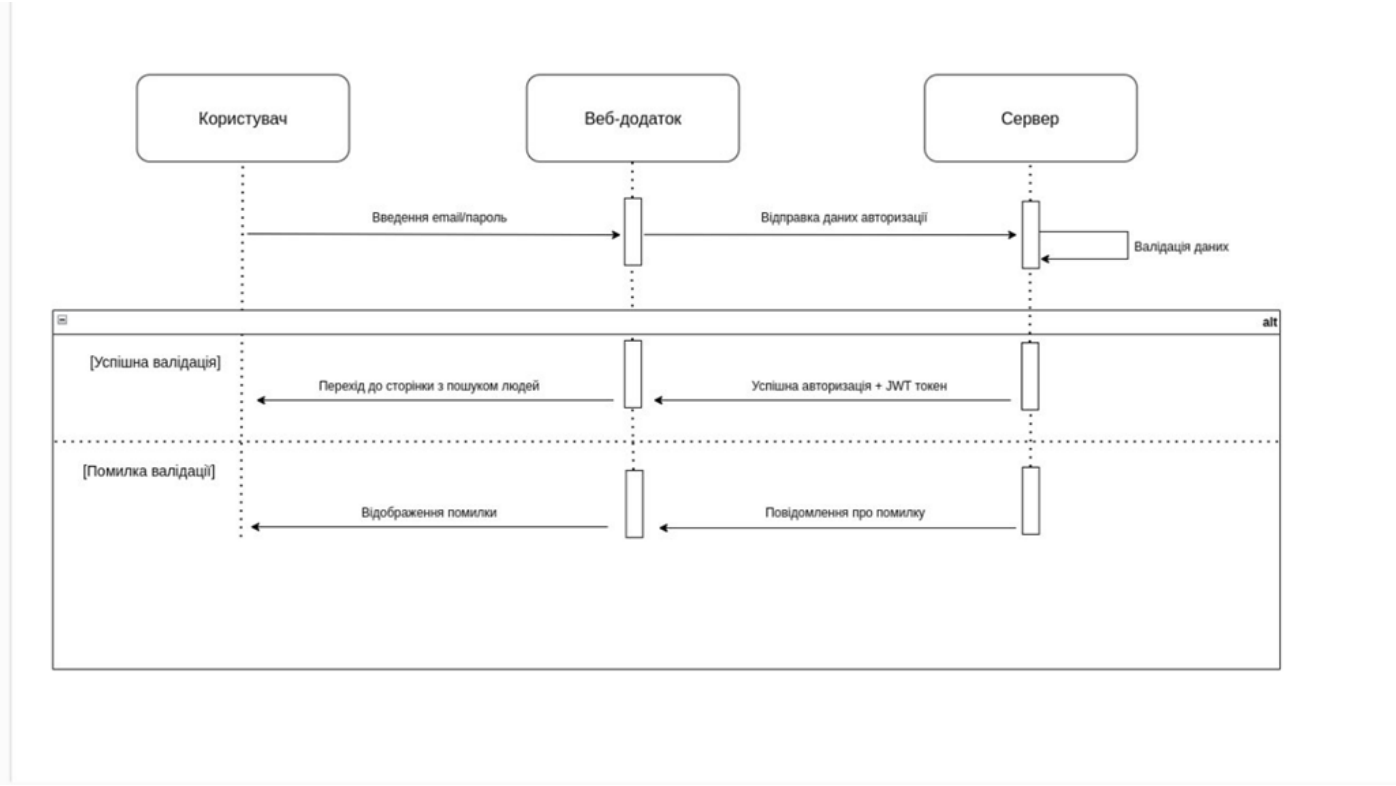


Рисунок 2.3 – Діаграма послідовності для прецеденту «Авторизація користувача»

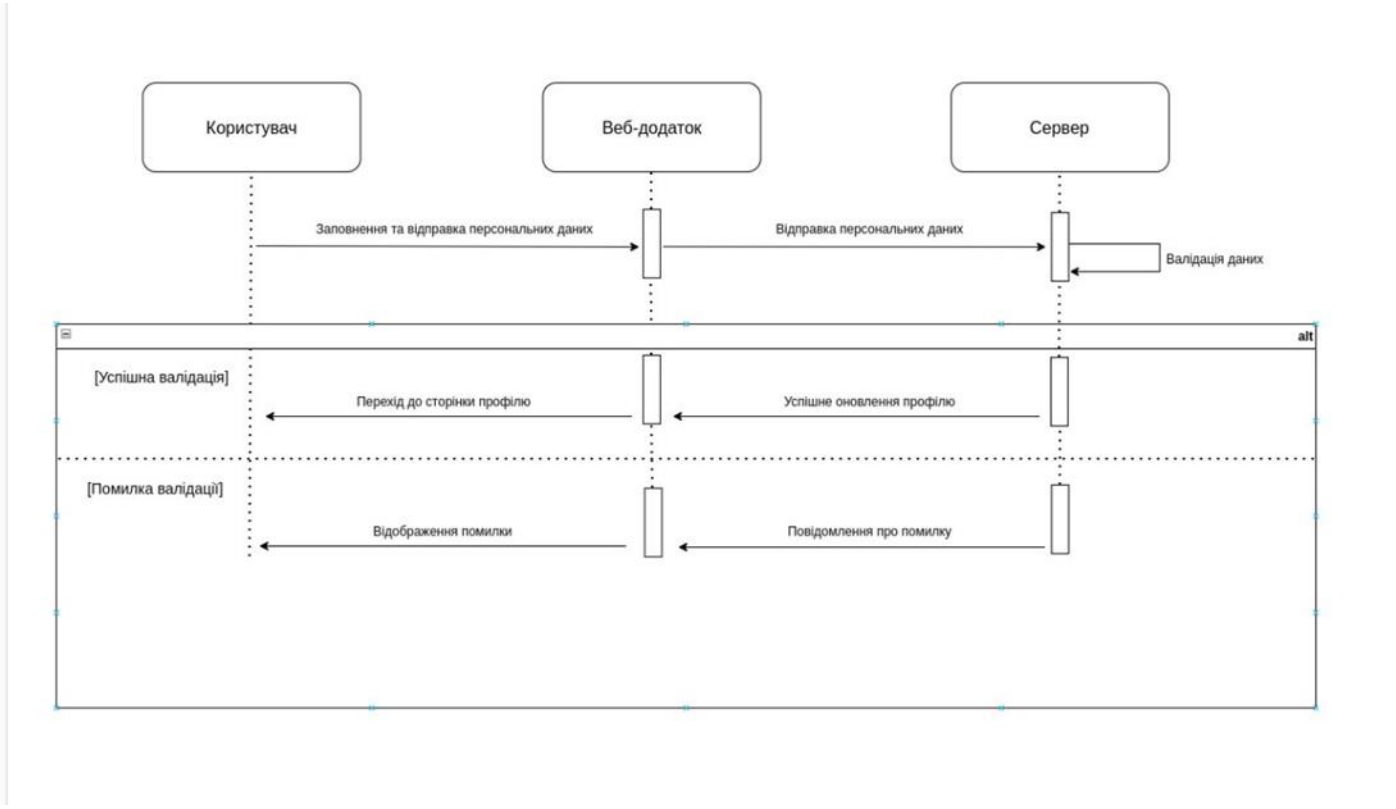


Рисунок 2.4 – Діаграма послідовності для прецеденту «Створення профілю»

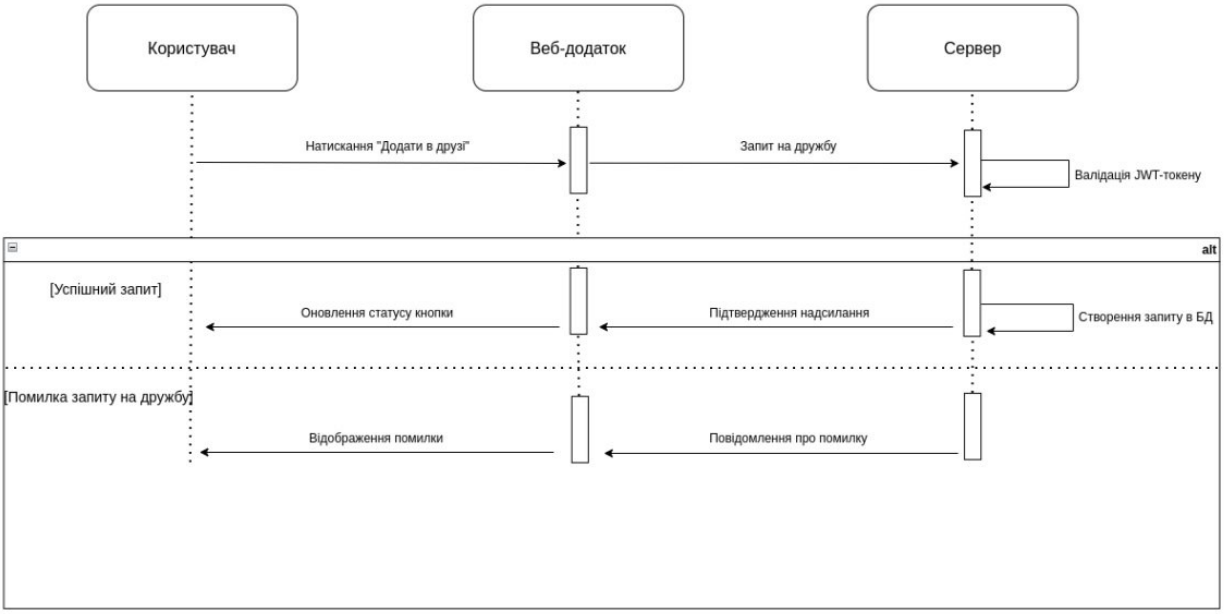


Рисунок 2.5 – Діаграма послідовності для прецеденту «Надсилання запиту на дружбу»

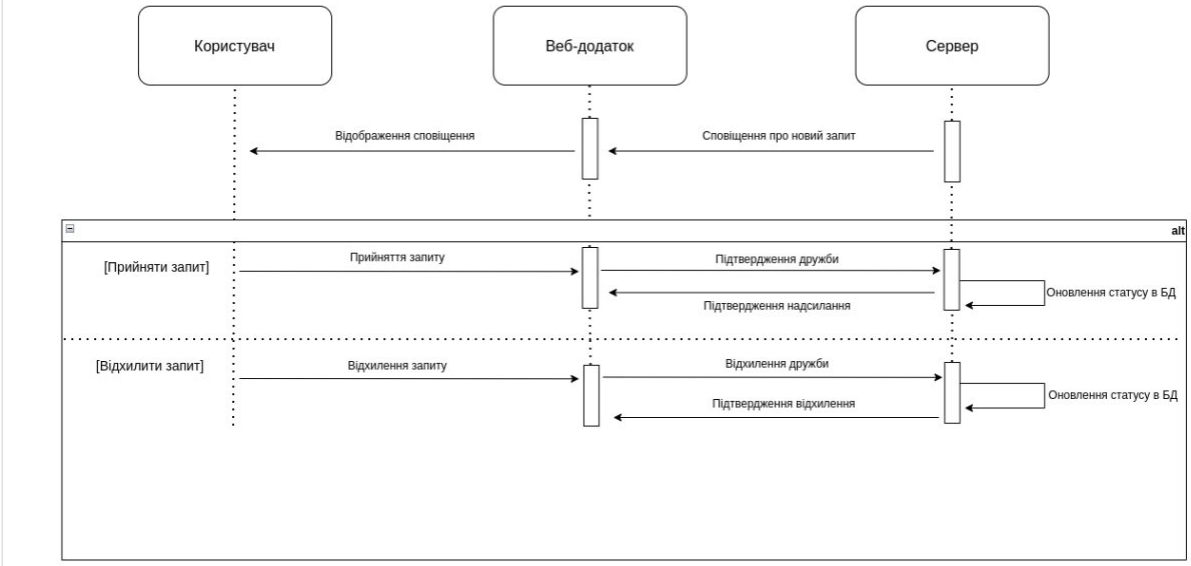


Рисунок 2.6 – Діаграма послідовності для прецеденту «Відповідь на запит дружби»

2.5 Розгорнуті описи прецедентів

Розгорнуті описи прецедентів деталізують функціональність вебсистеми, описують вимоги до кожного сценарію, очікувані результати, основні та альтернативні потоки подій.

2.5.1 Прецедент П1. Реєстрація користувача

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

Користувач: Бажає створити обліковий запис, щоб отримати доступ до функцій додатку.

Передумови: Користувач повинен мати доступ до інтернету та електронну пошту.

Результати (Постумови): Обліковий запис створено, користувач може увійти в систему.

Основний успішний сценарій:

– користувач переходить на сторінку реєстрації;

– користувач заповнює форму реєстрації;

– користувач підтверджує введені дані та натискає кнопку «Зареєструватися»;

– система перевіряє коректність даних та відправляє електронний лист для підтвердження реєстрації;

– користувач підтверджує реєстрацію через посилання в електронному листі;

– система активує обліковий запис та перенаправляє користувача на сторінку входу.

Розширення або альтернативні потоки:

Поля форми заповнені некоректно:система виводить повідомлення про помилку та просить виправити дані.

Адреса електронної пошти вже зареєстрована: система виводить відповідне

повідомлення та пропонує скористатися функцією відновлення пароля.

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Часто.

Відкриті питання: Відсутні.

2.5.2 Прецедент П2. Авторизація користувача

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

Користувач: Бажає увійти в систему для використання додатку.

Передумови: Користувач має зареєстрований обліковий запис.

Результати (Постумови): Користувач отримує доступ до персоналізованого інтерфейсу.

Основний успішний сценарій:

– користувач переходить на сторінку входу;

– користувач вводить електронну пошту та пароль;

– користувач натискає кнопку «Увійти»;

– система перевіряє відповідність даних у базі користувачів;

– система авторизує користувача та перенаправляє його на головну сторінку профілю.

Розширення або альтернативні потоки:

Введено некоректні дані: система виводить повідомлення про помилку та просить спробувати ще раз.

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Часто.

Відкриті питання: Відсутні.

2.5.3 Прецедент П3. Створення профілю

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

– Користувач: Бажає персоналізувати свій обліковий запис, щоб зробити

його привабливим для інших користувачів.

Передумови: Користувач має активний обліковий запис.

Результати (Постумови): Профіль створено та збережено в системі.

Основний успішний сценарій:

– користувач заходить у розділ «Редагування профілю»;

– користувач додає персональні дані (ім'я, вік, місто, опис інтересів, фото);

– користувач зберігає внесені дані;

– система оновлює профіль користувача та підтверджує успішне збереження.

Розширення або альтернативні потоки:

Додано некоректний формат фото: система повідомляє про помилку та просить обрати інший файл.

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Іноді.

Відкриті питання: Відсутні.

2.5.4 Прецедент П4. Пошук людей за інтересами

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

– Користувач: Бажає знайти людей із схожими інтересами для знайомства.

Передумови: Користувач авторизований у системі.

Результати (Постумови): Система виводить список користувачів, які відповідають пошуковим критеріям.

Основний успішний сценарій:

– користувач відкриває розділ пошуку людей;

– користувач задає пошукові критерії (інтереси, вік, геолокацію);

– користувач натискає кнопку «Пошук»;

– система обробляє запит та відображає список відповідних користувачів.

Розширення або альтернативні потоки:

Система не знаходить відповідних користувачів: виводиться повідомлення

«Результатів не знайдено».

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Часто.

Відкриті питання: Відсутні.

2.5.5 Прецедент П5. Надсилання запиту на дружбу

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

– Користувач: Бажає ініціювати дружні зв'язки з іншими користувачами.

Передумови: Користувач авторизований у системі.

Результати (Постумови): Запит на дружбу успішно відправлено.

Основний успішний сценарій:

– користувач переходить на профіль іншого користувача;

– користувач натискає кнопку «Додати в друзі»;

– система відправляє запит на дружбу іншому користувачу;

– інший користувач отримує сповіщення про запит.

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Часто.

Відкриті питання: Відсутні.

2.5.6 Прецедент П6. Відповідь на запит дружби

Основний виконавець: Користувач.

Зацікавлені особи та їх потреби:

– Користувач: Бажає прийняти або відхилити запит на дружбу.

Передумови: Користувач авторизований у системі.

Результати (Постумови): Запит на дружбу прийнято або відхилено.

Основний успішний сценарій:

– користувач відкриває сповіщення про запит на дружбу;

– користувач переглядає профіль відправника запиту;

– користувач обирає одну з опцій: «Прийняти» або «Відхилити»;

– система виконує дію відповідно до вибору користувача;

– відправник запиту отримує сповіщення про відповідь.

Розширення або альтернативні потоки:

Користувач не реагує на запит: система залишає запит у статусі «Очікування» до подальшої дії.

Спеціальні вимоги: Відсутні.

Список технологій та типів даних: Відсутні.

Частота використання: Іноді.

Відкриті питання: Відсутні.

3 РОЗРОБКА СИСТЕМИ

3.1 Розробка клієнтської частини вебсистеми

3.1.1 Налаштування середовища розробки

Розробка клієнтської частини вебсистеми "mateup" розпочинається з налаштування проєктного середовища через файл package.json, який визначає залежності, сценарії запуску та конфігурацію проєкту. Код даного файлу наведено у лістингу 3.1.

Лістинг 3.1 – Код файлу package.json

```
{"name": "mateup",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev --turbopack",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "@heroui/react": "^2.8.0-beta.2",
    "@hookform/resolvers": "^5.0.1",
    "@tailwindcss/postcss": "^4.1.4",
    "country-state-city": "^3.2.1",
```

```
    "framer-motion": "^12.7.4",
    "jsonwebtoken": "^9.0.2",
    "ky": "^1.8.1",
    "next": "15.3.1",
    "react": "^19.0.0",
    "react-dom": "^19.0.0",
    "react-hook-form": "^7.56.1",
    "react-icons": "^5.5.0",
    "zod": "^3.24.3"
  },
  "devDependencies": {
    "@eslint/eslintrc": "^3",
    "@types/node": "^20",
    "@types/react": "^19",
    "@types/react-dom": "^19",
    "autoprefixer": "^10.4.21",
    "eslint": "^9",
    "eslint-config-next": "15.3.1",
    "postcss": "^8.5.3",
    "tailwindcss": "^4.1.4",
    "typescript": "^5"}}
```

У ньому вказана назва "mateup", версія 0.1.0, приватний статус та скрипти для розробки ("next dev --turbopack"), збірки ("next build"), запуску ("next start") і перевірки коду ("next lint"). Залежності поділені на основні та розробницькі, забезпечуючи функціональність і інструментарій. Next.js 15.3.1 є основою для серверно-рендерених React-додатків, React 19.0.0 і react-dom 19.0.0 відповідають за динамічний інтерфейс, react-hook-form 7.56.1 і zod 3.24.3 спрощують роботу з формами та валідацію. Tailwindcss 4.1.4 з autoprefixer 10.4.21 забезпечує адаптивний дизайн, framer-motion 12.7.4 додає анімації, а react-icons 5.5.0 іконки, country-state-city 3.2.1 і react-country-state-city 1.1.12 – геодані, ky 1.8.1 – HTTP-запити, jsonwebtoken 9.0.2 – аутентифікація. TypeScript 5, eslint 9 і eslint-config-next 15.3.1 підтримують якість коду. Ці бібліотеки забезпечують масштабованість, безпеку та зручність інтерфейсу.

3.1.2 Створення структури проєкту

Розробка сучасного веб-додатку вимагає не просто написання функціонального коду, а й глибокого розуміння принципів архітектурного проектування. Створення логічної, масштабованої та легко підтримуваної структури проєкту є фундаментальним кроком, що визначає успішність та

довговічність програмного продукту. Це не просто питання естетики або сліпого дотримання гайдлайнів, а стратегічне рішення, що впливає на швидкість розробки, ефективність співпраці в команді, мінімізацію технічного боргу та легкість впровадження нових функціональностей. При проектуванні веб-додатку "MATEUP" головним пріоритетом було формування інтуїтивно зрозумілої та гнучкої архітектури, здатної адаптуватися до майбутніх змін та розширень, спираючись на передові практики фронтенд-розробки. Структура проєкту "MATEUP" наведена на рисунку 3.1.

