Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматики
Кафедра комп'ютерних наук

Бакалаврська кваліфікаційна робота на тему:

ПРОГРАМНИЙ МОДУЛЬ ДЛЯ ПЛАТФОРМИ ОЦІНЮВАННЯ ТА РЕЦЕНЗУВАННЯ ВІДЕОІГОР

Виконав: студент 4 курсу, групи 1КН-21б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Лікнарович В. В.
(прізвище та ініціали)
Керівник: к. т. н., доцент каф. КН
Арсенюк І. Р.
(прізвище та ініціали)
«04» червня 2025 р.
Рецензент: к. т. н., доц., доц. кафедри АІІТ
Барабан М. В.
(прізвище та ініціали)
«05» червня 2025 р.

Допущено до захисту
Зав. кафедри Яровий А. А.
« 06 » червня 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
д.т.н., проф. Яровий А. А.
«05» травня 2025 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
Лікнаровичу Владиславу Валерійовичу

1. Тема роботи: Програмний модуль для платформи оцінювання та рецензування відеоігор.
   Керівник роботи: Арсенюк Ігор Ростиславович, к. т. н., доц.
   Затверджені наказом вищого навчального закладу «20»03 2025 року № 97
2. Термін подання студентом роботи 30 травня 2025 р.
3. Вихідні дані до роботи:

кількість відеоігор у базі даних – не менше 100;
кількість сутностей у базі даних – 8;
мова програмування – об'єктно-орієнтована;
тип застосунку – веб-застосунок;
СУБД – реляційна.

4. Зміст текстової частини (перелік питань, які потрібно розробити): аналіз предметної області та постановка задачі; проєктування програмного модуля оцінювання та рецензування відеоігор; програмна реалізація програмного модуля оцінювання та рецензування відеоігор; тестування програмного модуля оцінювання та рецензування відеоігор
5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень): діаграма розгортання програмного модуля; діаграма прецедентів програмного модуля; ER-діаграма програмного модуля; схема бази даних програмного модуля; схема алгоритму роботи програмного модуля

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	виконання прийняв
1-3	Арсенюк І. Р., к. т. н., доцент каф. КН		

7. Дата видачі завдання 05 травня 2025 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз сучасних програмних модулів оцінювання та рецензування відеоігор	05.05.25	08.05.25	
2	Проектування програмного модуля оцінювання та рецензування відеоігор	09.05.25	12.05.25	
3	Програмна реалізація модуля оцінювання та рецензування відеоігор	13.05.25	29.05.25	
4	Розробка інструкції користувача	30.05.25	01.06.25	
5	Оформлення матеріалів до захисту БКР	02.06.25	04.06.25	

Студент _____ (підпис) Лікнарович В. В.

Керівник роботи _____ (підпис) Арсенюк І. Р.

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 96 сторінок формату А4, на яких є 31 рисунок, 1 таблиця, список використаних джерел містить 26 найменувань.

Бакалаврська кваліфікаційна робота присвячена розробці програмного модуля оцінювання та рецензування відеоігор, що дозволяє користувачам залишати оцінки, публікувати рецензії, брати участь у форумі, а також здійснювати пошук інших гравців за географічними критеріями. Розроблено діаграми розгортання, прецедентів та загальний алгоритм програмного модуля. Обґрунтовано використання мов програмування Go та JavaScript, середовища розробки Visual Studio Code та СУБД PostgreSQL. Програмний модуль успішно протестовано на коректність всіх заданих функцій.

Ключові слова: відеоігри, рецензування відеоігор, соціальна взаємодія, форум, оцінювання відеоігор.

ABSTRACT

The bachelor's thesis consists of 96 pages of A4 format, which includes 31 figures, 1 table, and a list of references containing 26 titles.

The bachelor's thesis is devoted to the development of a software module for evaluating and reviewing video games, which allows users to leave ratings, publish reviews, participate in a forum, and search for other players by geographic criteria. Deployment diagrams, precedents, and a general algorithm of the program module are developed. The use of the Go and JavaScript programming languages, the Visual Studio Code development environment, and the PostgreSQL database is substantiated. The program module has been successfully tested for the correctness of all the specified functions.

Keywords: video games, video game review, social interaction, forum, video game evaluation.

ЗМІСТ

ВСТУП

Актуальність досліджень. У сучасному світі відеоігри стали невіддільною частиною глобальної цифрової культури. Вони перестали бути лише засобом розваги та перетворилися на повноцінний медіапростір для творчого самовираження, спілкування та обміну досвідом між мільйонами користувачів по всьому світу. З розвитком інтернету зросла і потреба в онлайн-інструментах, які дозволяють гравцям не тільки грати, а й ділитися своїми враженнями, думками та рекомендаціями щодо пройдених ігор. У цьому контексті особливого значення набувають веб-платформи, що поєднують функції оцінювання, рецензування та соціальної взаємодії.

Багато гравців ведуть власні журнали проходження, створюють списки бажаних ігор, залишають рецензії та рейтинги, що допомагає їм структурувати власний ігровий досвід, а також ділитися ним з іншими користувачами. Проте наявні платформи часто обмежені в можливостях соціальної взаємодії або не забезпечують зручного та інтуїтивного інтерфейсу для обговорень, ведення записів і пошуку однодумців.

Запропонований у межах цієї роботи програмний модуль покликаний вирішити вказані проблеми, об’єднавши в єдиній системі такі функції, як створення особистого журналу проходження ігор, написання рецензій, оцінювання ігор, публікація постів на особистих сторінках користувачів та залишення коментарів до них. Додатково передбачено наявність тематичного форуму для обговорення ігор, запитань про ігрові механіки або поради, а також можливість знаходити інших користувачів за країною чи містом проживання. Такий підхід сприятиме формуванню активної спільноти навколо відеоігр, де кожен учасник зможе знайти цікаві ігри, отримати допомогу, залишити власну думку або просто поспілкуватися з іншими гравцями.

Метою дослідження є розширення функціональних можливостей програмного модуля для оцінювання та рецензування відеоігор. На відміну від

аналогів розроблений програмний модуль буде поєднувати функції ведення журналу пройдених ігор, публікації розгорнутих рецензій, форумного обговорення та соціальної взаємодії разом.

Об’єктом дослідження є процес оцінювання, рецензування та обговорення відеоігор.

Предмет дослідження є програмний модуль оцінювання, рецензування та обговорення відеоігор.

Задачі дослідження:

1. Аналіз предметної області оцінювання та рецензування відеоігор.
2. Огляд та порівняльний аналіз програмних засобів оцінювання та рецензування відеоігор.
3. Розробка UML-діаграм розгортання та прецедентів програмного модуля.
4. Розробка загального алгоритму роботи програмного модуля.
5. Програмна реалізація та тестування модуля оцінювання та рецензування відеоігор.
6. Розробка інструкції користувача.

Апробація роботи.

Робота апробувалась на конференції «Молодь в науці: дослідження, проблеми, перспективи 2025 р.», доповідь «Обґрунтування доцільності створення програмного модуля для оцінювання та рецензування відеоігор» [1].

Публікації: електронні тези конференції «Молодь у науці: дослідження, проблеми, перспективи, 2025 р.» [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ОЦІНЮВАННЯ ТА РЕЦЕНЗУВАННЯ ВІДЕОІГОР

1.1 Обґрунтування доцільності розробки програмного модуля для оцінювання та рецензування відеоігор

Розробка програмного модуля для оцінювання та рецензування відеоігор із розширеними соціальними функціями є доцільною з огляду на кілька ключових факторів.

Глобальний ринок відеоігор демонструє стабільне зростання протягом останніх років. За прогнозами аналітиків, до кінця 2025 року ринок досягне позначки в 200 мільярдів доларів [2]. Кількість активних гравців продовжує збільшуватись, охоплюючи різні вікові та соціальні категорії населення. Цей тренд створює потребу в спеціалізованих платформах для обміну думками та враженнями про ігровий досвід.

Аналіз існуючих платформ для оцінювання та рецензування відеоігор виявив суттєві обмеження в їхній функціональності. Більшість платформ фокусуються лише на базових функціях оцінювання та написання коротких відгуків, не надаючи повноцінних інструментів для ведення детального журналу ігрового прогресу. Також спостерігаються обмежені можливості соціальної взаємодії між користувачами – відсутність інтегрованих форумів для обговорення конкретних ігор або допомоги з проходженням. Наявні рішення часто не забезпечують особистих сторінок користувачів з можливістю публікації власного контенту та коментування. Практично відсутні функції пошуку користувачів за географічним принципом, що ускладнює формування локальних спільнот гравців.

Соціальний аспект геймінгу продовжує набирати вагу в сучасному світі. У культурному плані ігрові спільноти об'єднують понад три мільярди людей по всьому світу, долаючи вікові, гендерні та географічні кордони. Відеоігри стали

мейнстрімним культурним феноменом: близько 85% населення планети взаємодіють з іграми, граючи або переглядаючи ігровий контент. Індустрія все частіше надає пріоритет інклюзивному оповіданню та культурній репрезентації, що відображає цінності значної частини аудиторії. У соціальному плані ці спільноти забезпечують життєво важливий простір для взаємодії, співпраці та побудови дружби. Багато геймерів повідомляють, що ігри допомагають їм знайти нових друзів і пропонують переваги для психічного здоров'я, такі як зняття стресу та відчуття досягнення [3].

Розроблювана система має на меті заповнити цю прогалину шляхом створення єдиного середовища, що поєднуватиме функції ведення журналу пройдених ігор, публікації розгорнутих рецензій, форумного обговорення та соціальної взаємодії. Особливу увагу буде приділено можливості пошуку користувачів за географічним принципом, що дозволить формувати локальні спільноти гравців, організовувати офлайн-зустрічі та посилювати соціальні зв'язки на основі спільних інтересів.

З огляду на всі вищеперераховані фактори, розробка програмного модуля для оцінювання та рецензування відеоігор із розширеними соціальними функціями є перспективним та доцільним проєктом, що має потенціал задовольнити зростаючі потреби ігрової спільноти та стати важливим інструментом для систематизації та обміну ігровим досвідом.

1.2 Аналіз відомих програмних рішень для оцінювання та рецензування відеоігор

Сфера програмних рішень для оцінювання та рецензування відеоігор наразі представлена кількома типами платформ, кожна з яких має свої особливості, переваги та недоліки. Для обґрунтованого підходу до розробки власного програмного модуля необхідно детально проаналізувати наявні рішення.

Перша категорія – це великі агрегатори оцінок та рецензій. Ця категорія представлена сервісами, що збирають та узагальнюють оцінки від професійних критиків та звичайних користувачів. Найбільш авторитетним представником є Metacritic – платформа з тривалою історією роботи, що агрегує рецензії з сотень спеціалізованих ігрових видань та веб-сайтів. Сервіс також охоплює інші медіа-категорії, включаючи фільми, телесеріали та музичні альбоми, що робить його універсальним довідковим ресурсом для споживачів розваг. Сервіс використовує зважену систему оцінювання, де деякі джерела мають більший вплив на фінальний рейтинг, хоча точна методика залишається непрозорою. Вага кожного джерела визначається на основі його репутації, тиражу та авторитету в ігровій індустрії, що теоретично має забезпечувати більш якісну агрегацію оцінок. Користувачам надається можливість залишити власну оцінку за 10-бальною шкалою та написати короткий відгук, проте без можливості форматування тексту чи додавання медіаконтенту. Обмеженість соціальних функцій проявляється у відсутності можливості підписки на інших користувачів, формування спільнот за інтересами чи проведення дискусій. Відсутній також функціонал для ведення особистого ігрового журналу або створення колекцій [4].