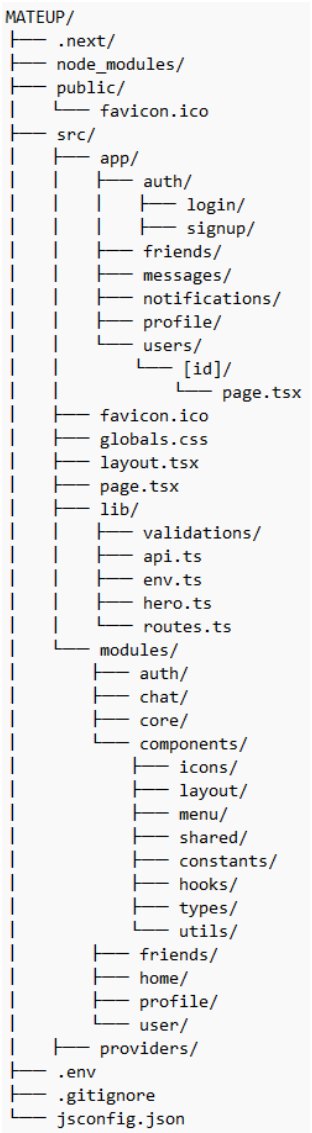


Рисунок 3.1 – Структура проєкту "MATEUP"

Основою структури є коренева папка src, що містить увесь вихідний код. У ній виділяється ключова директорія app, яка реалізує файлову маршрутизацію. Вона організовує додаток за функціональними областями, де кожна підпапка відповідає за окремий маршрут:

– auth/: функціонал автентифікації (вхід, реєстрація);

– friends/: управління списком друзів;

– profile/: інформація та редагування профілю користувача;

– users/[id]/: динамічні маршрути для профілів користувачів.

На кореневому рівні src також розміщені глобальні файли: favicon.ico, globals.css (стилі), layout.tsx (макет сторінок) та page.tsx (коренева сторінка).

Директорія lib/ містить загальні утиліти та допоміжні функції для повторного використання коду та єдиного підходу:

– validations/: модулі для валідації даних;

– api.ts: централізована взаємодія з API;

– env.ts: управління змінними середовища;

– hero.ts: допоміжні функції/константи;

– routes.ts: централізоване визначення маршрутів.

Папка modules/ забезпечує подальшу декомпозицію функціоналу. Вона включає:

– auth/, chat/, core/: глобальні функціональні комплекси;

– components/: центральний репозиторій для повторно використовуваних UI-компонентів, організованих за категоріями (icons, layout, menu, shared, constants, hooks, types, utils);

– додаткові функціональні модулі (friends/, home/, profile/, user/), що містять компоненти та логіку, специфічну для цих областей, але не є окремими сторінками.

Директорія providers/ слугує для розміщення компонентів-провайдерів, що забезпечують глобальний доступ до даних або сервісів.

На верхньому рівні проєкту розташовані стандартні службові та конфігураційні файли: .next/, node_modules/, public/, .env, .gitignore, jsconfig.json.

Застосована модульна та ієрархічна структура проєкту "MATEUP" демонструє принципи чистої архітектури. Вона забезпечує чітке розділення відповідальності, мінімізує залежності, полегшує тестування, прискорює розробку та покращує підтримуваність, створюючи міцну основу для масштабування додатка.

3.1.3 Побудова глобального макета додатка (Layout)

Створення послідовного інтерфейсу є ключовим для будь-якого веб-додатку. Концепція глобального макета (Layout) дозволяє відокремити універсальні елементи дизайну від унікального вмісту сторінок, спрощуючи розробку, підтримку та забезпечуючи єдиний користувацький досвід. Це дозволяє уникнути дублювання таких компонентів, як навігаційні панелі, визначаючи їх лише один раз.

У Next.js, файл layout.tsx у кореневій директорії src слугує основним шаблоном для всього додатка. Він обгортає всі сторінки спільними компонентами, надаючи глобальні стилі, шрифти та метадані (лістинг 3.2).

Лістинг 3.2 – Код файлу layout.tsx

```
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
import "./globals.css";
import { Providers } from "@/providers/providers";
import NavBar from "@/modules/core/components/layout/NavBar";

const geistSans = Geist({
  variable: "--font-geist-sans",
  subsets: ["latin"],
});

const geistMono = Geist_Mono({
  variable: "--font-geist-mono",
  subsets: ["latin"],
});

export const metadata: Metadata = {
  title: "MateUp",
};

export default function RootLayout({
  children,
}: {
```

```
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body>
        <NavBar />
        <Providers>{children}</Providers>
      </body>
    </html>
  );
}
```

Цей файл імпортує оптимізовані шрифти Geist та Geist_Mono з next/font/google, глобальні стилі з globals.css, а також компоненти Providers та NavBar. Providers обгортає вміст сторінки (children), надаючи доступ до глобальних контекстів чи станів. Метадані для всього додатка, зокрема title: "MateUp", визначаються тут. Основний компонент RootLayout рендерить базову HTML-структуру (<html> з lang="en", <body>), всередині якого розміщені NavBar (фрагмент коду компонента наведений в лістингу 3.3, повний код у додатку А) та Providers з children.

Лістинг 3.3 – Фарагмент коду компонента NavBar

```
export default function NavBar() {
  const pathname = usePathname();

  return (
    <Navbar  maxWidth="full" className="bg-dark-slate p-2">
      <div className="flex w-full items-center">
        <NavbarBrand className="flex-shrink-0">
          <HeroLink href="/">
            <LogoIcon className="w-[270px] h-[70px]" />
          </HeroLink>
        </NavbarBrand>
        <div className="flex-1 flex justify-center">
          <NavbarContent className="hidden lg:flex justify-center
flex-1 gap-15">
            <NavbarItem className="group">
              <HeroLink href="/messages">
                <BsWechat
                  className={`w-[35px] h-[35px] ${
                    pathname === "/messages" ? "text-coral-red" :
"text-white"
                  }`}
                />
```

```
              </HeroLink>
            </NavbarItem>

            <NavbarItem className="group">
              <HeroLink href="/friends">
                <FaUsers
                  className={`w-[35px] h-[35px] ${
                    pathname === "/friends" ? "text-coral-red" :
"text-white"
                  }`}
                />
              </HeroLink>
            </NavbarItem>

            <NavbarItem className="group">
              <HeroLink href="/notifications">
                <IoMdNotifications
                  className={`w-[35px] h-[35px] ${
                    pathname === "/notifications"
                      ? "text-coral-red"
                      : "text-white"
                  }`}
                />
              </HeroLink>
            </NavbarItem>
          </NavbarContent>
        </div>
(Далі йде частина коду, яка відповідає за відображення елементів
правої частини навігаційної панелі)
```

Компонент NavBar.tsx реалізує навігаційну панель, що є невід'ємною частиною глобального макета. Позначений директивою "use client", він є інтерактивним. NavBar побудований з використанням UI-компонентів бібліотеки @heroui/react. Для навігації між сторінками використовується Link з next/link, забезпечуючи швидкі переходи.

Активне посилання динамічно виділяється за допомогою хука usePathname(), який змінює стиль іконок навігаційних елементів (Повідомлення, Друзі, Сповіщення). Структура NavBar включає логотип (LogoIcon), що веде на головну сторінку, центральні навігаційні іконки (з react-icons), видимі на великих екранах, та елементи управління користувачем у правій частині. Остання секція містить кнопки "Login", "Sign Up" (для не авторизованих користувачів) або компонент UserMenu (для аватара та випадаючого меню).

3.1.4 Реалізація сторінок автентифікації

Функціонал автентифікації користувачів є фундаментальною складовою більшості сучасних веб-додатків, забезпечуючи безпечний та персоналізований доступ до системи. У проєкті "MATEUP" цей критично важливий блок реалізований через дві окремі, але взаємопов'язані сторінки: "Вхід" та "Реєстрація", кожна з яких організована з використанням композиції компонентів для підвищення модульності та читабельності коду.

Кожна сторінка автентифікації, login/page.tsx та signup/page.tsx (код сторінок наведений в лістингах 3.4 та 3.5) виконує роль обгортки, що надає відповідній формі мінімальне візуальне оточення. Наприклад, LoginPage.tsx імпортує компонент LoginForm (фрагмент коду компонента наведений в лістингу 3.6, повний код у додатку А) і розміщує його в контейнері з фоновим градієнтом (bg-gradient-to-r from-coral-red to-black-pink), що забезпечує візуально привабливий та єдиний стиль для всіх сторінок автентифікації. Аналогічно, SignUpPage.tsx (фрагмент коду компонента наведений в лістингу 3.7, повний код у додатку А) відповідає за відображення SignUpForm у схожому візуальному контексті. Такий підхід підкреслює розділення відповідальності: сторінки керують загальним макетом та фоном, тоді як самі форми інкапсулюють всю бізнес-логіку та UI-елементи, пов'язані з введенням та обробкою даних користувача.

Лістинг 3.4 – Код сторінки login/page.tsx

```
import React from "react";
import LoginForm from "../../../modules/auth/components/LoginForm";

export default function LoginPage() {
  return (
    <div className="h-screen overflow-auto bg-gradient-to-r from-
coral-red to-black-pink">
      <LoginForm />
    </div>
  );
}
```

Лістинг 3.5 – Код сторінки signup/page.tsx

```
import React from 'react'
import SignUpForm from
"../../../modules/auth/components/SignUpForm";

export default function SignUpPage() {
  return (
    <div className="overflow-hidden no-scrollbar h-screen bg-
gradient-to-r from-coral-red to-black-pink">
      <SignUpForm />
    </div>
  )
}
```

Лістинг 3.6 – Фрагмент коду компонента LoginForm.tsx

```
export default function LoginForm() {
  const {register, handleSubmit, formState: { isValid, errors },
 } = useForm<loginSchema>({resolver: zodResolver(loginSchema),
 mode: "onTouched", });
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const router = useRouter();
  const [isVisible, setIsVisible] = React.useState(false);
  const toggleVisibility = () => setIsVisible(!isVisible);
(Функція для API запиту на логін)
  return (
    <div >
      <div >
        <h1 > Login Form</h1>
        {error && (<div className="text-red-500 text-center mb-
4">{error}</div> )}
        <Form onSubmit={handleSubmit(onSubmit)}>
          <Input label="Email Address" labelPlacement="outside"
type="email" isRequired placeholder="Enter your email"
className="mb-4" {...register("email")} isInvalid={!!errors.email}
errorMessage={errors.email?.message}/>
          <Input label="Password" labelPlacement="outside"
type={isVisible ? "text" : "password"} isRequired
placeholder="Password" className="mb-4" {...register("password")}
isInvalid={!!errors.password}
errorMessage={errors.password?.message} endContent={
              <button type="button" onClick={toggleVisibility}
className="focus:outline-none" aria-label={isVisible ? "Hide
password" : "Show password"}>
                {isVisible ? (Іконка перекресленого ока) />) :
(Іконка ока/>)}
              </button>}/>
(Далі йде частина коду, яка відповідає за кнопку "Забули пароль?" та
кнопку "Увійти")
```

Лістинг 3.7 – Фрагмент коду компонента SignUp.tsx

```
export default function SignUpForm() {
  const {register, handleSubmit, formState: { isValid, errors },
 } = useForm<signupSchema>({resolver: zodResolver(signupSchema),
mode: "onTouched",});
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const router = useRouter();
  const [isVisible, setIsVisible] = React.useState(false);
  const toggleVisibility = () => setIsVisible(!isVisible);
(Функція для API запиту на реєсьрацію)
  return (
    <div >
      <div
        <div >
          <h1>Sign Up Form </h1>
          <p>Welcome, future mate! &#x1F609;</p>
        </div>
        {error && ( <div className="text-red-500 text-center mb-
4">{error}</div>)}
        <Form onSubmit={handleSubmit(onSubmit)}>
          <Input label="Name" labelPlacement="outside"
type="username" isRequired placeholder="Enter your name"
{...register("username")}isInvalid={!!errors.username}
errorMessage={errors.username?.message}/>
          <Input label="Email Address" labelPlacement="outside"
type="email" isRequired placeholder="Enter your email"
{...register("email")} isInvalid={!!errors.email}
errorMessage={errors.email?.message} />
          <Input label="Password" labelPlacement="outside"
type={isVisible ? "text" : "password"} isRequired
placeholder="Password" {...register("password")}
isInvalid={!!errors.password}
errorMessage={errors.password?.message}endContent={
              <button type="button" onClick={toggleVisibility}
aria-label={isVisible ? "Hide password" : "Show password"}>
                {isVisible ? (Іконка перекресленого ока) />) :
(Іконка ока/>)}
              </button>}/>
(Далі йде частина коду, яка відповідає за кнопку "Зареєструватися"
та кнопку "Маєте акаунт?")
```

Компоненти LoginForm.tsx та SignUpForm.tsx – це інтерактивні клієнтські форми, що становлять ядро автентифікації. Вони використовують react-hook-form та zod (@hookform/resolvers/zod) для надійного управління станом форми та валідації вхідних даних на клієнтській стороні, забезпечуючи миттєвий зворотний

зв'язок. UI-елементи імпортуються з бібліотеки @heroui/react для консистентного дизайну.

Після успішної валідації, дані асинхронно відправляються на сервер через централізований API-клієнт (api з @/lib/api) та відповідні маршрути (API_ROUTES). Успішна відповідь сервера призводить до збереження даних користувача та токенів у localStorage та перенаправлення на головну сторінку. Обробка помилок забезпечує зворотний зв'язок користувачу, а індикатор isLoading візуалізує процес завантаження. Форми також включають функціонал перемикання видимості пароля та посилання для переходу між формами входу/реєстрації, а також на відновлення пароля.

3.1.5 Реалізація сторінки керування профілем користувача

Сторінка профілю користувача profile/page.tsx (фрагмент коду наведений в лістингу 3.8, повний код у додатку А) являє собою комплексну систему для створення та редагування персональної інформації в додатку знайомств. Архітектура побудована на основі React Hook Form з валідацією через Zod схему, що забезпечує надійну обробку даних та миттєву валідацію полів форми. Система автоматично визначає режим роботи - при першому відвідуванні користувач бачить порожню форму для заповнення базової інформації, а при повторних візитах дані підвантажуються з сервера через API запити за допомогою бібліотеки ky. Інтерфейс розроблений з урахуванням сучасних принципів UX/UI дизайну, використовуючи Hero UI компоненти для створення елегантного та функціонального користувацького досвіду.

Фрагмент коду profile/page.tsx наведений в лістингу 3.8, повний код у додатку А

Лістинг 3.8 - Фрагмент коду profile/page.tsx

```
export default function Profile() {
  const {register,handleSubmit, formState: { errors, isValid },
control, trigger,reset,} = useForm<ProfileFormData>({ resolver:
zodResolver(profileSchema), mode: "onBlur", defaultValues: {(перелік
значень за замовченням)},});
  const router = useRouter();
  const fetchProfile = async () => {
```

```
    const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");

    try {
      const response = await api.get(API_ROUTES.GET_PROFILE, {
          headers: {"Content-Type":
"application/json",Authorization: `Bearer
${userAuthData.tokens.accessToken}`,},
        .json<{status: string; message: string | null; data: any;
}>();
      (Обробка відповіді на запит від серверу) };
  const onSubmit = async (data: ProfileFormData) => {
    const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
    try {
      const response = await api.put(API_ROUTES.UPDATE_PROFILE, {
          method: "POST",
          headers: {"Content-Type": "application/json",
Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},
          body: JSON.stringify(data), })
        .json<{status: string; message: string | null; data: any;
}>();
     (Обробка відповіді на запит від серверу)};
  useEffect(() => {fetchProfile()}, [reset])
  return (<div
      <Card
        <h6 >Your profile</h6>
        (Використання компонентів ProfileImage, ProfileForm,
ProfileBio InterestsSection)
      </Card>}
```

Для управління формою в файлі profile/page.tsx використовується react-hook-form із zod та zodResolver для валідації даних за схемою profileSchema. Компонент інтегрує бібліотеку @heroui/react для рендерингу UI-елементів (Card, CardBody, Button). Для навігації застосовується хук useRouter із next/navigation, який перенаправляє користувача на головну сторінку після успішного оновлення профілю. Інтеграція з API реалізується через утиліту api з маршрутами API_ROUTES, використовуючи токен авторизації з localStorage. Дані про країни обробляються за допомогою бібліотеки country-state-city. Компонент розбито на модулі: ProfileImage для фото, ProfileForm (Фрагмент коду наведений в лістингу 3.9, повний код у додатку А) для основних даних, ProfileBio для біографії та InterestsSection для інтересів, що інтегруються з формою через пропси react-hook-form. Помилки мережі логуються для дебагінгу.

Лістинг 3.9 – Фрагмент коду ProfileForm

```
export default function ProfileForm({register, errors,trigger,
control,}: ProfileFormProps) {
(Стилі для елементів форм (input, select, radio)
  const selectedCountry = useWatch({ control, name: "country" });
  const countryOptions = Country.getAllCountries().map((c) => ({
    value: c.isoCode,label: c.name,}));
  const rawCities = selectedCountry ?
City.getCitiesOfCountry(selectedCountry) ?? []: [];
  const cityOptions = rawCities.map((c) => ({
    id: c.latitude,value: c.name,label: c.name,}));
  return (
    <div >
      <div>
        <RHFController
         name="firstName" control={control} render={({ field }) => (
            <Input
              {...field} onBlur={() => trigger("firstName")}
              label="First name"/>)}/>
        {errors.firstName && (
<p className="text-red-500 text-sm">{errors.firstName.message}</p>)}
      </div>
      (Поле для вводу lastname)
      <div>
        <RHFController
         name="dateOfBirth"control={control}render={({ field }) => (
            <DatePicker
              label="Date of Birth" onChange={(value)
=>field.onChange(value ? new Date(value.toString()) : null) }
              onBlur={() => trigger("dateOfBirth")}/>)}/>
        {errors.dateOfBirth && (
<pclassName="text-red-500 text">{errors.dateOfBirth.message}</p> )}
      </div>
      <div>
        (Поля для вибору країни та міста)
      </div>
      (Поле для вибору гендера)
    </div>);}
```

ProfileForm.tsx є найскладнішим компонентом, який інкапсулює множинні форм контроли з різними типами input елементів. Використовує useWatch хук для реактивного відстеження змін в полі country та динамічного оновлення cityOptions через City.getCitiesOfCountry() API з бібліотеки country-state-city. Містить три різні classNames конфігурації (inputClassNames, selectClassNames, radioClassNames) для стилізації різних типів контролів згідно з дизайн системою. Реалізує Controller компоненти (RHF) для складних контролів як DatePicker, Select

та RadioGroup, з кастомними onChange обробниками для трансформації даних (наприклад, new Date(value.toString()) для DatePicker). Використовує country-state-city для отримання списків країн та міст, з map трансформацією в потрібний формат для SelectItem компонентів.

3.1.6 Реалізація головної сторінки

Файл page.tsx (фрагмент коду наведений в лістингу 3.10, повний код у додатку А) є клієнтським React-компонентом на TypeScript, який реалізує головну сторінку веб-додатку для відображення стрічки користувачів із можливістю фільтрації за інтересами, віком та пошуковим запитом. Компонент забезпечує завантаження даних користувачів через API-запит до маршруту API_ROUTES.GET_FEED, використовуючи токен авторизації з localStorage. Для управління станом використовуються хуки React: useState для збереження списку користувачів, стану завантаження та помилок, а також useEffect для виклику функції fetchFeed під час ініціалізації. Фільтрація даних реалізується через кастомний хук useCardFilter, який повертає параметри пошуку, вибрані інтереси, діапазон віку та відфільтровані картки користувачів. Для стилізації застосовується Tailwind CSS, що забезпечує адаптивний дизайн із градієнтним фоном і відступами.

Компонент складається з кількох підкомпонентів: SearchBar для введення пошукового запиту, AgeSlider для вибору вікового діапазону, InterestFilter для фільтрації за інтересами та FeedCard (фрагмент коду наведений в лістингу 3.11, повний код у додатку А) для відображення картки кожного користувача. Пагінація реалізована через компонент CardPagination, який дозволяє перемикатися між сторінками, показуючи по 12 карток на сторінку. UI-елементи, такі як Card, походять із бібліотеки @heroui/react, що забезпечує стилізоване відображення контейнера стрічки. Обробка помилок мережі здійснюється з виведенням повідомлень у консоль та стан error, а стан isLoading контролює відображення процесу завантаження. Компонент забезпечує інтерактивний і зручний інтерфейс для пошуку та перегляду профілів користувачів.