OpenCritic позиціонується як більш прозорий конкурент Metacritic. Платформа фокусується на професійних оглядах і дає можливість фільтрувати огляди за критеріями, що важливі саме для конкретного користувача. Незважаючи на ці переваги, OpenCritic також не пропонує розвинених соціальних функцій, можливостей ведення ігрового журналу чи публікації детальних користувацьких рецензій. Обидві платформи використовуються гравцями переважно як довідкові ресурси для отримання базової інформації про якість гри, а не як інструменти для постійного використання та взаємодії з іншими гравцями [5]. Головна сторінка платформи Metacritic представлена на рисунку 1.1.

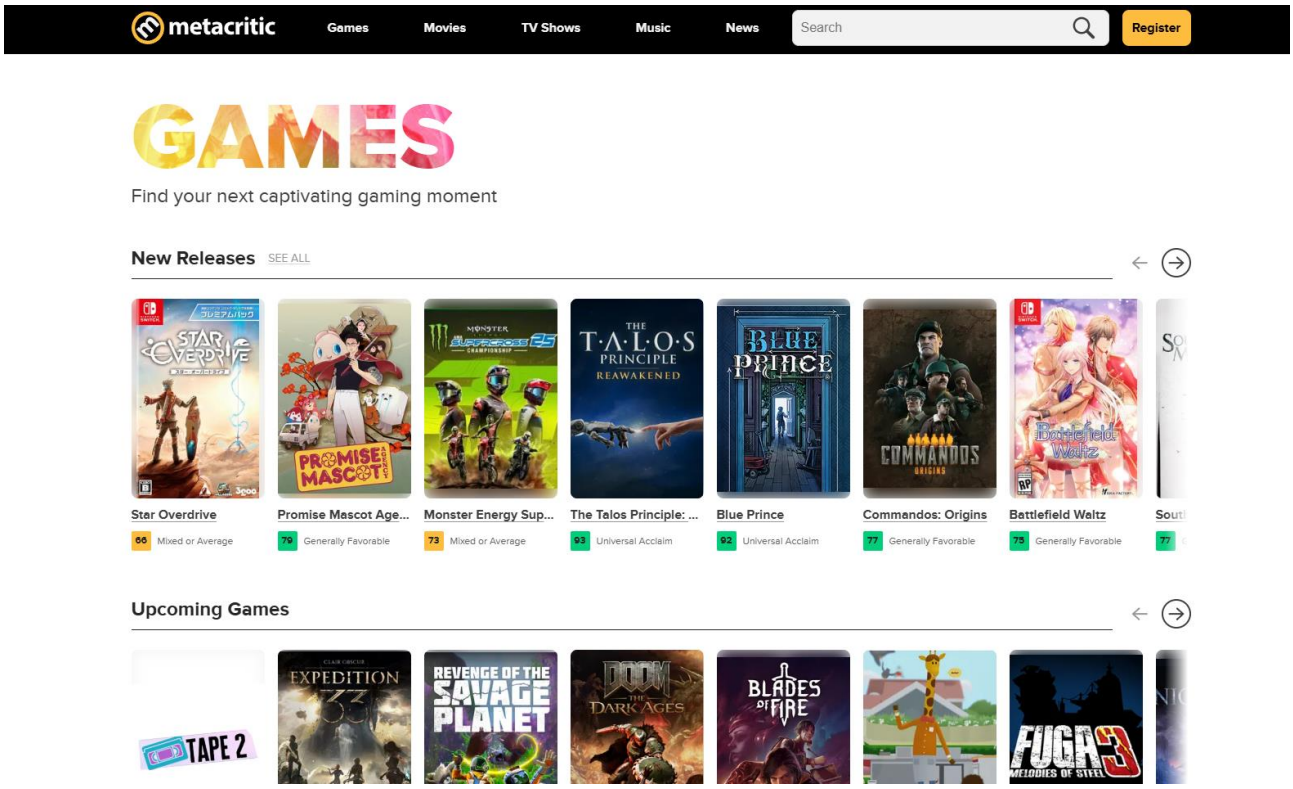

Рисунок 1.1 – Головна сторінка платформи Metacritic

Друга категорія – спеціалізовані платформи для ведення каталогу ігор. HowLongToBeat представляє собою унікальний сервіс, що першочергово зосереджений на відстеженні часу проходження ігор [6]. Платформа дозволяє користувачам відстежувати не лише основну сюжетну лінію, але й додатковий контент, проходження на 100% та різні стилі гри. Кожен користувач може вносити власні дані про тривалість проходження, що потім агрегуються для отримання середніх показників. Сервіс також пропонує функції для створення та ведення особистої ігрової колекції з розподілом на категорії "Пройдені", "Граю зараз", "У планах" та "Покинуті". Однак функціонал рецензування на цій платформі надзвичайно обмежений і дозволяє лише давати базові оцінки без можливості написання розгорнутих відгуків. Соціальні можливості зводяться до мінімуму: користувачі можуть підписуватися один на одного та бачити оновлення активності, але повноцінна комунікація між ними практично відсутня. Також платформа не підтримує пошук користувачів за географічним принципом

або інші розширені соціальні функції. Окремої уваги заслуговує Backloggd - відносно новий сервіс, який поєднує функції каталогізації ігрової колекції з елементами соціальної мережі. Платформа дозволяє користувачам вести журнал пройдених ігор, оцінювати їх та писати рецензії. Проте Backloggd має обмежені можливості для розгорнутого спілкування між користувачами та не пропонує функціоналу для пошуку гравців за географічним принципом, що є суттєвим недоліком для формування локальних спільнот. На рисунку 1.2 представлено головну сторінку платформи HowLongToBeat.

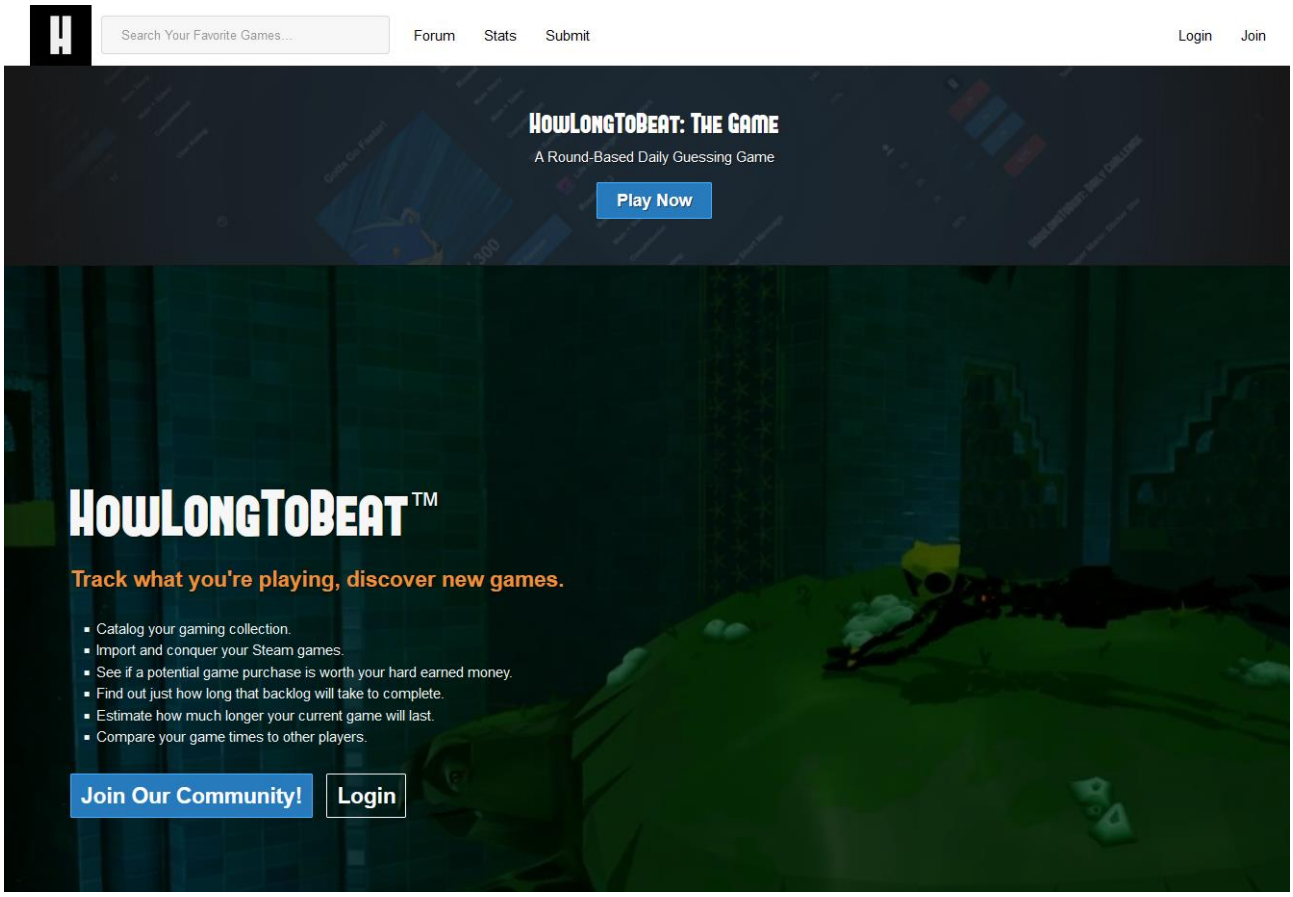

Рисунок 1.2 – Головна сторінка платформи HowLongToBeat

Третя категорія – інтегровані платформи дистрибуції з елементами соціальних мереж. Ця категорія представлена цифровими магазинами ігор, які додатково пропонують функції соціальної взаємодії та рецензування. Steam від

компанії Valve є безумовним лідером у цьому сегменті, пропонуючи широкий спектр соціальних можливостей [7]. Система рецензування Steam дозволяє користувачам залишати розгорнуті відгуки про ігри, голосувати за корисність рецензій інших користувачів, а також фільтрувати відгуки за різними параметрами, включаючи час гри автора рецензії. Платформа також надає функціонал для створення кураторських списків, де впливові користувачі або організації можуть рекомендувати ігри своїй аудиторії.

У Steam реалізовано розгалужену систему форумів для кожної гри, де користувачі можуть обговорювати різні аспекти, шукати рішення проблем або ділитися досвідом. Соціальний функціонал також включає можливість додавання друзів, створення та вступу до спільнот, участі в групових чатах та трансляції ігрового процесу. Однак система ведення ігрового журналу реалізована досить обмежено – користувачі можуть лише відмічати ігри як "бажані" або додавати їх до різних колекцій без детального відстеження прогресу чи ведення нотаток. Пошук користувачів за географічним принципом існує, але в дуже обмеженому вигляді – переважно для пошуку партнерів для гри з урахуванням регіону серверів, а не для формування локальних спільнот.

Epic Games Store, запущений у 2018 році як конкурент Steam, значно поступається своєму опоненту в соціальних функціях. Платформа лише нещодавно почала впроваджувати базові інструменти оцінювання ігор, при цьому повноцінна система рецензування та форуми досі відсутні. Соціальні можливості обмежуються простим списком друзів та обміном повідомленнями. Ведення ігрового журналу або створення колекцій також реалізовані мінімально через систему "бажаних" ігор [8].