Лістинг 3.10 – Фрагмент коду головної сторіеки page.tsx

```
export default function Home() {
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const [users, setUsers] = useState<any[]>([]);
  const { searchQuery, setSearchQuery, selectedInterests,
handleInterestClick, ageRange, setAgeRange, filteredCards,
  } = useCardFilter(users);
  const [currentPage, setCurrentPage] = useState<number>(1);
  const itemsPerPage = 12;
  const startIndex = (currentPage - 1) * itemsPerPage;
  const currentCards = filteredCards.slice(
    startIndex,
    startIndex + itemsPerPage
  );
  const handleAgeChange = (min: number, max: number) => {
    setAgeRange({ minAge: min, maxAge: max });
  };

  const fetchFeed = async () => {
    const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
setError(null); setIsLoading(true);
    try {
      const response = await api.post(API_ROUTES.GET_FEED, {
          headers: {"Content-Type": "application/json",
Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},})
        .json<{
status: string; message: string | null; data: { user: any }[]; }>();
   };
  useEffect(() => {fetchFeed();}, []);
  return (
    <div >
      <div >
        <h1>Find Your People: Connect with Others Who Share Your
Interests</h1>
 <p>Because life is more interesting when you're surrounded by like-
minded individuals. Start exploring now and find your tribe!</p>
      </div>
      <Card >
  (Використання компонентів ProfileImage, ProfileForm, ProfileBio
InterestsSection)
      </Card>
    </div>
  );
}
```

Лістинг 3.11 – Фрагмент коду FeedCard

```
export default function FeedCard({ user }: { user: IFeedCard }) {
  const json = encodeURIComponent(JSON.stringify(user));
  const displayedInterests = user.profile.interests.slice(0, 3);
```

```
  const hasMoreInterests = user.profile.interests.length > 2;
  const age = calculateAge(user.profile.dateOfBirth);

  return (
(<Link
      href={{
        pathname: `/users/${user.id}`,
        query: { data: json },}}>
      <Card
        <Image
          src={user.profile.profilePhoto}
          alt={`${user.profile.firstName}'s photo`}/>
        <div
          <p > {user.profile.firstName} <span >{age}</span></p>
          <p >user.profile.country}, {user.profile.city}</p>
          <div >
            (Виведення інтересів користувачв)
          </div>
        </div>
      </Card>
    </Link>);}
```

Компонент FeedCard компактно відображає профіль користувача, деструктуризуючи пропс user типу IFeedCard. Він обмежує показ інтересів до трьох за допомогою user.interests.slice(0, 3) та використовує hasMoreInterests для індикації наявності додаткових інтересів. Вік користувача динамічно обчислюється функцією calculateAge з user.birthDate.

Картка є Hero UI Card компонентом зі стилями bg-gray-100 border-2 border-black-pink rounded-lg w-64 h-96 relative overflow-hidden, інтегрована з Next.js маршрутизацією через as={Link} та href={/users/${user.id}}, і є інтерактивною завдяки isPressable. Зображення user.imageSrc повністю покриває картку. Градієнтний оверлей h-1/3 bg-gradient-to-t from-black-pink to-transparent z-10 створює напівпрозорий ефект. Текстовий контент, розташований абсолютно, включає ім'я та вік користувача, географічну інформацію (user.country, user.city), а також перелік інтересів, де span теги генеруються через displayedInterests.map, а ... відображається за наявності додаткових інтересів.

3.1.7 Реалізація сторінка профілю інших користувачів

Файл user/page.tsx (лістинг 3.12) є клієнтським React-компонентом на TypeScript, який відповідає за відображення сторінки профілю конкретного

користувача у веб-додатку. Компонент отримує параметри маршруту (id) та query-параметри (data) через пропси PageProps, використовуючи хук useParams із бібліотеки next/navigation для доступу до динамічних параметрів URL. Основна логіка полягає в обробці JSON-даних користувача, переданих через query-параметр data, з їх декодуванням та парсингом у тип IFeedCard. У разі відсутності або некоректного формату даних компонент відображає повідомлення про помилку. Для стилізації застосовуються класи Tailwind CSS, зокрема для відображення повідомлень про помилки в червоному кольорі з відступами.

Компонент використовує єдиний підкомпонент UserProfile (фрагмент коду наведений в лістингу 3.13, повний код у додатку А), якому передається об’єкт користувача (user) типу IFeedCard для рендерингу профілю. Якщо дані успішно декодовані та розпарсені, UserProfile відображає детальну інформацію про користувача. У разі помилок, таких як відсутність даних або невалідний JSON, користувачу показується відповідне повідомлення про помилку. Компонент не взаємодіє з API безпосередньо, покладаючись на дані, передані через URL, що робить його легким і залежним від коректності вхідних параметрів. Використання use client вказує на те, що компонент рендериться на клієнтській стороні, забезпечуючи інтерактивність.

Лістинг 3.12 – Код сторінки page.tsx

```
"use client";
import { IFeedCard } from "@/modules/core/types";
import UserProfile from "@/modules/user/components/UserProfile";
import { useParams } from "next/navigation";
import React, { use } from "react";

interface PageProps {
  params: { id: string };
  searchParams: { data?: string };
}

export default function UserProfilePage({ params, searchParams }:
PageProps) {
  const { data } = searchParams;

  if (!data) {
    return (
      <p className="p-4 text-red-600">
```

```
        Error: No user data provided via query string.
      </p>
    );
  }

  let user: IFeedCard;
  try {
    const decoded = decodeURIComponent(data);
    user = JSON.parse(decoded) as IFeedCard;
  } catch {
    return (
      <p className="p-4 text-red-600">
        Error: Invalid user data format.
      </p>
    );
  }

  return <UserProfile user={user} />;
}
```

Лістинг 3.13 – Фрагмент коду UserProfile.tsx

```
export default function UserProfile({ user }: { user: IFeedCard }) {
  const searchParams = useSearchParams();
  const initialTab = searchParams.get("tab") || "profile";
  const [tab, setTab] = useState(initialTab);
  useEffect(() => {setTab(initialTab);}, [initialTab]);
  return (
    <div >
      <div >
        <div >
          <div >
            <Image
              src={user?.profile?.profilePhoto}
              alt={`${user?.profile?.firstName}'s photo`}/>
          </div>
           (Виведення основної інфломації про користувача)
          <div >
(Кнопки для переключення між компонентами інформації профілю та
чатом, кнопка для додавання в друзі)
        </div>

        {tab === "chat" ? (
          <div >
            <ChatComponent profileName={user?.profile?.firstName} />
          </div>
        ) : (
          <div >
(Виведення детальної інформації про користувача (біо та інтереси)
            </div>)}
      </div>
    </div>);}
```

UserProfile.tsx - функціональний React компонент, який реалізує детальний перегляд профілю користувача з табовою навігацією між секціями профілю та чату. Компонент використовує useSearchParams хук з Next.js для отримання параметру tab з URL та useState для управління активним табом, що дозволяє користувачам переключатися між переглядом інформації профілю та чат-інтерфейсом. Структура компонента побудована на CSS Grid системі з адаптивним дизайном через md:grid-cols-2, де лівий блок містить зображення користувача з Next.js Image компонентом та навігаційні кнопки, а правий блок умовно рендерить або ChatComponent або секції біографії та інтересів залежно від активного табу. Дані профілю зберігаються в локальному об'єкті profiles як заглушка, але в реальному додатку ці дані отримуються з серверу через ky HTTP клієнт з endpoint /api/users/{id}, забезпечуючи актуальну інформацію про користувача включаючи ім'я, вік (розрахований через calculateAge утиліту), місцезнаходження, біографію та список інтересів.

3.1.8 Реалізація сторінки друзів

Файл friends/page.tsx (фрагмент коду наведений в лістингу 3.14, повний код у додатку А) є клієнтським React-компонентом на TypeScript, який реалізує сторінку управління друзями та запитами на дружбу у веб-додатку. Компонент дозволяє користувачу переглядати список друзів та отриманих запитів на дружбу, перемикаючись між ними за допомогою вкладок, а також фільтрувати їх за пошуковим запитом. Для управління станом використовуються хуки useState для збереження активної вкладки, пошукового запиту, списків друзів та запитів, а також useEffect для виклику функцій fetchFriends і fetchReceivedRequests під час ініціалізації. Ці функції здійснюють API-запити до маршрутів API_ROUTES.GET_FRIENDS і API_ROUTES.GET_RECEIVED_FRIENDSHIP_REQUESTS, використовуючи токен авторизації з localStorage. Для стилізації застосовується Tailwind CSS, що забезпечує адаптивний дизайн із градієнтним фоном і стилізацію вкладок із динамічними ефектами.

Компонент використовує три підкомпоненти: FriendsCard (фрагмент коду наведений в лістингу 3.15, повний код у додатку А) для відображення картки друга, RequestCard (фрагмент коду наведений в лістингу 3.16, повний код у додатку А)для відображення запиту на дружбу та SearchBar для введення пошукового запиту. UI-контейнер реалізовано через компонент Card із бібліотеки @heroui/react, який створює стилізовану картку для вмісту сторінки. Логіка перемикання між вкладками ("Your Friends" та "Friend Requests") реалізується через стан activeTab, а відображення списків залежить від вибраної вкладки. Помилки мережі логуються в консоль, а у разі невдачі API-запиту виводяться відповідні повідомлення. Компонент забезпечує інтерактивний інтерфейс для перегляду та управління соціальними зв’язками користувача.

Лістинг 3.14 – Фрагмент коду friends/page.tsx

```
export default function FriendsPage() {
  const [activeTab, setActiveTab] = useState("friends");
  const [searchQuery, setSearchQuery] = useState("");
  const [friends, setFriends] = useState<any[]>([]);
const [receivedRequests, setReceivedRequests] = useState<any[]>([]);
  const fetchFriends = async () => { const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
    try {const response = await api.get(API_ROUTES.GET_FRIENDS, {
          headers: { "Content-Type": "application/json",
     Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},})
    .json<{status: string; message: string | null; data: any[];}>();
     (Обробка відповіді на запит від серверу)};
  const fetchReceivedRequests = async () => {
const userAuthData =JSON.parse(localStorage.getItem("auth_data") ||
"{}");
    try {const response = await
api.get(API_ROUTES.GET_RECEIVED_FRIENDSHIP_REQUESTS, {
          headers: {"Content-Type": "application/json",
    Authorization: `Bearer ${userAuthData.tokens.accessToken}`, },})
    .json<{status: string; message: string | null;data: any[];}>();
      (Обробка відповіді на запит від серверу)  };
  const currentList = activeTab === "friends" ? friends :
receivedRequests;
  useEffect(() => {fetchFriends()fetchReceivedRequests()}, []);
  return (
    <div>
      <div >
        <Card >
          <div >
       (Кнопки переключення між вкладками друзів і запитів в друзі)
```

```
          </div>
          (Компонент пошуку)
          <div >
            {activeTab === "friends" ? friends.map((friend:
IFriendCard) => (<FriendsCard key={friend.id} friend={friend} />)) :
receivedRequests.map((request: IFriendCard) => (<RequestCard
key={request.id} request={request} />))}
          </div>
        </Card>
```

Лістинг 3.15 – Фрагмент коду FriendsCard.tsx

```
export default function FriendsCard({ friend }: { friend:
IFriendCard }) {
  const age = calculateAge(friend?.profile?.dateOfBirth ?? 0);
  const router = useRouter();
  const [isPopoverOpen, setIsPopoverOpen] = useState(false);
  const handleChatClick = () =>
{router.push(`/users/${friend.id}?tab=chat`);};
  return (
    <div>
      <Card>
        <CardBody as={Link} href={`/users/${friend.id}`}>
          <Image
            src={friend?.profile?.profilePhoto}
            alt={`${friend?.profile?.firstName}'s photo`}/>
        </CardBody>
        (Випадаючий список для видалення або додавання в блок)
        <div >
          <p >
            {friend?.profile?.firstName}{" "}
            <span >{age}</span>
          </p>
          {friend?.profile?.country ? (
            <p>{friend?.profile?.country},
{friend?.profile?.city}</p>
          ) : <p>Unknown</p>}
          <Button
            className="absolute rounded-full size-14 min-w-0 bottom-
1 right-1 bg-coral-red"
            onPress={handleChatClick}
            aria-label="Chat"
          >
            <BsChatDotsFill className="text-white text-2xl" />
          </Button>
        </div>
      </Card>
    </div>);}
```

FriendsCard.tsx спеціалізований компонент для відображення карток існуючих друзів з повним набором інтерактивних функцій управління дружніми зв'язками. Компонент використовує HeroUI Dropdown систему для контекстного меню з опціями видалення та блокування друга, де кожна дія викликає відповідний ky запит до серверних endpoints /api/friends/{id}/delete або /api/friends/{id}/block з автоматичним оновленням локального стану після успішної операції. Карта містить gradient overlay ефект для покращення читабельності тексту поверх зображення, кнопку швидкого переходу до чату через router.push з параметром tab=chat, та інтеграцію з calculateAge утилітою для динамічного розрахунку віку користувача, при цьому вся інформація про друга отримується з серверу через initial data fetching при завантаженні компонента.

Лістинг 3.16 – Код компоненту RequestCard.tsx

```
import { IFriendCard } from "../../../modules/core/types";
import { Card, Image, CardBody, Link, Button } from "@heroui/react";
import { calculateAge } from "@/modules/core/utils/calculate-age";
import { useRouter } from "next/navigation";
import React, { useState } from "react";
import { FaUserPlus, FaUser } from "react-icons/fa";
import { MdDeleteForever } from "react-icons/md";

export default function RequestCard({ request }: { request:
IFriendCard }) {
  const age = calculateAge(request.profile.dateOfBirth ?? 0);
  const router = useRouter();
  const [isPopoverOpen, setIsPopoverOpen] = useState(false);
  return (
    <div>
      <Card>
        <Button
          className="absolute top-3 right-4 rounded-full size-14
min-w-0 z-30 "aria-label="AddFriend">
          <MdDeleteForever className="text-dark-slate text-2xl" />
        </Button>
        <Button
          className="absolute top-3 left-4 rounded-full size-14 min-
w-0 bg-black-pink z-30" aria-label="ViewProfile">
          <FaUserPlus className="text-dark-slate text-2xl" />
        </Button>
  <CardBody className="p-0" as={Link} href={`/users/${request.id}`}>
          <Image
            src={request?.profile?.profilePhoto}
            alt={`${request?.profile?.firstName}'s photo`}
```

```
            className="w-64 h-96 object-cover rounded-lg"/>
        </CardBody>
        <div className="absolute bottom-0 left-0 w-full h-1/3 bg-
gradient-to-t from-black-pink to-transparent z-10"></div>
        <div className="absolute bottom-4 left-4 right-4 text-white
z-20">
          <p className="text-xl font-bold">
            {request?.profile?.firstName} <span className="font-
normal">{age}</span></p>
          <p className="uppercase text-sm mt-1">
            {request?.profile?.country}, {request?.profile?.city}
          </p>
        </div>
      </Card>
    </div>
  );
}
```

Компонент RequestCard.tsx використовується для відображення вхідних запитів на дружбу з спрощеним інтерфейсом, який містить кнопки прийняття та відхилення запиту замість dropdown меню. При натисканні на кнопку прийняття запиту компонент виконує POST запит до /api/friend-requests/{id}/accept через ky клієнт, що переміщує користувача зі списку запитів до списку друзів, а кнопка відхилення викликає DELETE запит до /api/friend-requests/{id}/decline для видалення запиту без додавання до друзів. Компонент має таку ж візуальну структуру як FriendsCard з gradient overlay та базовою інформацією користувача, але з оптимізованим UX для швидкого прийняття рішень щодо нових знайомств, включаючи автоматичне видалення картки з DOM після успішної обробки запиту та відображення loading стану під час виконання серверних операцій.

3.2 Розробка серверної частини вебсистеми

3.2.1 Проектування структури бази даних

ER-схема наведена на рисунку 3.2 та містить такі основні таблиці: таблицю основних даних користувача (UserCore), налаштування фільтрації (UserFilters),

інформацію профілю (UserProfile), заявки на встановлення дружніх зв'язків (FriendshipRequest) та записи про дружбу (Friends).

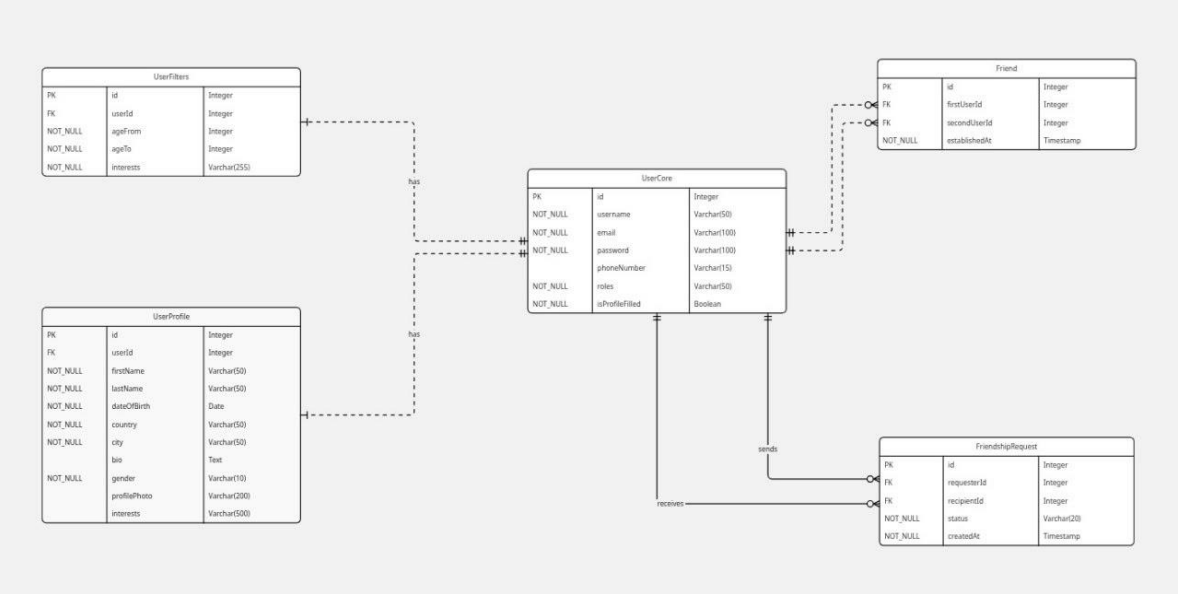


Рисунок 3.2 – ER-модель бази даних

Детальна структура представлена в таблицях 3.1-3.5.

Таблиця 3.1 - Склад таблиці UserCore

Атрибут	**Тип даних**	**Допустимі значення**	**Значення за замовчуванням**	**Обов'язковість**	**Примітки**
id	UUID	Унікальний UUID	-	Так	Первинний ключ, автоінкремент
username	varchar(255)	Латиниця	-	Так	Унікальне значення
email	varchar	Латиниця	-	Так	Унікальне значення
password	varchar	Латиниця, кирилиця	-	Так	-
phoneNumber	varchar(15)	Латиниця	-	Ні	-
roles	array(enum: Role	Значення з Role	[Role.USER]	Так	-
isProfileFilled	boolean	true/false	false	Так	-
filters	OneToOne	UserFilters	-	Ні	Каскадне видалення
profile	OneToOne	UserProfile	-	Ні	Каскадне видалення
sentFriendshipRequests	OneToMany	FriendshipRequest[]	-	Ні	Каскадне видалення
receivedFriendshipRequests	OneToMany	FriendshipRequest[]	-	Ні	Каскадне видалення
friends	ManyToMany	Friend[]	-	Ні	Через таблицю зв'язку

Таблиця users_core містить фундаментальні відомості про користувачів системи. Включає унікальні поля для імені користувача та електронної пошти, зашифрований пароль, додаткову інформацію як номер телефону та ролі, а також зв'язки з профілем та фільтрами. Первинний ключ формується як UUID. Код сутності UserCore наведений в лістингу 3.17.

Лістинг 3.17 – Код сутності UserCore

```
@Entity("users_core")
@Unique(["username", "email"])
export class UserCore extends BaseEntity {
  @Index("users_id_idx")
  @PrimaryGeneratedColumn("uuid")
  id: string;

  @Column({ type: "varchar", nullable: false, length: 255 })
  username: string;

  @Column({ type: "varchar", nullable: false })
  email: string;

  @Column({ type: "varchar", nullable: false })
  password: string;

  @Column({ type: "varchar", nullable: true, length: 15 })
  phoneNumber: string;

  @Column({ type: "enum", enum: Role, array: true, nullable: false,
default: [Role.USER] })
  roles: Role[];

  @Column({ type: "boolean", nullable: false, default: false })
  isProfileFilled: boolean;

  @OneToOne(() => UserFilters, (filters: UserFilters) =>
filters.user, { cascade: true, onDelete: "CASCADE" })
  filters: UserFilters;

  @OneToOne(() => UserProfile, (profile) => profile.user, { cascade:
true, onDelete: "CASCADE" })
  profile: UserProfile;

  @OneToMany(() => FriendshipRequest, (friendship) =>
friendship.requester, { onDelete: "CASCADE" })
  sentFriendshipRequests: FriendshipRequest[];

  @OneToMany(() => FriendshipRequest, (friendship) =>
friendship.recipient, { onDelete: "CASCADE" })
  receivedFriendshipRequests: FriendshipRequest[];
```

```
  @ManyToMany(() => Friend, (friend: Friend): UserCore[] =>
friend.users)
  @JoinTable()
  friends: Friend[];
}
```

Таблиця 3.2 - Склад таблиці UserFilters

Атрибут	Тип даних	Допустимі значення	Значення за замовчуванням	Обов'язковість	Примітки
id	UUID	Унікальний UUID	-	Так	Первинний ключ, автоінкремент
user	OneToOne	→ UserCore	-	Так	Зовнішній ключ, каскадне видалення
ageFrom	int	Натуральне число	-	Так	-
ageTo	int	Натуральне число	-	Так	-
interests	array(enum: Interest)	Значення з Interest	-	Ні	-

Сутність UserFilters (лістинг 3.18) зберігає конфігурацію фільтрів користувача для персоналізації стрічки в програмі "MateUp". Вона має зв'язок один-до-одного з UserCore.

Лістинг 3.18 – Код сутності UserFilters

```
@Entity("user_filters")
export class UserFilters extends BaseEntity {
  @PrimaryGeneratedColumn("uuid")
  id: string;

  @OneToOne(() => UserCore, (user: UserCore): string => user.id, {
    onDelete: "CASCADE",
  })
  @JoinColumn({ name: "userId" })
  user: UserCore;

  @Column({ type: "int", nullable: false })
  ageFrom: number;

  @Column({ type: "int", nullable: false })
  ageTo: number;

  @Column({ type: "enum", array: true, enum: Interest, nullable:
false })
  interests: Interest[];
}
```

Таблиця 3.3 - Склад таблиці UserProfile

Атрибут	Тип даних	Допустимі значення	Значення за замовчуванням	Обов'язковість	Примітки
id	UUID	Унікальний UUID	-	Так	Первинний ключ, автоінкремент
user	OneToOne	UserCore	-	Так	Зовнішній ключ, каскадне видалення
firstName	varchar(255)	Латиниця, кирилиця	-	Так	-
lastName	varchar(255)	Латиниця, кирилиця	-	Так	-
dateOfBirth	date	-	-	Ні	-
country	varchar(255)	Латиниця, кирилиця	-	Ні	-
city	varchar(255)	Латиниця, кирилиця	-	Ні	-
bio	varchar(300)	Латиниця, кирилиця	-	Так	-
gender	enum: Gender	Значення з Gender	-	Так	-
profilePhoto	varchar(255)	URL (латиниця)	-	Ні	URL фото
interests	array(enum: Interest)	Значення з Interest	-	Ні	-

Сутність UserProfile (лістинг 3.19) містить розширену інформацію про профіль користувача, включаючи персональні відомості та захоплення.

Лістинг 3.19 – Код сутності UserProfile

```
@Entity("users_profile")
export class UserProfile extends BaseEntity {
  @PrimaryGeneratedColumn("uuid")
  id: string;
  @OneToOne(() => UserCore, (user) => user.id, {
    onDelete: "CASCADE",})
  @JoinColumn({ name: "userId" })
  user: UserCore;
  @Column({
    type: "varchar",
    length: 255,
    nullable: false,})
  firstName: string;
  @Column({
    type: "varchar",
    length: 255,
    nullable: false,})
  lastName: string;
  @Column({
    type: "date",
    nullable: true,})
  dateOfBirth: Date;
```

```
  @Column({
    type: "varchar",
    length: 255,
    nullable: true,})
  country: string;
  @Column({
    type: "varchar",
    length: 255,
    nullable: true,})
  city: string;
  @Column({
    type: "varchar",
    length: 300,
    nullable: false,})
  bio: string;
  @Column({
    type: "enum",
    enum: Gender,
    nullable: false,})
  gender: Gender;
  @Column({ type: "varchar", length: 255, nullable: true })
  profilePhoto?: string;
  @Column({ type: "enum", enum: Interest, array: true, nullable:
true })
  interests?: Interest[];}
```

Таблиця 3.4 - Склад таблиці FriendshipRequest

Атрибут	Тип даних	Допустимі значення	Значення за замовчуванням	Обов'язковість	Примітки
id	UUID	Унікальний UUID	-	Так	Первинний ключ, автоінкремент
requesterId	UUID	Унікальний UUID	-	Так	Зовнішній ключ, індексований
requester	ManyToOne	UserCore	-	Так	Каскадне видалення
recipientId	UUID	Унікальний UUID	-	Так	Зовнішній ключ, індексований
recipient	ManyToOne	UserCore	-	Так	Каскадне видалення
status	enum: FriendshipStatus	Значення з FriendshipStatus	PENDING	Так	-
createdAt	timestamp	-	Поточна дата/час	Так	Автоматично генерується

Сутність FriendshipRequest (лістинг 3.20) містить дані про заявки на дружбу між користувачами системи.

Лістинг 3.20 – Код сутності FriendshipRequest

```
@Entity("friendship_requests")
export class FriendshipRequest extends BaseEntity {
```

```
  @PrimaryGeneratedColumn("uuid")
  id: string;

  @Index("friendship_request_requester_id_idx")
  @Column({ type: "uuid", nullable: false })
  requesterId: string;

  @ManyToOne(() => UserCore, (userCore) =>
userCore.sentFriendshipRequests, {
    onDelete: "CASCADE",
  })
  @JoinColumn({ name: "requesterId" })
  requester: UserCore;

  @Index("friendship_request_recipient_id_idx")
  @Column({ type: "uuid", nullable: false })
  recipientId: string;

  @ManyToOne(
    () => UserCore,
    (userCore) => userCore.receivedFriendshipRequests,
    {
      onDelete: "CASCADE",
    },
  )
  @JoinColumn({ name: "recipientId" })
  recipient: UserCore;
  @Column({
    type: "enum",
    enum: FriendshipStatus,
    default: FriendshipStatus.PENDING,
  })
  status: FriendshipStatus;
  @CreateDateColumn()
  createdAt: Date;
}
```

Таблиця 3.5 - Склад таблиці Friend

Атрибут	Тип даних	Допустимі значення	Значення за замовчуванням	Обов'язковість	Примітки
id	UUID	Унікальний UUID	-	Так	Первинний ключ, автоінкремент
firstUserId	UUID	Унікальний UUID	-	Так	Зовнішній ключ, індексований
firstUser	ManyToOne	UserCore	-	Так	Каскадне видалення
secondUserId	UUID	Унікальний UUID	-	Так	Зовнішній ключ, індексований
secondUser	ManyToOne	UserCore	-	Так	Каскадне видалення
users	ManyToMany	UserCore[]	-	Ні	Через таблицю зв'язку
establishedAt	timestamp	-	Поточна дата/час	Так	Автоматично генерується

Сутність Friend (лістинг 3.21) містить інформацію про сформовані дружні відносини між користувачами.

Лістинг 3.21 – Код сутності Friend

```
@Entity("friends")
@Check(`"firstUserId" < "secondUserId"`)
export class Friend extends BaseEntity {
  @PrimaryGeneratedColumn("uuid")
  readonly id: string;
  @Index("first_user_id_friend_idx")
  @Column({
    type: "uuid",
    nullable: false,
  })
  firstUserId: string;
  @ManyToOne(() => UserCore, {
    onDelete: "CASCADE",
    nullable: false,
  })
  @JoinColumn({ name: "firstUserId", referencedColumnName: "id" })
  readonly firstUser: UserCore;
  @Index("second_user_id_friend_idx")
  @Column({
    type: "uuid",
    nullable: false,
  })
  secondUserId: string;
  @ManyToOne(() => UserCore, {
    onDelete: "CASCADE",
    nullable: false,
  })
  @JoinColumn({ name: "secondUserId", referencedColumnName: "id" })
  readonly secondUser: UserCore;
  @ManyToMany(() => UserCore, (user: UserCore) => user.friends)
  readonly users: UserCore[];
  @CreateDateColumn()
  readonly establishedAt: Date;
  @BeforeInsert()
  ensureCanonicalOrder(): void {
    if (this.firstUserId > this.secondUserId) {
      const temp: string = this.firstUserId;
      this.firstUserId = this.secondUserId;
      this.secondUserId = temp;}}}
```

3.2.2 Створення модуля автентифікації

Система автентифікації реалізує ключові функції входу, реєстрації, отримання інформації про користувача, виходу та поновлення токенів для

програми "MateUp". Реалізація здійснена через контролер AuthController, який взаємодіє з сервісом AuthService для виконання бізнес-логіки. Автентифікація базується на JWT токенах із можливістю оновлення через RefreshAuthGuard, а контроль доступу здійснюється через анотації @Public() та @UseGuards().

Контролер AuthController (лістинг 3.22) має тег @ApiTags("auth") і базовий маршрут /auth. Включає п'ять головних ендпоінтів для керування автентифікацією: публічний ендпоінт логіну (/login) з анотацією @Public() приймає користувацькі дані через UserLogInDto та повертає LoginResponse з токенами, використовуючи HTTP статус 200; публічний ендпоінт реєстрації (/sign-up) з гардом DoesUserExistGuard для валідації унікальності користувача, приймає UserSignUpDto та повертає SignUpResponse; захищений ендпоінт отримання користувацьких даних (/user) з JwtAuthGuard, що повертає об'єкт UserCore для авторизованого користувача; захищений ендпоінт виходу (/logout) з JwtAuthGuard, що виконує logout користувача та повертає повідомлення про успішний вихід; публічний ендпоінт оновлення токенів (/refresh) з RefreshAuthGuard, що приймає refreshToken та повертає нові Tokens на основі JwtPayload.

Лістинг 3.22 – Код контролера AuthController

```
@ApiTags("auth")
@Controller("auth")
export class AuthController {
  constructor(private readonly authService: AuthService) {}
  @ApiBody({ type: UserLogInDto })
  @Public()
  @Post("/login")
  @HttpCode(HttpStatus.OK)
  async login(@Body() userLogInDto: UserLogInDto):
Promise<LoginResponse> {return await
this.authService.login(userLogInDto);}

  @Public()
  @UseGuards(DoesUserExistGuard)
  @Post("/sign-up")
  @HttpCode(HttpStatus.OK)
  async signUp(@Body() userSignUpDto: UserSignUpDto):
Promise<SignUpResponse> {
    return await this.authService.signUp(userSignUpDto);
```

```
  @UseGuards(JwtAuthGuard)
  @Get("/user")
  @HttpCode(HttpStatus.OK)
  async getUser(@GetCurrentUserId() userId: string):
Promise<UserCore> {
    return await this.authService.getUser(userId);}
  @Post("/logout")
  @HttpCode(HttpStatus.OK)
  async logout(@GetCurrentUserId() userId: string): Promise<string>
{await this.authService.logout(userId);
return "Logged out successfully";}
  @Public()
  @UseGuards(RefreshAuthGuard)
  @Post("/refresh")
  @HttpCode(HttpStatus.OK)
  async refreshTokens(
    @GetCurrentUser() user: UserCore,
    @GetCurrentUserProperty("refreshToken") refreshToken: string,
  ): Promise<Tokens> { const jwtPayload: JwtPayload = { sub:
user.id, email: user.email,};
    return await this.authService.refreshTokens(jwtPayload,
refreshToken);
  }
}
```

3.2.3 Створення модуля профілю користувача

Система управління профілем забезпечує операції створення, отримання та редагування профілю авторизованого користувача в програмі "MateUp". Реалізація здійснена через контролер ProfileController, який використовує сервіс ProfileService для обробки бізнес-логіки. Всі ендпоінти захищені через JwtAuthGuard, що гарантує доступ виключно для авторизованих користувачів.

Контролер ProfileController (код наведено у лістинг 3.23) має тег @ApiTags("Profile") і базовий маршрут /profile. Містить три основні ендпоінти для роботи з профілем користувача: ендпоінт створення профілю (/create-profile) з методом POST приймає дані через CreateProfileDto та userId, повертаючи створений UserProfile з HTTP статусом 201; ендпоінт отримання профілю (/get-profile) з методом GET повертає UserProfile за userId з HTTP статусом 200; ендпоінт оновлення профілю (/update-profile) з методом PUT оновлює профіль через UpdateProfileDto та повертає оновлений UserProfile з HTTP статусом 200.

Лістинг 3.23 – Реалізація контролера ProfileController

```
@ApiTags("Profile")
@Controller("profile")
@UseGuards(JwtAuthGuard)
export class ProfileController {
  constructor(private readonly profileService: ProfileService) {}

  @ApiBearerAuth()
  @Post("/create-profile")
  @HttpCode(HttpStatus.CREATED)
  async createProfile(
    @GetCurrentUserId() userId: string,
    @Body() createProfileDto: CreateProfileDto,
  ): Promise<UserProfile> {
    return await this.profileService.createProfile(userId,
createProfileDto);
  }

  @ApiBearerAuth()
  @Get("/get-profile")
  @HttpCode(HttpStatus.OK)
  async getProfile(@GetCurrentUserId() userId: string):
Promise<UserProfile> {
    return await this.profileService.getProfile(userId);
  }

  @ApiBearerAuth()
  @Put("/update-profile")
  @HttpCode(HttpStatus.OK)
  async updateProfile(
    @GetCurrentUserId() userId: string,
    @Body() updateProfileDto: UpdateProfileDto,
  ): Promise<UserProfile> {
    return await this.profileService.updateProfile(userId,
updateProfileDto);
  }
}
```

3.2.4 Створення модуля керування файлами

Система керування файлами надає функціональність завантаження, отримання, перегляду переліку та видалення зображень профілю користувачів у програмі "MateUp". Реалізація здійснена через контролер FileController, який взаємодіє з сервісом FileService для бізнес-логіки та WinstonService для логування помилок. Всі ендпоінти потребують JWT авторизації через @ApiBearerAuth().

Контролер FileController (фрагмент коду наведений в лістингу 3.24, повний код у додатку А має тег @ApiTags("Files") і базовий маршрут /file. Включає

чотири головні ендпоінти для роботи з файлами: ендпоінт завантаження зображення (POST /file) з методом POST дозволяє upload фото профілю, використовує FileInterceptor для обробки файлу та ParseFilePipeBuilder для валідації типу (VALID_UPLOADS_MIME_TYPES) та розміру (MAX_PICTURE_SIZE_IN_BYTES), повертає результат з HTTP статусом 201; ендпоінт отримання зображення (GET /file/:filename) з методом GET повертає потік даних зображення за назвою файлу для авторизованого користувача з HTTP статусом 200; ендпоінт отримання переліку зображень (GET /file) з методом GET повертає масив метаданих (FileDetails[]) всіх зображень користувача з HTTP статусом 200; ендпоінт видалення зображення (DELETE /file/:filename) з методом DELETE видаляє зображення за назвою файлу, повертаючи результат з HTTP статусом 200.

Лістинг 3.24 – Фрагмент коду контролера FileController

```
@Controller("file")
export class FileController {
  constructor(
    private readonly fileService: FileService,  ) {}
  @Post()
  @HttpCode(HttpStatus.CREATED)
  @UseInterceptors(FileInterceptor("photo"))
  async uploadPhoto(
    @UploadedFile(
      new ParseFilePipeBuilder()
        .addValidator(
          new UploadFileTypeValidator({
            fileType: VALID_UPLOADS_MIME_TYPES,}),)
 .addMaxSizeValidator({ maxSize: MAX_PICTURE_SIZE_IN_BYTES, message:
`Photo should be less than ${MAX_PICTURE_SIZE_IN_BYTES} bytes`,})
   .build({ errorHttpStatusCode:HttpStatus.UNPROCESSABLE_ENTITY }),)
    file: Express.Multer.File,
    @Request() req: any,
    @Response() res: any,) {const userId = req?.user?.id;
if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
      return await this.fileService.uploadFile(file, userId);}
  @Get(":filename")
  async getPhoto(@Param("filename") filename: string,@Request() req:
any,@Response({ passthrough: true }) res: any,) {const userId =
req?.user?.id;
if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
```

```
      return await this.fileService.getFile(userId, filename);}
  @Get()
  @HttpCode(HttpStatus.OK)
  async getPhotos(@Request() req: any,@Response() res: any,):
Promise<FileDetails[]> {const userId = req?.user?.id;
if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
      return await this.fileService.getFiles(userId);}}}
```

3.2.5 Створення модуля керування головною сторінкою

Система керування головною сторінкою надає функціональність отримання персоналізованого переліку користувачів для авторизованих користувачів у програмі "MateUp". Реалізація здійснена через контролер FeedController, який використовує сервіс FeedService для бізнес-логіки. Ендпоінт потребує JWT авторизації через декоратор @UseGuards(JwtAuthGuard) та @ApiBearerAuth().

Контролер FeedController (код наведено у лістинг 3.25) має тег @ApiTags("feed") і базовий маршрут /feed. Містить один головний ендпоінт для роботи зі стрічкою: отримання рекомендацій (POST /feed) з методом POST повертає масив користувачів (UserCore[]) для авторизованого користувача на основі його userId з HTTP статусом 200.

Лістинг 3.25 – Реалізація контролера FeedController

```
@ApiTags("feed")
@Controller("feed")
@UseGuards(JwtAuthGuard)
export class FeedController {
 constructor(private readonly feedService: FeedService) {}

 @ApiBearerAuth()
 @HttpCode(HttpStatus.OK)
 @Post()
 async getFeed(@GetCurrentUserId() userId: string):
Promise<UserCore[]> {
   try {
     const response = await this.feedService.getFeed(userId);
     return response;
   } catch (error) {
     throw new HttpException(
       {
          type: ResponseType.ERROR,
          message: error.message || "Error occurred while retrieving
feed",
```

```
          data: null,
        },
        HttpStatus.INTERNAL_SERVER_ERROR,
      );
    }
  }
}
```

3.2.6 Створення модуля керування фільтрами

Система керування фільтрами надає функціональність отримання та редагування фільтрів для персоналізації стрічки користувачів у програмі "MateUp". Реалізація здійснена через контролер FiltersController, який використовує сервіс FiltersService для бізнес-логіки та WinstonService для логування помилок. Всі ендпоінти потребують JWT авторизації через декоратор @UseGuards(JwtAuthGuard) та @ApiBearerAuth().

Контролер FiltersController ((фрагмент коду наведений в лістингу 3.26, повний код у додатку А) має тег @ApiTags("filters") і базовий маршрут /filters. Включає два головні ендпоінти для роботи з фільтрами: ендпоінт отримання фільтрів (GET /filters/get-filters) з методом GET повертає налаштування фільтрів користувача (UserFilters) на основі його userId з HTTP статусом 200; ендпоінт оновлення фільтрів (PUT /filters/update-filters) з методом PUT дозволяє редагувати налаштування фільтрів користувача на основі даних у тілі запиту (UpdateFiltersDto), повертає оновлені фільтри (UserFilters) з HTTP статусом 200.