GOG Galaxy позиціонується як універсальний ігровий лаунчер, що об'єднує ігри з різних платформ [9]. Сервіс пропонує базові можливості для відстеження ігрового часу та прогресу, включаючи статистику досягнень та інформацію про останні сеанси гри. Соціальні функції включають можливість додавання друзів з різних платформ та відстеження їхньої активності. Система

рецензування в GOG більш розвинена, ніж у Epic Games Store, але поступається Steam у гнучкості та функціональності. Пошук користувачів за географічними критеріями фактично відсутній. На рисунку 1.3 представлено головну сторінку платформи Steam.

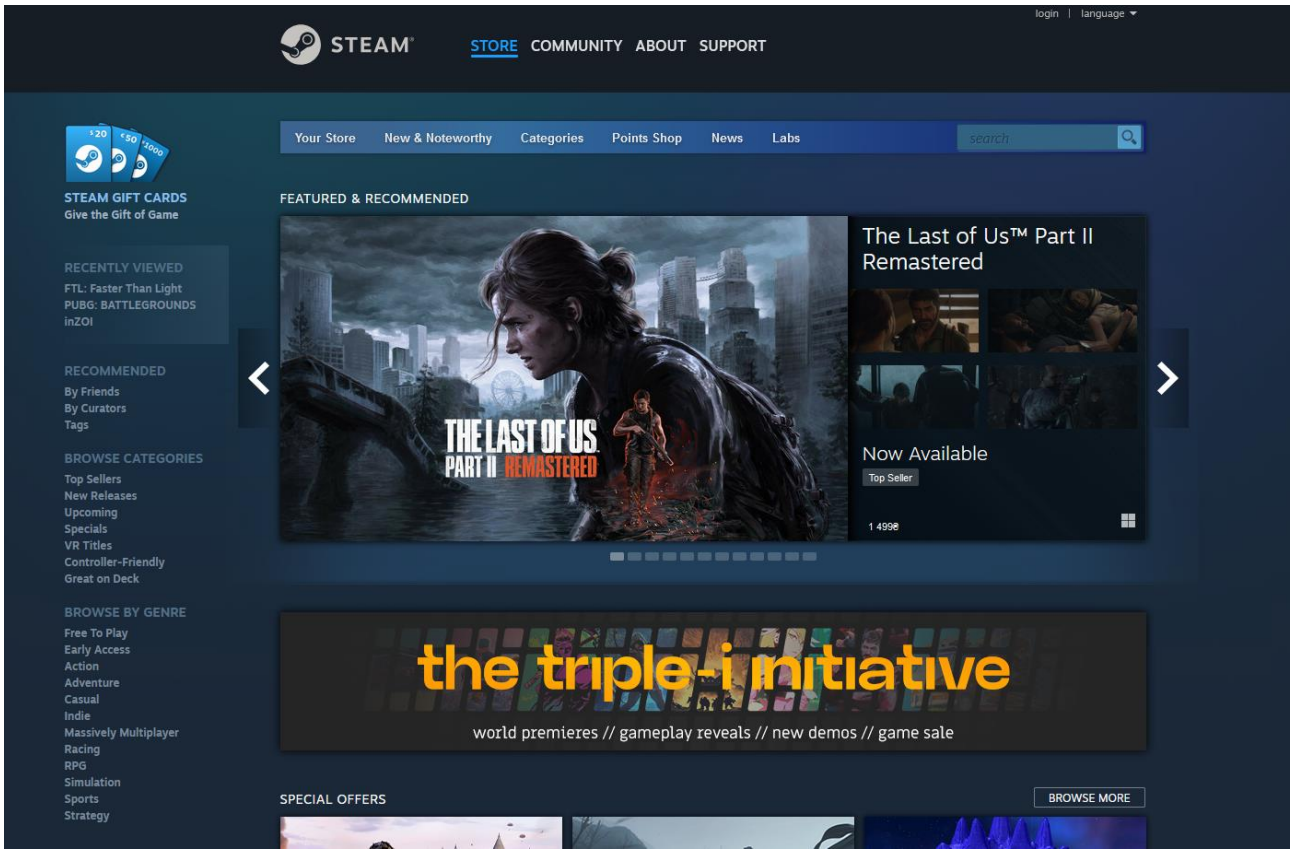

Рисунок 1.3 – Головна сторінка платформи Steam

Четверта категорія – платформи соціальних мереж для геймерів. Discord став домінуючою платформою для комунікації геймерів, пропонуючи текстові та голосові канали, відеоконференції, трансляції ігрового процесу та групові чати. Структура Discord базується на серверах, які можуть створюватися для конкретних ігор, тематик або спільнот. Кожен сервер може мати свої правила, ієрархічну структуру ролей та систему модерації [10]. Платформа чудово справляється з функцією миттєвого спілкування між гравцями, організації спільних ігрових сесій та формування довготривалих спільнот.

Однак Discord не розроблявся як інструмент для каталогізації чи рецензування ігор. На платформі відсутні системні інструменти для оцінювання

ігрових проєктів, ведення журналу пройдених ігор або детального порівняння характеристик різних ігор.

PlayStation Network та Xbox Live – це екосистеми для користувачів відповідних ігрових консолей [11]. Обидві платформи пропонують можливості для додавання друзів, обміну повідомленнями, створення груп та відстеження ігрової активності. Системи досягнень в обох мережах дозволяють відстежувати прогрес у різних іграх, а статистика ігрового часу дає уявлення про інтенсивність використання конкретних ігрових проєктів [12].