Лістинг 3.26 – Фрагмент коду контролера FiltersController

```
@Controller("filters")
@UseGuards(JwtAuthGuard)
export class FiltersController {
  constructor(
    private readonly filtersService: FiltersService,  ) {}

  @ApiBearerAuth()
  @HttpCode(HttpStatus.OK)
  @Get("/get-filters")
  async getFilters(@GetCurrentUserId() userId: string):
Promise<UserFilters> {
      return await this.filtersService.getFilters(userId);
    }
```

```
  @ApiBearerAuth()
  @HttpCode(HttpStatus.OK)
  @Put("/update-filters")
  async updateFilters(
    @Body() updateFilterDto: UpdateFiltersDto,
    @GetCurrentUserId() userId: string,
  ): Promise<UserFilters> {
      return await
this.filtersService.updateFilters(updateFilterDto, userId);
    }
}
```

3.2.7 Створення модуля керування друзями

Система керування друзями надає функціональність управління дружніми відносинами в програмі "MateUp", включаючи отримання переліку друзів, відправлення та обробку заявок на дружбу, а також видалення друзів. Реалізація здійснена через контролер FriendsController, який використовує сервіси FriendsService та FriendshipRequestService для бізнес-логіки. Всі ендпоінти потребують JWT авторизації через декоратор @UseGuards(JwtAuthGuard) та @ApiBearerAuth().

Контролер FriendsController (фрагмент коду наведений в лістингу 3.27, повний код у додатку А) має тег @ApiTags("friends") і базовий маршрут /friends. Включає вісім головних ендпоінтів для роботи з дружніми відносинами: ендпоінт отримання переліку друзів (GET /friends) повертає масив друзів користувача (UserCore[]) на основі його userId з HTTP статусом 200; ендпоінт отримання відправлених заявок на дружбу (GET /friends/sent-requests) повертає масив користувачів (UserCore[]), яким поточний користувач відправив заявки на дружбу з HTTP статусом 200; ендпоінт отримання отриманих заявок на дружбу (GET /friends/received-requests) повертає масив користувачів (UserCore[]), від яких поточний користувач отримав заявки на дружбу з HTTP статусом 200; ендпоінт відправлення заявки на дружбу (POST /friends/request-friendship/:recipientId) дозволяє відправити заявку на дружбу іншому користувачу за його recipientId, повертає результат (RequestFriendshipResponse) з HTTP статусом 200; ендпоінт прийняття заявки на дружбу (POST /friends/accept-friendship/:recipientId) дозволяє прийняти заявку на дружбу від іншого користувача за його recipientId, повертає

результат (AcceptFriendshipResponse) з HTTP статусом 200; ендпоінт відхилення заявки на дружбу (POST /friends/reject-friendship/:recipientId) дозволяє відхилити заявку на дружбу від іншого користувача за його recipientId, повертає результат (AcceptFriendshipResponse) з HTTP статусом 200; ендпоінт видалення друга (DELETE /friends/:friendId) видаляє дружні відносини з користувачем за його friendId, повертає результат (DeleteFriendResponse) з HTTP статусом 200; ендпоінт видалення всіх друзів (DELETE /friends) видаляє всі дружні відносини поточного користувача, повертає результат (DeleteFriendResponse) з HTTP статусом 200.

Лістинг 3.27 – Фрагмент коду FriendsController

```
@Controller("friends")
@UseGuards(JwtAuthGuard)
export class FriendsController {
constructor(
private readonly friendsService: FriendsService,
private readonly friendshipRequestService:
FriendshipRequestService,) {}
@ApiBearerAuth()
@Get()
@HttpCode(HttpStatus.OK)
async getFriends(
@GetCurrentUserId() userId: string,
): Promise<UserCore[]> {
return await this.friendsService.getFriends(userId);
@Get("/sent-requests")
@HttpCode(HttpStatus.OK)
async getSentFriendshipRequests(
@GetCurrentUserId() userId: string,
): Promise<UserCore[]> {
return await
this.friendshipRequestService.getSentFriendshipRequests(userId);}
@Get("/received-requests")
@HttpCode(HttpStatus.OK)
async getReceivedFriendshipRequests(
@GetCurrentUserId() userId: string,
): Promise<UserCore[]> {
return await
this.friendshipRequestService.getReceivedFriendshipRequests(userId);
@ApiBearerAuth()
@Post("/request-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
async requestFriendship(
@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string,
): Promise<RequestFriendshipResponse> {
```

```
return await this.friendshipRequestService.requestFriendship(userId,
recipientId);
@ApiBearerAuth()
@Post("/accept-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
async acceptFriendship(
@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string,
): Promise<AcceptFriendshipResponse> {
return await
this.friendshipRequestService.acceptFriendshipRequest(userId,
recipientId);
@ApiBearerAuth()
@Post("/reject-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
async rejectFriendship(
@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string,
): Promise<AcceptFriendshipResponse> {
return await
this.friendshipRequestService.rejectFriendshipRequest(userId,
recipientId);}
(Енлпоінти для видаленння конкретного друга та усіх друзів)
```

4 ВИПРОБУВАННЯ КОМП’ЮТЕРНОЇ СИСТЕМИ

Після завершення розробки веб-системи для пошуку друзів за інтересами було проведено комплексне тестування її функціональності. Випробування включали перевірку всіх основних компонентів системи: реєстрації користувачів, створення профілів, пошуку користувачів за інтересами та функціональності дружби.

4.1 Випробування процесу реєстрації

Першим етапом випробування була перевірка форми реєстрації нових користувачів (рис.4.1). Система успішно обробляє введення обов'язкових полів:

ім'я користувача, електронна адреса та пароль. Форма містить привітальне повідомлення "Welcome, future mate!" з емодзі, що створює дружню атмосферу для нових користувачів. Під час випробування було перевірено валідацію введених даних, включаючи перевірку формату email та складності паролю, функціонування кнопки показу та приховування паролю, коректне відображення помилок при некоректному введенні, перенаправлення на сторінку створення профілю після успішної реєстрації та роботу посилання для користувачів, які вже мають акаунт.

Паралельно було випробувано форму авторизації (рис.5.2) існуючих користувачів. Форма входу містить поля для введення електронної адреси та паролю з можливістю приховування та показу введеного пароля. Система включає функціональність відновлення паролю через посилання "Forgot password?", що дозволяє користувачам відновити доступ до свого акаунту. Тестування показало коректну роботу валідації облікових даних, відображення повідомлень про помилки при введенні неправильних даних та успішне перенаправлення на головну сторінку після вдалої авторизації. Додатково було перевірено посилання "Sign up now" для швидкого переходу до форми реєстрації нових користувачів.

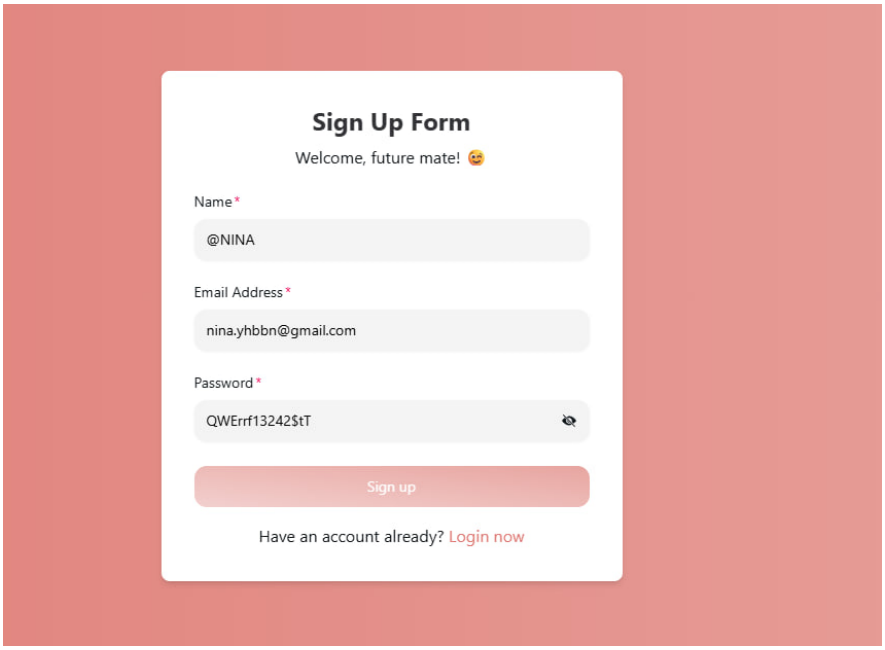


Рисунок 4.1 – Форма реєстрації

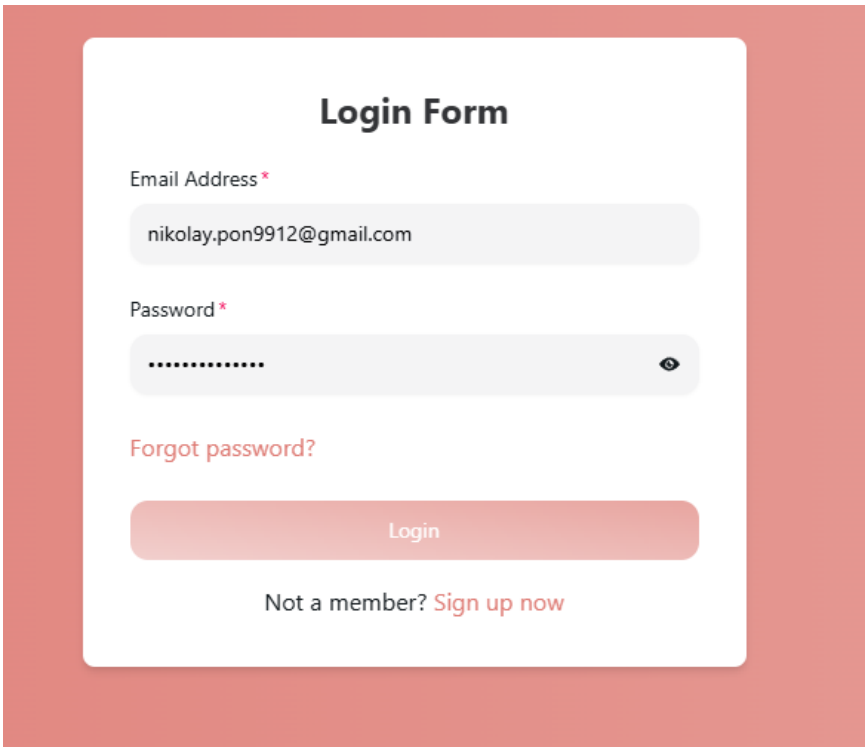


Рисунок 4.2 – Форма авторизації

4.2 Випробування створення профілю

Наступним етапом було випробування функціональності створення користувацького профілю (рис. 4.3 та 4.4). Система надає можливість користувачам заповнити детальну інформацію про себе, включаючи особисті дані як ім'я, прізвище та дата народження, географічну інформацію про країну та місто проживання, стать користувача через радіокнопки, завантаження фотографії профілю та текстовий опис у полі біографії. Випробування показало коректну роботу всіх полів форми, включаючи випадаючі списки для вибору країни та міста, календар для вибору дати народження. Особливу увагу було приділено функціональності завантаження зображень та відображення попереднього перегляду фотографії.

Критично важливою частиною системи є функціональність вибору інтересів, оскільки саме на їх основі відбувається пошук потенційних друзів.

Система пропонує широкий спектр категорій інтересів, включаючи технічні напрямки як програмування, технології та наука, творчі сфери як мистецтво, музика, фотографія та письменство, активні захоплення як спорт, фітнес та подорожі, побутові інтереси як кулінарія, читання та ігри, соціальні активності як волонтерство, вивчення мов та підприємництво, а також розважальні напрямки як кіно, мода, природа та історія. Під час випробування було перевірено можливість вибору декількох інтересів одночасно, візуальну індикацію обраних інтересів через зміну кольору кнопок, лічильник обраних інтересів, збереження вибраних інтересів у базі даних та коректну роботу кнопки збереження профілю.

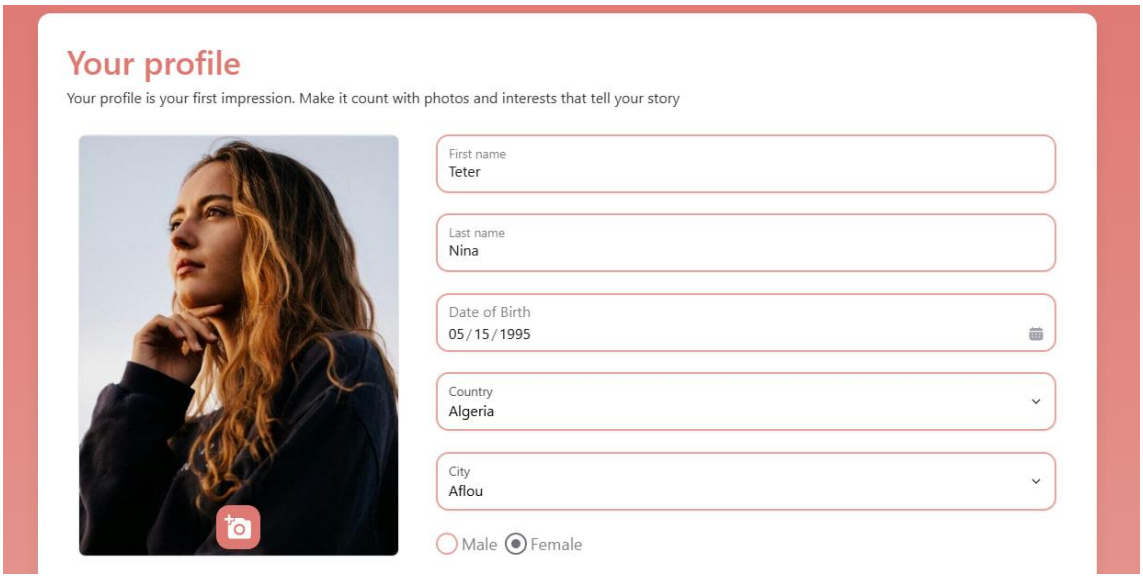


Рисунок 4.3 – Форма для створення та редагування профілю (частина 1)

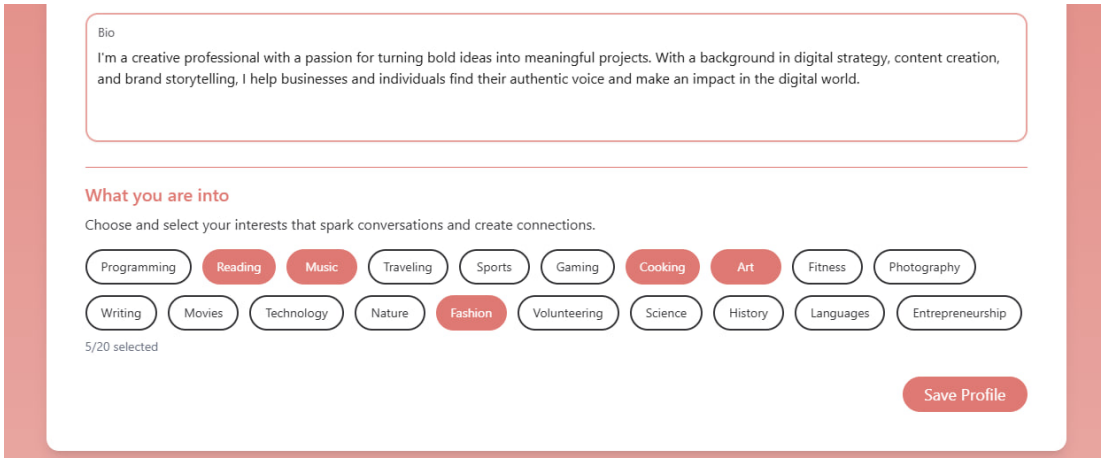


Рисунок 4.4 – Форма для створення та редагування профілю (частина 2)

4.3 Випробування системи пошуку

Основна функціональність системи - пошук користувачів за спільними інтересами - була ретельно випробувана. Головна сторінка пошуку (рис.4.5) містить заклик "Find Your People: Connect with Others Who Share Your Interests" та включає пошукове поле для введення ключових слів, слайдер для вибору вікового діапазону від 18 до 100 років з можливістю вибору конкретних категорій, фільтри за інтересами, ідентичні тим, що використовуються при створенні профілю, та кнопку скидання всіх фільтрів. Випробування показало ефективну роботу алгоритму пошуку, який коректно фільтрує користувачів за заданими критеріями та відображає результати у вигляді карток профілів з основною інформацією про користувача.

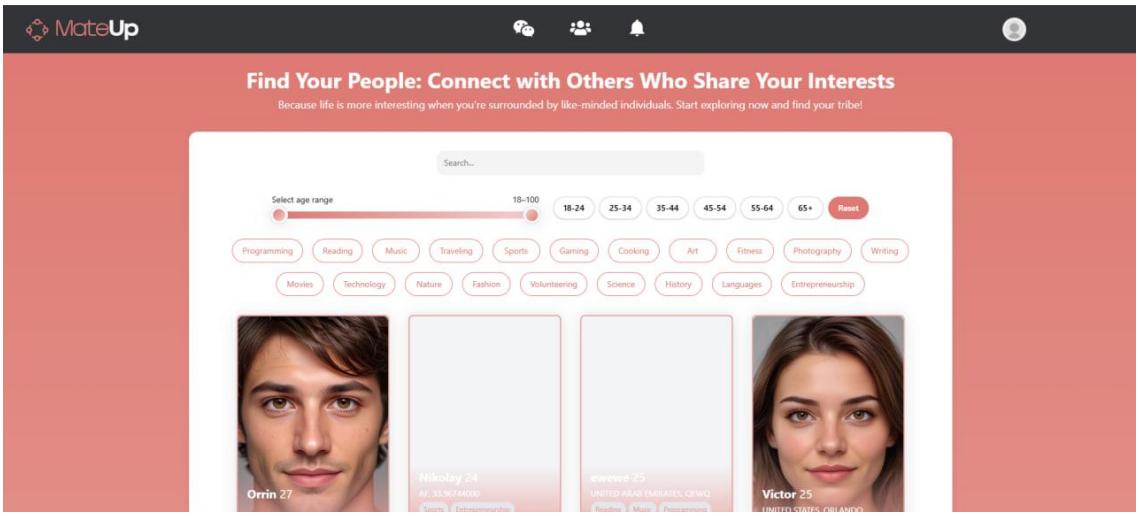


Рисунок 4.5 – Головна сторінка для пошуку

4.4 Випробування системи дружби

Останнім етапом випробування була перевірка функціональності управління дружніми зв'язками (рис.4.6). Система включає два основні розділи:

список підтверджених друзів та вхідні запити на дружбу. Під час випробування було перевірено можливість надсилання запитів на дружбу іншим користувачам, отримання та відображення вхідних запитів, прийняття або відхилення запитів на дружбу, видалення користувачів зі списку друзів, функціональність пошуку серед друзів та ініціювання чату з друзями через кнопку повідомлень.

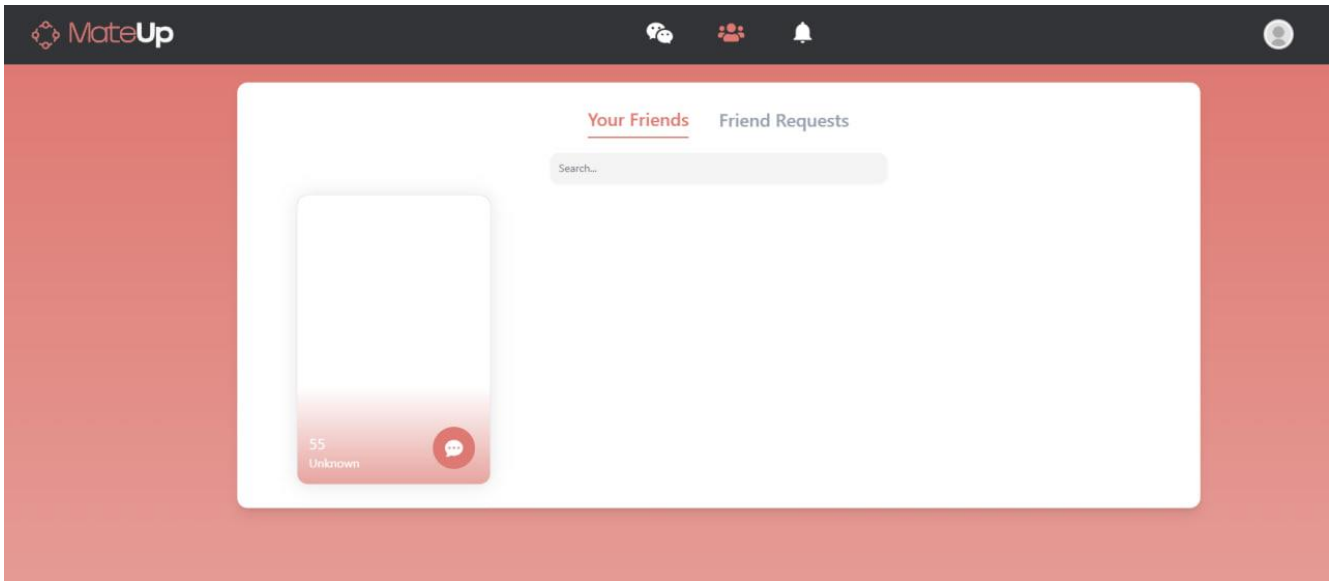


Рисунок 4.6 – Сторінка друзів

Всі протестовані функції працюють стабільно та відповідають технічним вимогам. Система демонструє високу надійність, зручний користувацький інтерфейс та ефективну роботу алгоритмів пошуку за інтересами.

ВИСНОВКИ

У рамках дипломної роботи було успішно розроблено та впроваджено веб-систему для пошуку друзів за інтересами, яка забезпечує ефективне знайомство та взаємодію людей зі спільними захопленнями.

На першому етапі роботи було проведено ґрунтовний аналіз існуючих архітектурних рішень, включаючи одношарову, клієнт-серверну, тришарову та мікросервісну архітектури. Детальне порівняння дозволило обрати оптимальну архітектуру для поставленої задачі. Також було здійснено комплексний огляд сучасних технологій розробки, зокрема фреймворків React, Angular, Vue та Next.js для клієнтської частини, а також серверних рішень Node.js, PHP, Java та Ruby. Окрему увагу приділено вибору системи керування базами даних через порівняння PostgreSQL, MongoDB, Firebase та SQLite, що дозволило визначити найбільш підходяще рішення для специфіки проекту.

Етап формування вимог до програмного забезпечення включав детальний аналіз функціональних та нефункціональних потреб системи з подальшою розробкою повноцінної специфікації вимог SRS. Це забезпечило чітке розуміння цільової функціональності та технічних обмежень проекту.

Процес моделювання програмного забезпечення передбачав створення комплексу UML-діаграм, включаючи діаграми варіантів використання для відображення взаємодії користувачів з системою, діаграми послідовності для деталізації алгоритмів роботи та розгорнуті описи прецедентів для точного визначення поведінки системи в різних сценаріях використання.

Фаза безпосередньої розробки включала створення клієнтської частини з інтуїтивно зрозумілим користувацьким інтерфейсом та розробку серверної частини з ефективною обробкою запитів. Особливу увагу приділено проєктуванню структури бази даних, що включає таблиці UserCore для зберігання основних даних користувачів, UserProfile для детальної інформації про профілі та FriendshipRequest для управління запитами на встановлення дружніх зв'язків.

Завершальний етап випробувань системи продемонстрував надійність та ефективність розробленого рішення. Тестування охопило всі ключові компоненти: процедури реєстрації та авторизації користувачів, функціональність створення та редагування профілів, алгоритми пошуку за інтересами та віковими категоріями, а також систему управління дружніми контактами.

Розроблена веб-система повністю задовольняє поставлені цілі та вимоги. Вона надає користувачам зручний інструмент для пошуку однодумців, сприяє формуванню нових соціальних зв'язків та створює платформу для спілкування людей зі схожими інтересами. Система характеризується високою функціональністю, стабільністю роботи та масштабованістю, що дозволяє в майбутньому розширювати її можливості та адаптувати до зростаючих потреб користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Web Application Architecture: How to Choose the Best for Your Product. URL: https://mobidev.biz/blog/web–application–architecture–types/ (дата звернення: 26.04.2025);

2 Understanding Software Architecture: Choosing the Right Model for Your Application. URL: https://logicloom.in/understanding–software–architecture–choosing–the–right–model–for–your–application/ (дата звернення: 26.04.2025);

3 What is monolithic architecture? URL: https://artkai.io/blog/mach–vs–monolithic–architecture (дата звернення: 26.04.2025);

4 Monolithic Architecture: Powerhouse for Rapid Development. URL: https://intellisoft.io/monolithic–architecture–powerhouse–for–rapid–development/ (дата звернення: 26.04.2025);

5 Web Application Architecture: The Latest Guide 2025. URL: https://peiko.space/blog/article/web–application–architecture/ (дата звернення: 26.04.2025);

6 Web Architecture. URL: https://en.ryte.com/wiki/Web_Architecture/ (дата звернення: 26.04.2025);

7 What is three–tier architecture? URL: https://www.ibm.com/think/topics/three–tier–architecture (дата звернення: 26.04.2025);

8 3–Tier Web Architecture. URL: https://arokiait.com/3–tier–web–architecture.htm (дата звернення: 26.04.2025);

9 Microservice architecture style. URL: https://learn.microsoft.com/en–us/azure/architecture/guide/architecture–styles/microservices (дата звернення: 26.04.2025);

10 Microservices architecture and design: A complete overview. URL: https://vfunction.com/blog/microservices–architecture–guide/ (дата звернення: 26.04.2025);

11 Advantages and Benefits of React JS: The Essential Tool for Modern Developers. URL: https://maybe.works/blogs/react–js–advantages (дата звернення: 26.04.2025);

12 React. The library for web and native user interfaces Developers. URL: https://react.dev/ (дата звернення: 26.04.2025);

13 Pros and Cons of Angular Framework You Need to Know. URL: https://www.robinwaite.com/blog/pros–and–cons–of–angular–framework–you–need–to–know (дата звернення: 26.04.2025);

14 Getting started with Angular. URL: https://developer.mozilla.org/en–US/docs/Learn_web_development/Core/Frameworks_libraries/Angular_getting_started (дата звернення: 26.04.2025);

15 Pros And Cons Of Using Vue.Js for Web App Development. URL: https://ficustechnologies.com/blog/pros–and–cons–of–using–vue–js–for–web–app–development/ (дата звернення: 26.04.2025);

16 The Progressive. JavaScript Framework. URL: https://vuejs.org/ (дата звернення: 26.04.2025);

17 Overview of Next.js for Modern Web Apps: Pros, Cons, and Use Cases. URL: https://leobit.com/blog/overview–of–next–js–for–modern–web–apps–pros–cons–and–use–cases/ (дата звернення: 26.04.2025);

18 Understanding Next.js: Key Features, Benefits, and Use Cases. URL: https://www.prismetric.com/understanding–next–js/ (дата звернення: 26.04.2025);

19 Nest.js. Introduction. URL: https://docs.nestjs.com/ (дата звернення: 26.04.2025);

20 Laravel vs PHP: Comparing Frameworks, Pros, Cons, and Use Cases. URL: https://www.milesweb.ae/blog/technology–hub/laravel–vs–php/ (дата звернення: 26.04.2025);

21 A Comparison Between Spring and Spring Boot. URL: https://www.baeldung.com/spring–vs–spring–boot (дата звернення: 26.04.2025);

22 Ruby: Web Development Explained. URL: https://www.netguru.com/glossary/ruby–web–development (дата звернення: 26.04.2025);

23 What is PostgreSQL and Everything You Need to Know. URL: https://techvify–software.com/what–is–postgresql/ (дата звернення: 26.04.2025);

24 MongoDB: Advantages and Disadvantages Your Enterprise Should Consider. URL: https://www.koombea.com/blog/mongodb–advantages–and–disadvantages/ (дата звернення: 26.04.2025);

25 What is Firebase? A Beginner's Guide. URL: https://www.theknowledgeacademy.com/blog/what–is–firebase/ (дата звернення: 26.04.2025);

26 What is SQLite? Features and Uses. URL: https://www.mygreatlearning.com/blog/sqlite–tutorial/ (дата звернення: 26.04.2025);

ДОДАТОК А

Лістинг А.1 – Файл NavBar.tsx

```
export default function NavBar() {
  const pathname = usePathname();
  return (<Navbar >
      <div className="flex w-full items-center">
        <NavbarBrand className="flex-shrink-0">
          <HeroLink href="/"> <LogoIcon /> </HeroLink>
        </NavbarBrand>
          <NavbarContent >
            <NavbarItem className="group">
              <HeroLink href="/messages">
                <BsWechat ${pathname === "/messages" ? "text-coral-
red" : "text-white"}`}/> </HeroLink>
            </NavbarItem>
            <NavbarItem className="group">
              <HeroLink href="/friends">
                <FaUsers${pathname === "/friends" ? "text-coral-red"
: "text-white"}`}/> </HeroLink>
            </NavbarItem>
            <NavbarItem className="group">
              <HeroLink href="/notifications">
                <IoMdNotifications${pathname === "/notifications"?
"text-coral-red": "text-white"}`}/> </HeroLink>
            </NavbarItem>
          </NavbarContent>
          <NavbarContent>
              <NavbarItem>
<Button as={Link} variant="light"href="/auth/login"> Login</Button>
              </NavbarItem>
              <NavbarItem>
<Button as={Link}variant="light" href="/auth/signup">SignUp</Button>
              </NavbarItem>
              <UserMenu />
          </NavbarContent></div></div></Navbar>);}
```

Лістинг А.2 – Файл LoginForm.tsx

```
export default function LoginForm() { const {register, handleSubmit,
formState: { isValid, errors },} = useForm<loginSchema>({resolver:
zodResolver(loginSchema),mode: "onTouched",});
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const router = useRouter();
  const [isVisible, setIsVisible] = React.useState(false);
  const toggleVisibility = () => setIsVisible(!isVisible);
```

```
  const onSubmit: SubmitHandler<loginSchema> = async (data) =>
{setError(null);setIsLoading(true);
 try {const response = await api.post(API_ROUTES.LOGIN, {json:
data,}).json<{status: string; message: string | null; data: { user:
IUser; accessToken: string; refreshToken: string };}>();
      console.log("Відповідь сервера:", response);
 if (response.status === "success" && response.data.accessToken) {
const authData = { user: {username: response.data.user.username,
email: response.data.user.email,},
  tokens: {accessToken: response.data.accessToken, refreshToken:
response.data.refreshToken, },};
   localStorage.setItem("auth_data", JSON.stringify(authData));
 response.data.user.isProfileFilled ? router.push("/") :
router.push("/profile");}
 } catch (error: any) {const errorMessage =
error.response?.data?.error || "Помилка при вході";
setError(errorMessage); console.error("Error:", error); } finally {
  setIsLoading(false); }};
  return (
    <div className="h-full overflow-hidden no-scrollbar flex ">
      <div className="bg-white p-8 rounded-lg shadow-md w-full ">
        <h1 className="text-2xl >Login Form </h1>
        {error && (<div className="text-red-500 ">{error}</div> )}
        <Form onSubmit={handleSubmit(onSubmit)}>
          <Input label="Email Address" type="email"
placeholder="Enter your email" {...register("email")}
isInvalid={!!errors.email} errorMessage={errors.email?.message}/>
          <Input label="Password" type={isVisible ? "text" :
"password"} placeholder="Password" {...register("password")}
isInvalid={!!errors.password}
errorMessage={errors.password?.message} endContent={
              <button type="button" onClick={toggleVisibility} aria-
label={isVisible ? "Hide password" : "Show password"} >
                {isVisible ? ( <IoIosEyeOff " /> ) : ( <IoIosEye/>)}
              </button>}/>
          <div className="text-right mb-4">
       <Link href="/forgot-password" >Forgot password? </Link>
          </div>
          <Button type="submit" isDisabled={!isValid || isLoading}
isLoading={isLoading}> Login </Button>
        </Form>
        <div > Not a member?{" "}
          <Link href="/auth/signup" > Sign up now</Link>
        </div> </div> </div>);}
```

Лістинг А.3 – Файл SignUpForm.tsx

```
export default function SignUpForm() {
const { register, handleSubmit, formState: { isValid, errors },} =
useForm<signupSchema>({resolver: zodResolver(signupSchema), mode:
"onTouched",});
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
```

```
  const router = useRouter();
  const [isVisible, setIsVisible] = React.useState(false);
  const toggleVisibility = () => setIsVisible(!isVisible);
  const onSubmit: SubmitHandler<signupSchema> = async (data) => {
    setError(null);
    setIsLoading(true);
    try { const response = await api.post(API_ROUTES.SIGNUP, { json:
data, }).json<{status: string; message: string | null; data: { user:
IUser; accessToken: string; refreshToken: string };}>();
      console.log("Відповідь сервера:", response);

  if (response.status === "success" && response.data.accessToken) {
const authData = { user: { username: response.data.user.username,
email: response.data.user.email, }, tokens: { accessToken:
response.data.accessToken,refrehToken: response.data.refreshToken,},
isProfileFilled: response.data.user.isProfileFilled, };
        localStorage.setItem("auth_data", JSON.stringify(authData));
        router.push("/profile");}} catch (error: any) {
const errorMessage=error.response?.data?.error||"Помилка при вході";
      setError(errorMessage);
console.error("Помилка:", error); } finally {setIsLoading(false);}};
  return (
    <div className="h-full overflow-hidden no-scrollbar flex ">
      <div className="bg-white p-8 rounded-lg shadow-md w-full ">
        <div className="mb-6 text-center">
          <h1 className="text-2xl mb-2 >Sign Up Form </h1>
          <p>Welcome, future mate! &#x1F609;</p>
        </div>
{error && (<div className="text-red-500 mb-4">{error}</div>)}
        <Form onSubmit={handleSubmit(onSubmit)}>
          <Input label="Name" type="username" placeholder="Enter
your name" {...register("username")} isInvalid={!!errors.username}
errorMessage={errors.username?.message}/>
          <Input label="Email Address" type="email"
placeholder="Enter your email" {...register("email")}
isInvalid={!!errors.email} errorMessage={errors.email?.message}/>
          <Input label="Password" type={isVisible ? "text" :
"password"} placeholder="Password" {...register("password")}
isInvalid={!!errors.password}errorMessage={errors.password?.message}
            endContent={
              <button type="button" onClick={toggleVisibility}aria-
label={isVisible ? "Hide password" : "Show password"}>
                {isVisible ? (<IoIosEyeOff/>) : (<IoIosEye />)}
              </button>}/>
          <Button type="submit" isDisabled={!isValid || isLoading}
isLoading={isLoading}> Sign up </Button>
        </Form>
        <div> Have an account already?{" "}
<Link href="/auth/login"> Login now </Link></div> </div></div>);}
```

Лістинг А.4 – Файл profile/page.tsx

```
export default function Profile() {
```

```
const {register, handleSubmit, formState: { errors, isValid },
control, trigger, reset, } = useForm<ProfileFormData>({ resolver:
zodResolver(profileSchema), mode: "onBlur",
    defaultValues: {firstName: "", lastName: "", dateOfBirth:
undefined, country: "", city: "", gender: "MALE", bio: "",interests:
[],profilePhoto: undefined, },});
const router = useRouter();
const fetchProfile = async () => {
const userAuthData = JSON.parse(localStorage.getItem("auth_data") ||
"{}");
    try {const response = await api.get(API_ROUTES.GET_PROFILE,
{headers: {"Content-Type": "application/json",Authorization: `Bearer
${userAuthData.tokens.accessToken}`,},})
     .json<{status: string; message: string | null; data: any; }>();
      if (response.status === "success" && response.data != null) {
        const transformedData = { ...response.data };
        const countryData = Country.getAllCountries().find(
          (c) => c.name === transformedData.country);
    if (countryData) {transformedData.country = countryData.isoCode;
        } else {transformedData.country = ""; }
        reset(transformedData); }
    } catch (error) {console.error("Network error:", error); }};
  const onSubmit = async (data: ProfileFormData) => { const
userAuthData = JSON.parse(localStorage.getItem("auth_data")|| "{}");
    try {const response = await api.put(API_ROUTES.UPDATE_PROFILE, {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            Authorization: `Bearer
${userAuthData.tokens.accessToken}`,},
          body: JSON.stringify(data), })
     .json<{status: string; message: string | null; data: any; }>();
      if (response.status === "success" && response.data != null) {
        router.push("/");}
    } catch (error) {console.error("Network error:", error); }};
  useEffect(() => {fetchProfile()}, [reset])
  return (
    <div >
      <Card className="max-w-6xl mx-auto shadow-lg p-8">
        <h6 >Your profile</h6>
        <p >Your profile is your first impression. Make it count
with photos and interests that tell your story</p>
        <CardBody>
          <form onSubmit={handleSubmit(onSubmit)}>
              <ProfileImage control={control} />
              <ProfileForm register={register} errors={errors}
trigger={trigger} control={control}/>
            <ProfileBio register={register} errors={errors}
trigger={trigger} control={control}/>
            <div className="w-full border-t border-coral-red " />
            <Controller name="interests" control={control}
render={({ field }) => (
```

```
          <InterestsSection value={field.value} onChange={(value) =>
{field.onChange(value); trigger("interests");}}errors={errors}/>)}/>
             <div className="flex justify-end mt-6">
    <Button type="submit" isDisabled={!isValid}>Save Profile</Button>
             </div> </form> </CardBody> </Card></div>);}
```

Лістинг А.5 – Файл ProfileForm.tsx

```
export default function ProfileForm({register, errors, trigger,
control,}: ProfileFormProps)
  const selectedCountry = useWatch({ control, name: "country" });
  const countryOptions = Country.getAllCountries().map((c) => ({
    value: c.isoCode, label: c.name, }));
  const rawCities = selectedCountry
    ? City.getCitiesOfCountry(selectedCountry) ?? []: [];
  const cityOptions = rawCities.map((c) => ({id: c.latitude, value:
c.name, label: c.name, }));
  return (
        <RHFController name="firstName" control={control}render={({
field }) => (
            <Input {...field} onBlur={() => trigger("firstName")}
              label="First name" />)}/>
        {errors.firstName && (
          <p >{errors.firstName.message}</p>)}
        <RHFController name="lastName"control={control}render={({
field }) => (<Input {...field} onBlur={() => trigger("lastName")}
              label="Last name" />)}/>
        {errors.lastName && (
          <p >{errors.lastName.message}</p>)}
        <RHFController name="dateOfBirth" control={control}
render={({ field }) => (
            <DatePicker label="Date of Birth" onChange={(value) =>
field.onChange(value ? new Date(value.toString()) : null) }
              onBlur={() => trigger("dateOfBirth")}/>)}/>
        {errors.dateOfBirth && (
          <p >{errors.dateOfBirth.message}</p>)}
        <RHFController name="country" control={control}
defaultValue="" render={({ field }) => (
            <Select label="Country"
selectedKeys={field.value ? new Set([field.value]) : new Set()}
onSelectionChange={(keys) => {const val = Array.from(keys)[0] as
string; field.onChange(val);
               trigger("country");}}
             onBlur={() => trigger("country")}>
             {countryOptions.map((option) => (<SelectItem
key={option.value} textValue={option.label}> {option.label}
               </SelectItem> ))} </Select>)}/>
        {errors.country && (<p >{errors.country.message}</p>)}
        <RHFController name="gender" control={control}
defaultValue="MALE" render={({ field }) => (
            <RadioGroup value={field.value}
 onChange={(value) => { field.onChange(value); trigger("gender");}}>
           <Radio value="MALE"> Male </Radio>
```

```
              <Radio value="FEMALE" >Female </Radio>
            </RadioGroup>)}/>
        {errors.gender && (
          <p>{errors.gender.message}</p>)});}
```