Функціонал рецензування в цих мережах обмежений простими оцінками (зірками або балами) та короткими відгуками без можливості форматування. Соціальні функції орієнтовані переважно на пошук партнерів для спільної гри, а не на формування тематичних спільнот чи обговорення ігрової індустрії в цілому. Пошук користувачів за географічним принципом реалізований переважно для оптимізації мережевого з'єднання, а не для соціальної взаємодії. Головний недолік цих платформ – обмеженість однією екосистемою, що не дозволяє охопити весь спектр ігрових проєктів, доступних користувачу на різних пристроях. На рисунку 1.4 зображено головну сторінку платформи Discord.

Серед веб-сервісів для трекінгу та рецензування відеоігор також варто відзначити RAWG та IGN. RAWG позиціонується як "IMDb для відеоігор", пропонуючи обширну базу даних ігрових проєктів з детальною інформацією про кожну гру [13]. Платформа дозволяє користувачам відстежувати свій ігровий прогрес через категорії "Пройдені", "Граю зараз", "Планую грати" та "Покинуті". Система оцінювання обмежується простою 5-зірковою шкалою, а рецензії – короткими текстовими відгуками. Соціальні функції включають можливість підписатися на інших користувачів та переглядати їхню активність, але без розвинених інструментів комунікації. Відсутній форумний функціонал та можливості для пошуку користувачів за географічними критеріями.

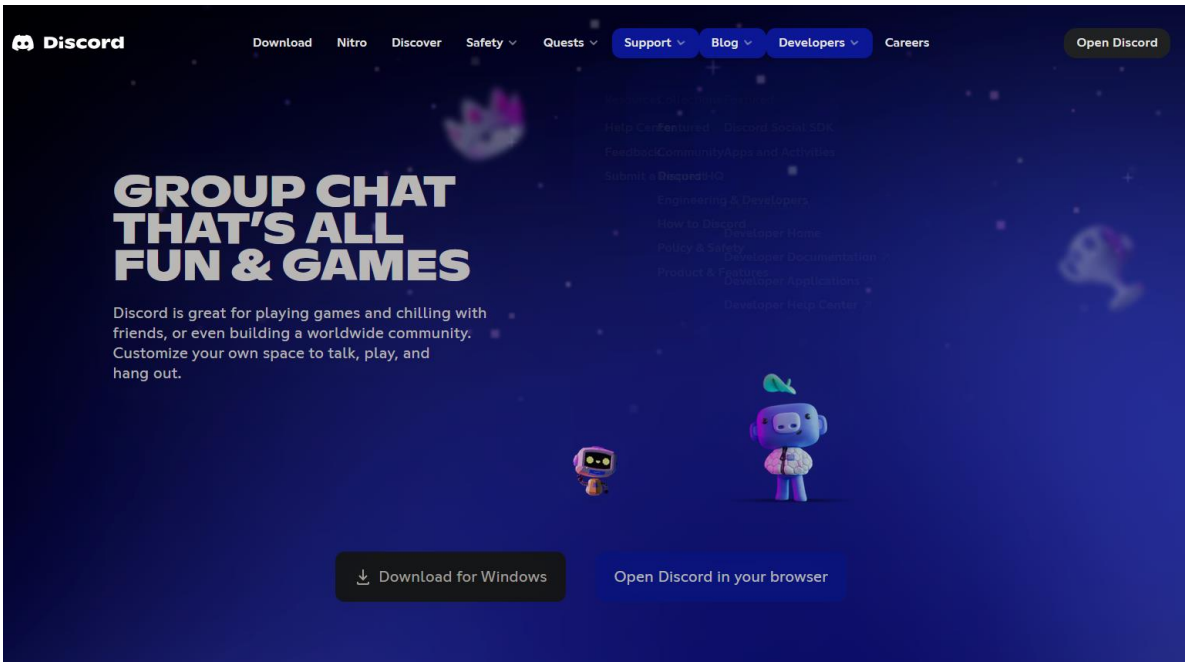

Рисунок 1.4 – Головна сторінка платформи Discord

IGN – один з найбільших ігрових медіа-порталів, що фокусується на професійних рецензіях, новинах та аналітичних матеріалах. Платформа використовує 10-бальну систему оцінювання та публікує детальні експертні огляди [14]. Можливості для користувацького контенту обмежуються коментарями під публікаціями та участю в обговореннях. IGN не надає інструментів для ведення особистого ігрового журналу або створення колекцій. Соціальні функції мінімальні та зводяться до базової взаємодії в коментарях. Пошук користувачів за географічним принципом відсутній повністю. На рисунку 1.5 зображено головну сторінку платформи IGN

Аналіз наявних рішень виявив кілька суттєвих прогалин у функціональності, які можуть бути усунуті в рамках розробки нового програмного модуля. Жодна з існуючих платформ, не поєднує функції детального ведення ігрового журналу, розгорнутого рецензування, глибокої соціальної взаємодії та можливості пошуку користувачів за географічним принципом.