Лістинг А.6 – Файл home/page.tsx

```
export default function Home() {
  const [error, setError] = useState<string | null>(null);
  const [isLoading, setIsLoading] = useState(false);
  const [users, setUsers] = useState<any[]>([]);
  const { searchQuery, setSearchQuery, selectedInterests,
handleInterestClick, ageRange, setAgeRange, filteredCards,
  } = useCardFilter(users);
  const [currentPage, setCurrentPage] = useState<number>(1);
  const itemsPerPage = 12;
  const startIndex = (currentPage - 1) * itemsPerPage;
  const currentCards = filteredCards.slice(startIndex,
    startIndex + itemsPerPage);
  const handleAgeChange = (min: number, max: number) => {
    setAgeRange({ minAge: min, maxAge: max });};
  const fetchFeed = async () => {const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
    setError(null); setIsLoading(true);
    try {
      const response = await api.post(API_ROUTES.GET_FEED, {
          headers: {"Content-Type": "application/json",
     Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},})
        .json<{ status: string; message: string | null; data: {
user: any }[];}>();
      if (response.status === "success" && response.data != null) {
        setUsers(response.data); }
    } catch (error: any) {const errorMessage =
error.response?.data?.error || "Помилка при вході";
      setError(errorMessage); console.error("Error:", error);
    } finally {setIsLoading(false); }};
  useEffect(() => {fetchFeed();}, []);
  return (
<h1>Find Your People: Connect Others Who Share Your Interests</h1>
<p >Because life is more interesting when you're surrounded by like-
minded individuals. Start exploring now and find your tribe! </p>
     <Card className="max-w-7xl mx-auto shadow-lg p-8">
        <SearchBar searchQuery={searchQuery}
setSearchQuery={setSearchQuery} />
        <AgeSlider minAge={ageRange.minAge} maxAge={ageRange.maxAge}
          onAgeChange={handleAgeChange}/>
        <InterestFilter interests={ALL_INTERESTS} selectedInterests
={selectedInterests} handleInterestClick={handleInterestClick}/>
        <div> {users.map((user: IFeedCard) => (<FeedCard
key={user.id} user={user} /> ))} </div>
        <CardPagination totalItems={filteredCards.length}
currentPage={currentPage} onPageChange={setCurrentPage}/>
      </Card></div>);}
```

Лістинг А.7 – Файл FeedCard.tsx

```
export default function FeedCard({ user }: { user: IFeedCard }) {
  const json = encodeURIComponent(JSON.stringify(user));
  const displayedInterests = user.profile.interests.slice(0, 3);
  const hasMoreInterests = user.profile.interests.length > 2;
  const age = calculateAge(user.profile.dateOfBirth);
  return (
<Link href={{ pathname:`/users/${user.id}`,query: { data: json },}}>
      <Card className="bg-gray-100 border-2 border-black-pink ">
        <Image
          src={user.profile.profilePhoto}
          alt={`${user.profile.firstName}'s photo`}/>
        <div className="absolute bottom-4 left-4 right-4 ">
          <p > {user.profile.firstName} <span>{age}</span></p>
          <p>{user.profile.country}, {user.profile.city}</p>
          <div> {displayedInterests.map((interest) => (
              <span key={interest}>{interest} </span>))}
            {hasMoreInterests && (
              <span> ... </span>)}</div> </Card> </Link>);}
```

Лістинг А.8 – Файл UserProfile.tsx

```
export default function UserProfile({ user }: { user: IFeedCard }) {
  const searchParams = useSearchParams();
  const initialTab = searchParams.get("tab") || "profile";
  const [tab, setTab] = useState(initialTab);
  useEffect(() => { setTab(initialTab);}, [initialTab]);
  return (
    <div className="min-h-screen bg-gradient-to-r ">
      <div className="grid grid-cols-1 md:grid-cols-2 gap-6 ">
        <div className="bg-white rounded-lg shadow-lg p-6 flex">
          <div className="relative w-full h-[400px] mb-4">
            <Image src={user?.profile?.profilePhoto}
              alt={`${user?.profile?.firstName}'s photo`} /> </div>
          <h1 className="text-2xl font-bold text-gray-900">
            {user?.profile?.firstName}{" "} <span className="font-
normal > {calculateAge(user?.profile?.dateOfBirth)} </span> </h1>
          <p className="text-sm text-gray-500 mt-1">
            {user?.profile?.country}, {user?.profile?.city}</p>
          <p >Don't miss out! Chat or add to friends and start a new
adventure. </p>
          <div className="flex justify-center gap-6 mt-4 p-4">
            <button onClick={() => setTab("profile")}
              className={`flex items-center justify-center w-15 h-15
rounded-full shadow-md ${ tab === "profile" ? "bg-coral-red" : "bg-
white"} hover:bg-gray-300`} aria-label="Profile">
              <FaUser className={`text-2xl ${tab === "profile" ?
"text-white" : "text-gray-800"}`}/> </button>
            <button onClick={() => setTab("chat")}
```

```
              className={`flex items-center justify-center w-15 h-15
rounded-full shadow-md ${ tab === "chat" ? "bg-coral-red" : "bg-
white"} hover:bg-gray-300`}aria-label="Chat">
              <BsChatDotsFill className={`text-2xl ${ tab === "chat"
? "text-white" : "text-gray-800"}`}/>
            </button>
            <button className="flex items-center justify-center w-15
h-15 rounded-full " aria-label="Add to friends">
              <FaUserPlus className="text-2xl text-gray-800" />
            </button> </div> </div>
        {tab === "chat" ? (
          <div className="bg-white rounded-lg shadow-lg]">
 <ChatComponent profileName={user?.profile?.firstName} /> </div>) :
(<div className="flex flex-col gap-6">
            <div className="bg-white rounded-lg shadow-lg p-6 ">
              <p >Bio</p>
              <p >Curious about me? Here's a little glimpse into my
world</p>
              <div className="w-full border-t border-coral-red"/>
              <p>{user?.profile?.bio}</p></div>
            <div className="bg-white rounded-lg shadow-lg p-6 w>
              <h2 >Interests</h2>
              <p> We might share some passions - check out my
interests! </p>
              <div className="w-full border-t border-coral-red " />
              <div className="mt-2 flex flex-wrap gap-2">
                {user?.profile?.interests.map((interest: any) => (
                  <span key={interest}>{interest}</span>))}</div>);}
```

Лістинг А.9 – Файл friend/page.tsx

```
export default function FriendsPage() {
  const [activeTab, setActiveTab] = useState("friends");
  const [searchQuery, setSearchQuery] = useState("");

  const [friends, setFriends] = useState<any[]>([]);
  const [receivedRequests, setReceivedRequests] =
useState<any[]>([]);
  const fetchFriends = async () => { const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
    try { const response = await api .get(API_ROUTES.GET_FRIENDS, {
          headers: { "Content-Type": "application/json",
     Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},})
   .json<{ status: string; message: string | null; data: any[];}>();
      if (response.status === "success" && response.data != null) {
        setFriends(response.data);
      } else { console.error("Error getting friends");}
    } catch (error) {console.error("Network error:", error);}};
  const fetchReceivedRequests = async () => {
    const userAuthData =
JSON.parse(localStorage.getItem("auth_data") || "{}");
    try {const response = await api
        .get(API_ROUTES.GET_RECEIVED_FRIENDSHIP_REQUESTS, {
```

```
          headers: {"Content-Type": "application/json",
  Authorization: `Bearer ${userAuthData.tokens.accessToken}`,},})
    .json<{ status: string; message: string | null; data: any[];}>();
       if (response.status === "success" && response.data != null) {
         setReceivedRequests(response.data);
       } else {console.error("Error getting received friend
requests");}
     } catch (error) {console.error("Network error:", error);}};
const currentList=activeTab==="friends"? friends : receivedRequests;
  useEffect(() => {fetchFriends()fetchReceivedRequests()}, []);
  return (
      <div className="min-h-screen bg-gradient-to-b ">
        <Card className="max-w-7xl mx-auto shadow-lg p-8">
          <div className="flex gap-10 mb-5 justify-center">
            <button className={`text-2xl font-semibold pb-2 ${
             activeTab === "friends" ? "text-coral-red border-b-2 "
             : "text-gray-400 hover:text-coral-red"}`}
              onClick={() => setActiveTab("friends")}>
              Your Friends </button>
            <button className={`text-2xl font-semibold pb-2 ${
             activeTab === "requests" ? "text-coral-red border-b-2 "
             : "text-gray-400 hover:text-coral-red"}`}
              onClick={() => setActiveTab("requests")}>
              Friend Requests</button></div>
<SearchBar searchQuery={searchQuery} setSearhQuery={setSearhQuery}/>
          <div className="flex flex-wrap gap-8 px-6 md:px-12">
            {activeTab === "friends" ? friends.map((friend:
IFriendCard) => (<FriendsCard key={friend.id} friend={friend} />))
             : receivedRequests.map((request: IFriendCard) => (
              <RequestCard key={request.id} request={request} />))}
        </Card>
</div> </div>);}
```

Лістинг А.10 – Файл FriendsCard.tsx

```
export default function FriendsCard({ friend }: { friend:
IFriendCard }) {
  const age = calculateAge(friend?.profile?.dateOfBirth ?? 0);
  const router = useRouter();
  const [isPopoverOpen, setIsPopoverOpen] = useState(false);
  const handleChatClick = () => {
    router.push(`/users/${friend.id}?tab=chat`);};
  return (
      <Card>
   <CardBody className="p-0" as={Link} href={`/users/${friend.id}`}>
          <Image src={friend?.profile?.profilePhoto}
            alt={`${friend?.profile?.firstName}'s photo`}/>
        </CardBody>
   <Dropdown isOpen={isPopoverOpen} onOpenChange={setIsPopoverOpen}>
          <DropdownTrigger>
            <Button aria-label="More options"><CgMore/></Button>
          </DropdownTrigger>
          <DropdownMenu aria-label="Static Actions">
```

```
            <DropdownItem key="delete"
              startContent={<MdDeleteForever/>
              Delete friend </DropdownItem>
            <DropdownItem key="block" startContent={</ImBlocked>}>
              Block friend</DropdownItem>
          </DropdownMenu></Dropdown>
          <p>{friend?.profile?.firstName}{" "}
            <span className="font-normal">{age}</span></p>
          {friend?.profile?.country ? (
            <p className="uppercase text-sm mt-
1">{friend?.profile?.country}, {friend?.profile?.city}</p>
          ) : <p>Unknown</p>}
          <Button onPress={handleChatClick}aria-label="Chat">
            <BsChatDotsFill className="text-white text-2xl" />
          </Button> </div> </Card> </div>);}
```

Лістинг А.11 – Код контролера AuthController

```
@ApiTags("auth")
@Controller("auth")
export class AuthController {
  constructor(private readonly authService: AuthService) {}
  @ApiBody({ type: UserLogInDto })
  @Public()
  @Post("/login")
  @HttpCode(HttpStatus.OK)
  async login(@Body() userLogInDto: UserLogInDto):
Promise<LoginResponse> {
    return await this.authService.login(userLogInDto);}
  @Public()
  @UseGuards(DoesUserExistGuard)
  @Post("/sign-up")
  @HttpCode(HttpStatus.OK)
  async signUp(@Body() userSignUpDto: UserSignUpDto):
Promise<SignUpResponse> {
    return await this.authService.signUp(userSignUpDto);}
  @UseGuards(JwtAuthGuard)
  @Get("/user")
  @HttpCode(HttpStatus.OK)
  async getUser(@GetCurrentUserId() userId: string):
Promise<UserCore> {
    return await this.authService.getUser(userId); }
  @Post("/logout")
  @HttpCode(HttpStatus.OK)
  async logout(@GetCurrentUserId() userId: string): Promise<string>
{await this.authService.logout(userId);
    return "Logged out successfully";}
  @Public()
  @UseGuards(RefreshAuthGuard)
  @Post("/refresh")
  @HttpCode(HttpStatus.OK)
  async refreshTokens( @GetCurrentUser() user: UserCore,
    @GetCurrentUserProperty("refreshToken") refreshToken: string,
```

```
  ): Promise<Tokens> { const jwtPayload: JwtPayload = { sub: user.id,
email: user.email, };
   return await this.authService.refreshTokens(jwtPayload,
refreshToken);}}
```

Лістинг А.12 – Код контролера FileController

```
@ApiTags("Files")
@ApiBearerAuth()
@Controller("file")
export class FileController {
  constructor(
    private readonly fileService: FileService,
    private readonly logger: WinstonService,) {}
  @Post()
  @HttpCode(HttpStatus.CREATED)
  @UseInterceptors(FileInterceptor("photo"))
  async uploadPhoto(@UploadedFile(new ParseFilePipeBuilder()
        .addValidator(new UploadFileTypeValidator({
            fileType: VALID_UPLOADS_MIME_TYPES, }),)
        .addMaxSizeValidator({maxSize: MAX_PICTURE_SIZE_IN_BYTES,
          message: `Photo should be less than
${MAX_PICTURE_SIZE_IN_BYTES} bytes`,})
  .build({ errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY }),)
    file: Express.Multer.File, @Request() req: any,
    @Response() res: any,) {
    try {const userId = req?.user?.id;
      if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
   const response = await this.fileService.uploadFile(file, userId);
  return res.status(HttpStatus.CREATED).json({type:
ResponseType.SUCCESS, message: null, data: response, });
    } catch (error) {
      this.logger.error("Error occurred while uploading photo", {
context: "FileController:uploadPhoto", error: error,  message:
error.message || "Unknown error", stack: error.stack,});}}
  @Get(":filename")
  @ApiParam({ name: "filename", description: "The name of the file to
retrieve", type: String, })
  async getPhoto(
    @Param("filename") filename: string,
    @Request() req: any,
    @Response({ passthrough: true }) res: any,) {
    try { const userId = req?.user?.id;
      if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
  const response = await this.fileService.getFile(userId, filename);
return res.status(HttpStatus.OK).json({ type: ResponseType.SUCCESS,
message: null, data: response.stream, });
    } catch (error) {
this.logger.error("Error occurred while getting photo", { context:
"FileController:getPhoto", error: error, message: error.message ||
"Unknown error", stack: error.stack, });}}
```

```
  @Get()
  @HttpCode(HttpStatus.OK)
  async getPhotos(
    @Request() req: any,
    @Response() res: any,
  ): Promise<FileDetails[]> {try { const userId = req?.user?.id;
      if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
      const response: FileDetails[] = await
this.fileService.getFiles(userId);
      return res.status(HttpStatus.OK).json({type:
ResponseType.SUCCESS, message: null, data: response, });
    } catch (error) {
      this.logger.error("Error occurred while getting a list of
photos", {context: "FileController:getPhotos",error: error, message:
error.message || "Unknown error",stack: error.stack, });}}
  @Delete(":filename")
  @HttpCode(HttpStatus.OK)
  @ApiParam({name: "filename",description: "The name of the file to
delete",type: String, })
  async deletePhoto(
    @Param("filename") filename: string,
    @Request() req: any,
    @Response() res: any,) {
    try {const userId = req?.user?.id;
      if (!userId) {
        throw new UnauthorizedException("User ID is not found");}
      const response = await this.fileService.deleteFile(userId,
filename);
      return res.status(HttpStatus.CREATED).json({type:
ResponseType.SUCCESS, message: null, data: response, });
    } catch (error) {
this.logger.error("Error occurred while deleting photo", {context:
"FileController:deletePhoto",error: error, message: error.message ||
"Unknown error",stack: error.stack, });}}}
```

Лістинг А.13 – Код контролера FiltersController

```
@ApiTags("filters")
@Controller("filters")
@UseGuards(JwtAuthGuard)
export class FiltersController {
  constructor(private readonly filtersService: FiltersService,
    private readonly logger: WinstonService,) {}
  @ApiBearerAuth()
  @HttpCode(HttpStatus.OK)
  @Get("/get-filters")
  async     getFilters(@GetCurrentUserId()     userId:     string):
Promise<UserFilters> {try {
      const response = await this.filtersService.getFilters(userId);
      return response;
    } catch (error) {
```

```
this.logger.error("Error occurred while retrieving filters",
{context: "FiltersController:getFilters", error: error, message:
error.message || "Unknown error",stack: error.stack, });
      throw new HttpException({type: ResponseType.ERROR, message:
error.message || "Error occurred while retrieving filters",data:
null, },HttpStatus.INTERNAL_SERVER_ERROR,); }}
  @ApiBearerAuth()
  @HttpCode(HttpStatus.OK)
  @Put("/update-filters")
  async updateFilters(
    @Body() updateFilterDto: UpdateFiltersDto,
    @GetCurrentUserId() userId: string,): Promise<UserFilters> {
    try { const response = await this.filtersService.updateFilters
(updateFilterDto, userId); return response;
    } catch (error) {
      this.logger.error("Error occurred while updating filters",
{context: "FiltersController:updateFilters",error: error, message:
error.message || "Unknown error",stack: error.stack, });
      throw new HttpException({type: ResponseType.ERROR, message:
error.message||"Erroroccurred while updating filters",data: null, },
       HttpStatus.INTERNAL_SERVER_ERROR,); }}}
```

Лістинг А.14 – Код контролера FriendsController

```
@ApiTags("friends")
@Controller("friends")
@UseGuards(JwtAuthGuard)
export class FriendsController {
constructor( private readonly friendsService: FriendsService,
private readonly friendshipRequestService: FriendshipRequestService,
private readonly logger: WinstonService,) {}
@ApiBearerAuth()
@Get()
@HttpCode(HttpStatus.OK)
async getFriends(
@GetCurrentUserId() userId: string,
): Promise<UserCore[]> {
try {const response = await this.friendsService.getFriends(userId);
return response;} catch (error) {
this.logger.error("Error occurred while retrieving friends", {
context: "FriendsController:getFriends",error: error, message:
error.message || "Unknown error", stack: error.stack,});
throw new HttpException({type: ResponseType.ERROR, message:
error.message || "Error occurred while retrieving friends",data:
null,},HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Get("/sent-requests")
@HttpCode(HttpStatus.OK)
async getSentFriendshipRequests(@GetCurrentUserId() userId: string,
): Promise<UserCore[]> { try {const response = await
this.friendshipRequestService.getSentFriendshipRequests(userId);
return response;} catch (error) {
```

```
this.logger.error("Error occurred while retrieving sent friendship
requests", {context: "FriendsController:getSentFriendshipRequests",
error: error, message: error.message || "Unknown error",stack:
error.stack,}); throw new HttpException({type: ResponseType.ERROR,
message: error.message || "Error occurred while retrieving sent
friendship requests",data: null,},
HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Get("/received-requests")
@HttpCode(HttpStatus.OK)
async getReceivedFriendshipRequests(
@GetCurrentUserId() userId: string,): Promise<UserCore[]> { try {
const response = await
this.friendshipRequestService.getReceivedFriendshipRequests(userId);
return response;} catch (error) {
this.logger.error("Error occurred while retrieving received
friendship requests", {context:
"FriendsController:getReceivedFriendshipRequests",error: error,
message: error.message || "Unknown error", stack: error.stack,});
throw new HttpException({ type: ResponseType.ERROR, message:
error.message || "Error occurred while retrieving received
friendship requests", data: null,},
HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Post("/request-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
@ApiParam({ name: "recipientId",description: "The ID of the user to
send a friendship request to",type: String,})
async requestFriendship(@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string,
): Promise<RequestFriendshipResponse> { try { const response = await
this.friendshipRequestService.requestFriendship(userId,
recipientId);
return response;} catch (error) {this.logger.error("Error occurred
while sending friendship request", {context:
"FriendsController:requestFriendship", error: error, message:
error.message || "Unknown error", stack: error.stack,});
throw new HttpException({ type: ResponseType.ERROR, message:
error.message || "Error occurred while sending friendship request",
data: null,},HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Post("/accept-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
@ApiParam({ name: "recipientId", description: "The ID of the user
whose friendship request to accept", type: String, })
async acceptFriendship(
@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string, ):
Promise<AcceptFriendshipResponse> {try {
const response = await
this.friendshipRequestService.acceptFriendshipRequest(userId,
recipientId); return response;
} catch (error) {
```

```
this.logger.error("Error occurred while accepting friendship
request", {context: "FriendsController:acceptFriendship", error:
error, message: error.message || "Unknown error", stack:
error.stack,});
throw new HttpException({type: ResponseType.ERROR,
message: error.message || "Error occurred while accepting friendship
request", data: null,},HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Post("/reject-friendship/:recipientId")
@HttpCode(HttpStatus.OK)
@ApiParam({ name: "recipientId", description: "The ID of the user
whose friendship request to reject", type: String,})
async rejectFriendship(
@GetCurrentUserId() userId: string,
@Param("recipientId") recipientId: string,
): Promise<AcceptFriendshipResponse> {try { const response = await
this.friendshipRequestService.rejectFriendshipRequest(userId,
recipientId);
return response;} catch (error) {
this.logger.error("Error occurred while rejecting friendship
request", {context: "FriendsController:rejectFriendship",
error: error,message: error.message || "Unknown error",stack:
error.stack,});
throw new HttpException({type: ResponseType.ERROR,message:
error.message || "Error occurred while rejecting friendship
request",data: null,},HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Delete("/:friendId")
@HttpCode(HttpStatus.OK)
@ApiParam({name: "friendId", description: "The ID of the friend to
delete", type: String,})
async deleteFriend(
@GetCurrentUserId() userId: string,
@Param("friendId") friendId: string,
): Promise<DeleteFriendResponse> { try {
const response = await this.friendsService.deleteFriend(userId,
friendId);
return response; } catch (error) {
this.logger.error("Error occurred while deleting friend", {context:
"FriendsController:deleteFriend", error: error, message:
error.message || "Unknown error", stack: error.stack,});
throw new HttpException({type: ResponseType.ERROR,
message: error.message || "Error occurred while deleting friend",
data: null,}, HttpStatus.INTERNAL_SERVER_ERROR,);}}
@ApiBearerAuth()
@Delete()
@HttpCode(HttpStatus.OK)
async deleteFriends(
@GetCurrentUserId() userId: string,
): Promise<DeleteFriendResponse> {try {
const response = await this.friendsService.deleteFriends(userId);
return response;} catch (error) {
```

```
this.logger.error("Error occurred while deleting all friends",
{context: "FriendsController:deleteFriends",error: error, message:
error.message || "Unknown error", stack: error.stack,});
throw new HttpException({ type: ResponseType.ERROR, message:
error.message || "Error occurred while deleting all friends", data:
null,},HttpStatus.INTERNAL_SERVER_ERROR,);}}   }
```

MateUp

РОЗРОБКА ВЕБСИСТЕМИ ДЛЯ ПОШУКУ ДРУЗІВ ЗА ІНТЕРЕСАМИ

Національний університет "Запорізька політехніка"

Кафедра комп'ютерних систем та мереж

Виконавець: Студентка групи КНТ-521 Коваль Марина Сергіївна

Керівник: Скрупський Степан Юрійович

Рік захисту: 2025

MateUp

Цілі та Завдання

1

Розробка вебсистеми

Створити зручну вебсистему для пошуку друзів за спільними інтересами.