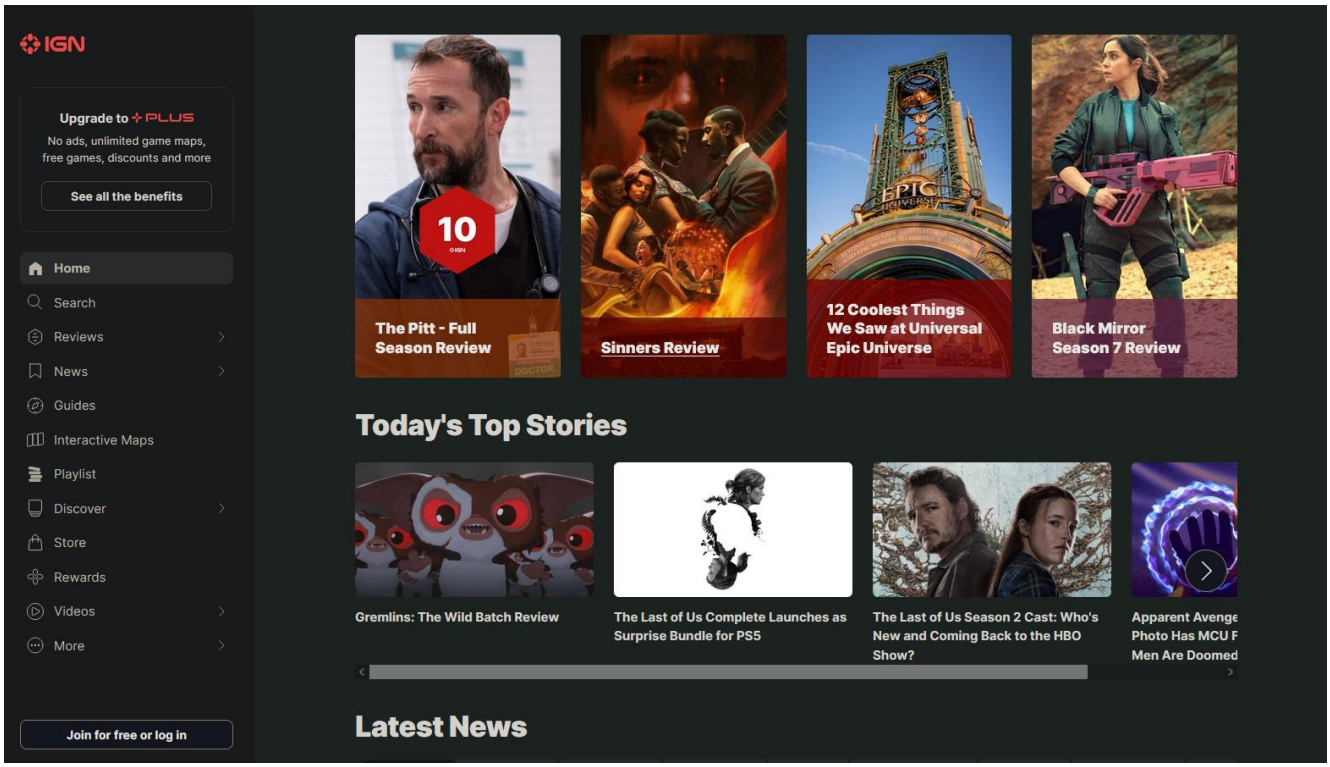

Рисунок 1.5 – Головна сторінка платформи IGN

Результати аналізу засвідчують необхідність розробки нового програмного рішення, яке б усунуло виявлені недоліки та запропонувало унікальний набір функцій для задоволення потреб сучасних гравців. Особливу увагу варто приділити інтеграції соціальних функцій безпосередньо в процес ведення ігрового журналу та рецензування, а також реалізації механізмів формування локальних спільнот за географічним принципом.

У таблиці 1.1 наведено порівняльний аналіз платформ оцінювання та рецензування відеоігор.

Як бачимо з порівняльної таблиці, жодна з існуючих платформ не реалізує повний спектр функціональних можливостей, необхідних для задоволення потреб користувачів. Функція ведення журналу ігор представлена лише в сервісі Backloggd, більшість платформ (Metacritic, OpenCritic, HowLongToBeat, Backloggd, Steam, GOG Galaxy, RAWG, IGN) пропонують системи оцінювання та розгорнуті рецензії, проте форуми наявні тільки в Steam та Discord. Особливо помітним є те, що жодна з проаналізованих платформ не підтримує пошук

користувачів за географічною локацією та не надає можливості створення користувацьких постів на сторінці профілю. Це підтверджує актуальність розробки нового програмного модуля, який об'єднає всі необхідні функціональні можливості в одному сервісі та усуне виявлені недоліки існуючих рішень.

Таблиця 1.1. Порівняльний аналіз платформ оцінювання та рецензування відеоігор.

Назва сервісу	Ведення журналу ігор	Система оцінювання	Розгорнуті рецензії	Форуми	Пошук за країною або містом	Користувацькі пости на сторінці профілю
Metacritic	-	+	+	-	-	-
OpenCritic	-	+	+	-	-	-
HowLongToBeat	-	+	+	-	-	-
Backloggd	+	+	+	-	-	-
Steam	-	+	+	+	-	-
Epic Games Store	-	-	-	-	-	-
GOG Galaxy	-	+	+	-	-	-
Discord	-	-	-	+	-	-
Playstation Network	-	-	-	-	-	-
Xbox Live	-	-	-	-	-	-
RAWG	-	+	+	-	-	-
IGN	-	+	+	-	-	-

1.3 Формулювання вимог та постановка задачі

Аналіз предметної області показав необхідність розробки програмного модуля для оцінювання та рецензування відеоігор. Даний модуль дозволить користувачам ділитися враженнями від проходження ігор, писати рецензії, ставити оцінки, а також спілкуватися з іншими користувачами через систему форумів та персональних постів.

На основі аналізу потреб цільової аудиторії було сформульовано наступні функціональні вимоги до програмного модуля:

- Забезпечення можливості ведення користувачем особистого журналу пройдених відеоігор з фіксацією дати проходження. Користувач зможе відмічати завершені ігри та зберігати базову інформацію про них без додаткових деталей щодо статусів, тегів або розподілу ігрового часу.
- Реалізація функціоналу для виставлення чисельних оцінок іграм за визначеною шкалою та написання детальних текстових відгуків. Користувачі матимуть можливість ділитися своїми враженнями та оцінювати різні аспекти ігрового досвіду для подальшого використання іншими учасниками спільноти.
- Створення персоналізованих текстових форумів для кожної гри, де користувачі зможуть обговорювати різні аспекти гри, ділитися порадами та враженнями.
- Забезпечення функціоналу для створення текстових постів на сторінці профілю користувача з можливістю отримання відповідей від інших користувачів. Система відображатиме пости в хронологічному порядку та забезпечуватиме зручну навігацію між дописами та коментарями.
- Реалізація окремої сторінки для пошуку та перегляду списку користувачів за обраним містом або країною. Користувачі зможуть вказувати свою локацію в профілі та знаходити інших учасників спільноти з тієї ж географічної зони для обміну досвідом та спілкування.