2

Пошук по вподобаннях

Забезпечити можливість знаходження нових контактів на основі вподобань, віку та інших параметрів.

3

Покращення соціальної взаємодії

Сприяти розширенню соціального кола користувачів у цифровому середовищі.

4

Підвищення зручності користування

Розробити інтуїтивний інтерфейс для комфортної роботи із системою.

MateUp

Актуальність Розробки

Криза Справжніх Знайомств

Сучасна молодь все частіше скаржиться на складність пошуку однодумців у реальному житті. Дослідження показують, що 50% людей віком 18-35 років відчувають труднощі у знаходженні друзів за інтересами. Традиційні способи знайомств через роботу чи навчання часто обмежені географічно та соціально, залишаючи багатьох без можливості розширити коло спілкування.

Алгоритмічна ізоляція

Існуючі додатки зосереджені на романтичних стосунках або професійному нетворкінгу, але ігнорують потребу в дружніх відносинах на основі хобі та інтересів. Люди змушені використовувати неспеціалізовані платформи, що ускладнює пошук справжніх однодумців та призводить до розчарувань у процесі знайомств.

Перевантаження Соціальних Мереж

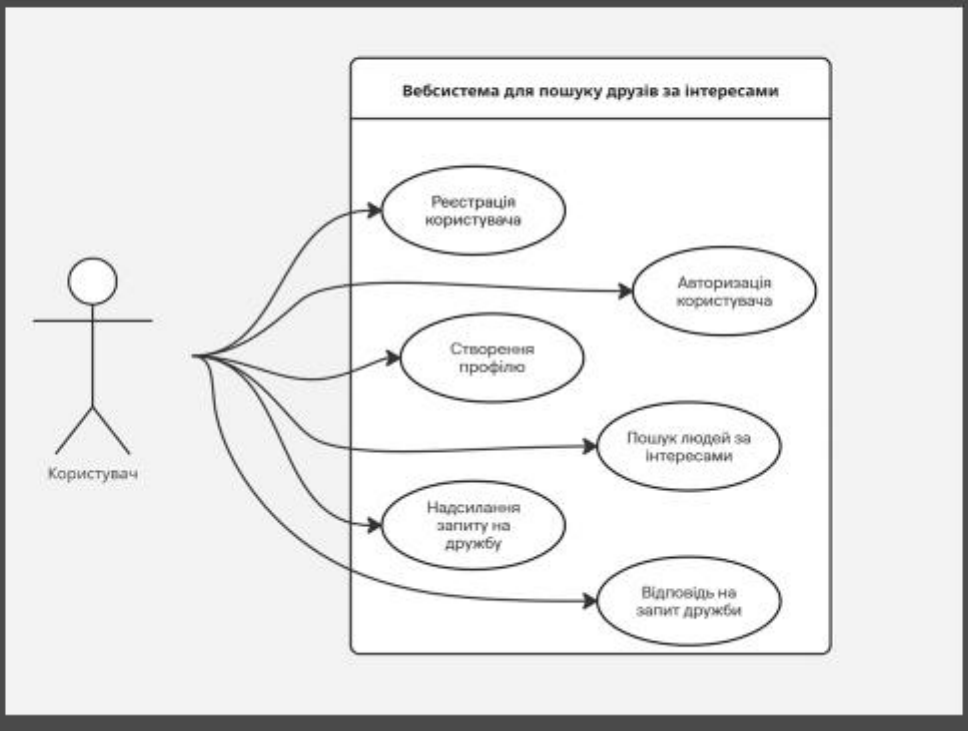
Популярні платформи створили парадокс: маючи сотні "друзів" онлайн, люди почуваються все більш ізольованими. Поверхневе спілкування, нав'язана реклама та алгоритми, які показують лише схожий контент, створюють "ехо-камери". Це призводить до зменшення якості соціальних зв'язків та утруднює пошук людей з реальними спільними захопленнями.

Необхідність Інноваційного Підходу

Час створити платформу, яка об'єднує людей не за зовнішністю чи статусом, а за справжніми захопленнями. Інтуїтивний інтерфейс, детальні профілі інтересів та система пошуку за категоріями хобі можуть створити значуще середовище для формування дружніх стосунків, базованих на взаємних захопленнях та спільних активностях.

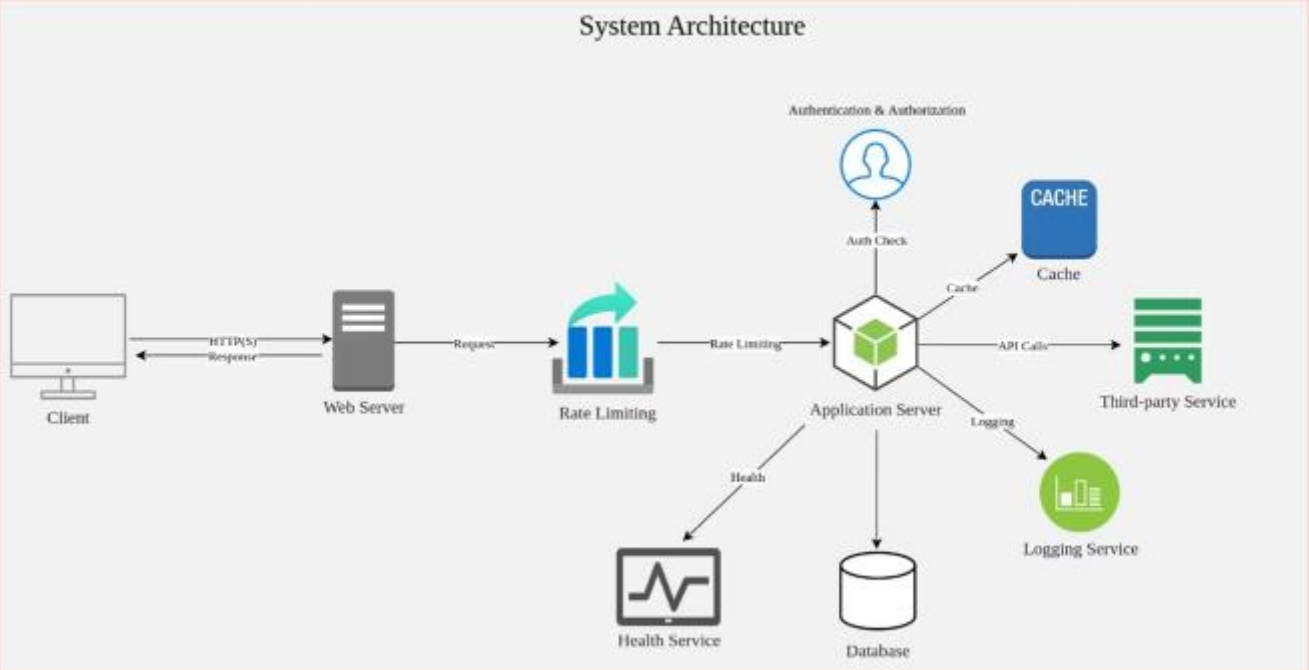
MateUp

Діаграма варіантів використання (Use Case)



MateUp

Архітектура Системи



MateUp

Мережева Архітектура Додатку

Архітектура розробки (Development)

HTTP протокол для швидкого прототипування На етапі розробки використовується HTTP-протокол через REST API-запити для простоти налагодження та тестування функцій знайомств. Локальне середовище дозволяє швидко перевіряти:

- Створення профілів користувачів
- Пошук за інтересами
- Відображення списків потенційних друзів

Продакшн архітектура (Production)

У реальному середовищі критично важливо використовувати HTTPS (HTTP + TLS шифрування) для захисту чутливої інформації користувачів:

Безпека особистих даних

- Профілі користувачів - захист імен, фото, контактної інформації
- Авторизація - захист паролів та токенів доступу

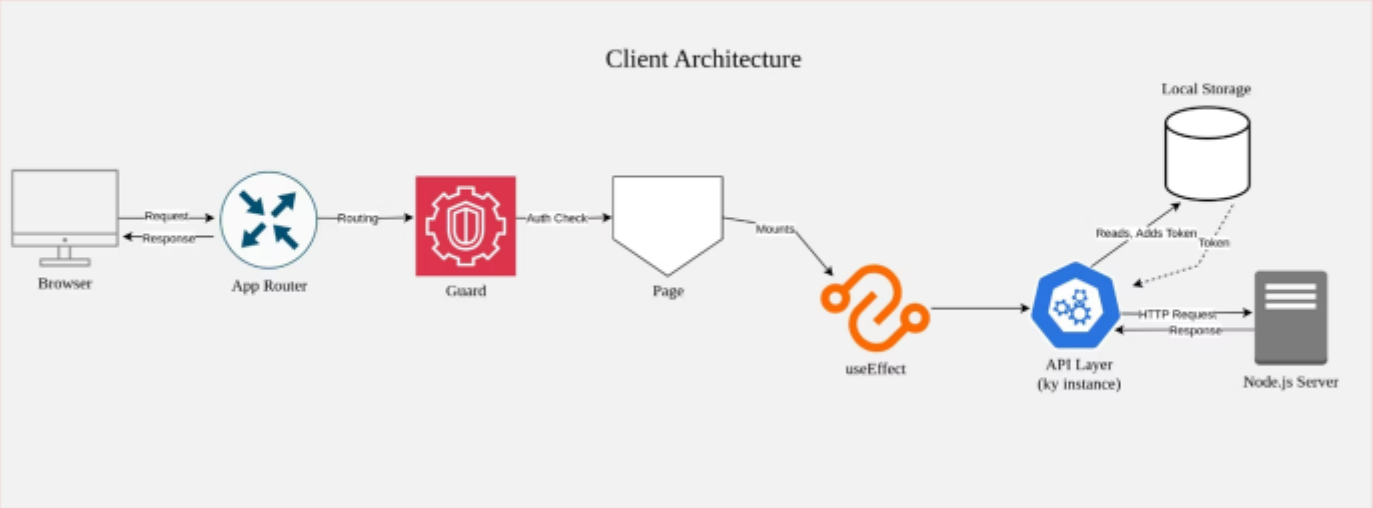
Довіра користувачів

Користувачі довіряють додатку свої особисті дані для знайомств. HTTPS гарантує, що інформація не може бути перехоплена під час передачі, що критично важливо для:

- Збереження приватності профілів
- Захисту від підробки акаунтів
- Запобігання витоку особистої інформації

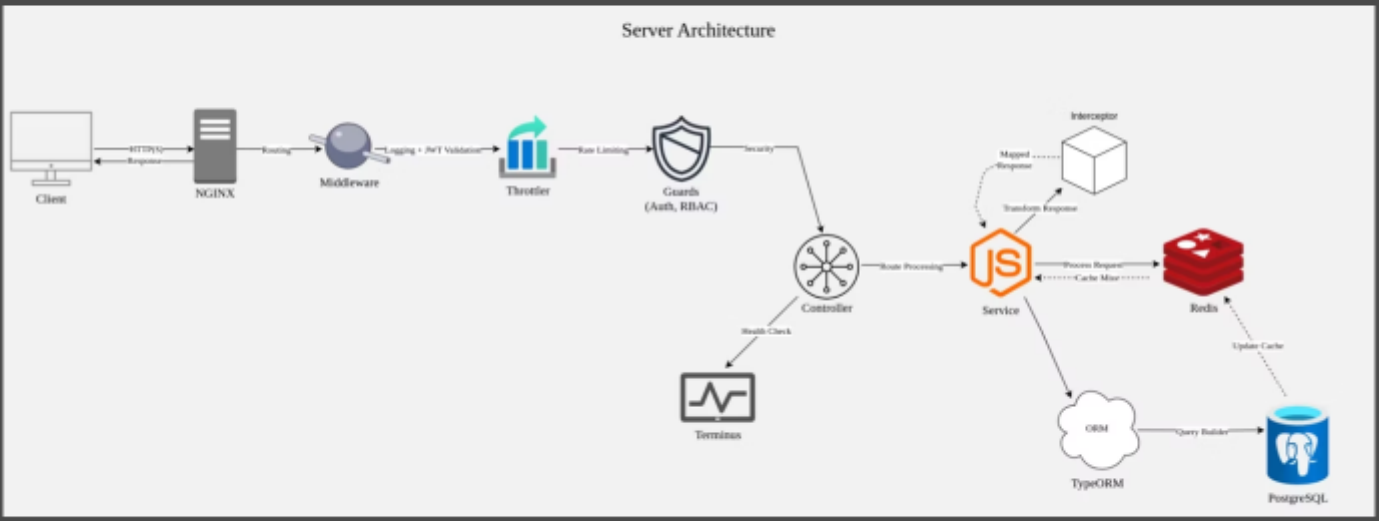
MateUp

Клієнтська Архітектура



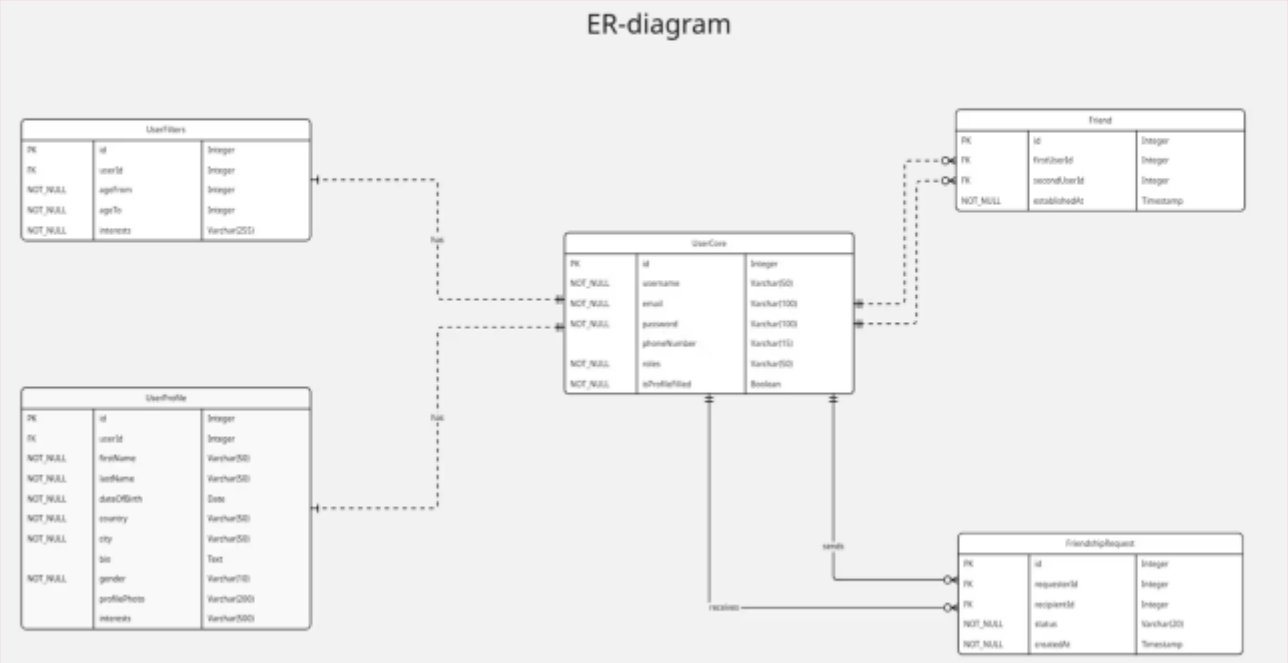
MateUp

Серверна Архітектура (частина перша)



MateUp

Серверна Архітектура (частина друга)



MateUp

Ключові Модулі

1

Автентифікація

Надійна система реєстрації та входу з використанням JWT-токенів. Підтримка входу через email/пароль з верифікацією електронної пошти для захисту від фейкових акаунтів.

2

Профілі Користувачів

Детальні профілі з можливістю додавання фото, опису інтересів, хобі та життєвих цілей. Налаштування приватності дозволяє контролювати, яку інформацію бачать інші користувачі.

3

Фільтрація Користувачів

Система фільтрів за інтересами та віком. Користувачі можуть налаштовувати персоналізовані критерії пошуку для знаходження найбільш підходящих кандидатів у друзі.

4

Система друзів

Можливість надсилати запити на дружбу, приймати або відхиляти запрошення. Ведення списку друзів з різними рівнями доступу до особистої інформації та можливістю блокування небажаних користувачів.

5

Пошук за Інтересами

Система категорій та тегів дозволяє швидко знаходити людей зі спільними захопленнями: спорт, творчість, технології, подорожі, книги, музика та багато іншого.

Результати Розробки (частина перша)

Екран реєстрації користувача для швидкого доступу до застосунку.

Інтерфейс профілю користувача з інформацією та налаштуваннями

Результати Розробки (частина друга)

Стрічка - список користувачів, де можна шукати нових друзів

Екран друзів з розділом списку друзів.

MoteUp

Системні Вимоги

Вимоги апаратної частини для запуску в браузері:

Компонент	Мінімальні характеристики	Оптимальні характеристики
Процесор (CPU)	2 ядра, 1.6 ГГц	4 ядра, 2.0 ГГц і вище
Оперативна пам'ять (RAM)	2 ГБ	4 ГБ і більше
Дисплей	Будь-який екран з роздільною здатністю від 1024×768	Екран з роздільною здатністю Full HD (1920×1080)
Веб-браузер	Сучасний браузер (Chrome, Firefox, Edge, Opera тощо)	Оновлений браузер останньої версії (Chrome, Firefox, Edge, Safari)
Доступ до Інтернету	Стабільне з'єднання зі швидкістю від 10 Мбіт/с	Надійне з'єднання від 25 Мбіт/с
Накопичувач	Не потребує встановлення (браузерна версія)	Не потребує встановлення (браузерна версія)

MoteUp

Висновки та Перспективи

Переваги рішення

Монолітна архітектура забезпечує швидку розробку та легке розгортання. Next.js та TypeScript гарантують високу продуктивність та швидкість UI.

Ефективність

Рішення демонструє високу ефективність у досягненні цілей і готове до подальшого масштабування.

Перспективи масштабування

Планується виділення мікросервісів для авторизації, пошуку та рекомендацій, а також інтеграція чатів і мультиплатформенність.

Подальший розвиток

Впровадження машинного навчання для покращення алгоритмів підбору друзів та збільшення залученості користувачів.