Програмний модуль повинен бути реалізований відповідно до всіх зазначених функціональних та нефункціональних вимог, забезпечуючи стабільну роботу, високу продуктивність та зручність використання для всіх категорій користувачів.

1.4 Висновок до розділу 1

Обґрунтовано актуальність розробки програмного модуля для оцінювання та рецензування відеоігор на основі аналізу існуючих рішень та тенденцій розвитку ігрової індустрії. Виявлено, що сучасний ринок відеоігор демонструє стабільне зростання охоплюючи все більш різноманітну аудиторію.

Здійснено аналіз відомих програмних рішень у сфері оцінювання та рецензування відеоігор. Аналіз показав що є ряд недоліків у існуючих платформах. Більшість платформ зосереджені виключно на оцінюванні та коротких відгуках, при цьому лише Backloggd пропонує повноцінне ведення журналу ігор, а форумні обговорення доступні тільки у Steam та Discord. Жодна з проаналізованих платформ не надає можливості пошуку користувачів за географічним принципом та не забезпечує створення користувацьких постів на сторінці профілю, що суттєво обмежує формування локальних спільнот та повноцінну соціальну взаємодію між гравцями зі спільними інтересами.

На основі виконаного аналізу сформульовано основні вимоги до програмного модуля оцінювання та рецензування відеоігор. Він має забезпечити повноцінну реалізацію всіх необхідних компонентів системи: журналювання ігор, рецензування, форумне спілкування, користувацькі пости та пошук користувачів за географічною локацією.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ ОЦІНЮВАННЯ ТА РЕЦЕНЗУВАННЯ ВІДЕОІГР

2.1 Обґрунтування вибору архітектури для програмного модуля оцінювання та рецензування відеоігор

У процесі розробки програмного модуля оцінювання та рецензування відеоігор одним із ключових етапів є вибір архітектурного підходу, який забезпечить ефективність, масштабованість, зручність у підтримці та подальшому розширенні системи. Архітектура програмного забезпечення визначає взаємодію між основними компонентами модуля, способи обробки запитів користувачів, зберігання та обробки даних, а також інтеграцію з іншими сервісами або підсистемами.

У сучасному програмуванні існує багато різних способів організації структури програм. Кожен із цих архітектурних варіантів має свої сильні сторони та певні недоліки. Серед найбільш популярних підходів можна назвати монолітну архітектуру, де всі частини системи тісно пов’язані між собою; мікросервісну архітектуру, яка розділяє програму на незалежні модулі з окремою відповідальністю; архітектурний шаблон Model-View-Controller (MVC), що дозволяє чітко поділити логіку, інтерфейс і обробку даних; а також клієнт-серверну модель, у якій функції взаємодії з користувачем і обробки даних виконуються окремо [15].

Монолітна архітектура представляє собою традиційний підхід до побудови програмних систем, у якому всі функціональні компоненти додатку об'єднані в єдину нероздільну структуру. У такій архітектурі вся бізнес-логіка, шар доступу до даних, інтерфейс користувача та допоміжні функції існують у рамках єдиного процесу або застосунку, що виконується на одному сервері. Всі компоненти системи тісно пов'язані між собою та спільно використовують ресурси

обчислювального середовища, пам'ять, файлову систему та базу даних [16]. Структура монолітної архітектури зображена на рисунку 2.1.

Історично монолітна архітектура була домінуючим підходом у розробці програмного забезпечення, особливо до поширення хмарних технологій та контейнеризації. Вона характеризується централізованою логікою управління, єдиною кодовою базою та спільним процесом розгортання для всіх функціональних компонентів системи.

У монолітному додатку внутрішні взаємодії між компонентами відбуваються через виклики функцій або методів в оперативній пам'яті, що забезпечує високу швидкість комунікації без мережевих затримок. Усі модулі компілюються разом і розгортаються як єдиний артефакт на сервері додатків, що суттєво спрощує процес інсталяції та конфігурації системи.

Монолітна архітектура має ряд суттєвих переваг. Насамперед, вона відзначається простотою проектування, розробки та розгортання, особливо на початкових етапах проекту. Розробники можуть швидко створювати функціональність без необхідності проектування складних механізмів міжсервісної комунікації. Моноліт також забезпечує високу продуктивність внутрішніх взаємодій між компонентами, оскільки вони відбуваються без мережевих викликів та серіалізації даних.

Однак монолітна архітектура має суттєві обмеження, які особливо проявляються зі зростанням системи. По-перше, масштабованість монолітного додатку обмежена вертикальним масштабуванням, тобто нарощуванням потужності сервера, що має фізичні та економічні обмеження. Горизонтальне масштабування можливе лише шляхом повного дублювання всього додатку, що часто неефективно, якщо окремі компоненти мають різні вимоги до обчислювальних ресурсів.

З розвитком проекту кодова база моноліту стає дедалі складнішою, що ускладнює її розуміння, обслуговування та модифікацію. Внесення змін у одну частину системи може мати непередбачувані наслідки для інших компонентів

через високий рівень зв'язності. Крім того, оновлення навіть невеликої частини функціональності вимагає перезавантаження всього додатку, що призводить до тимчасової недоступності системи.

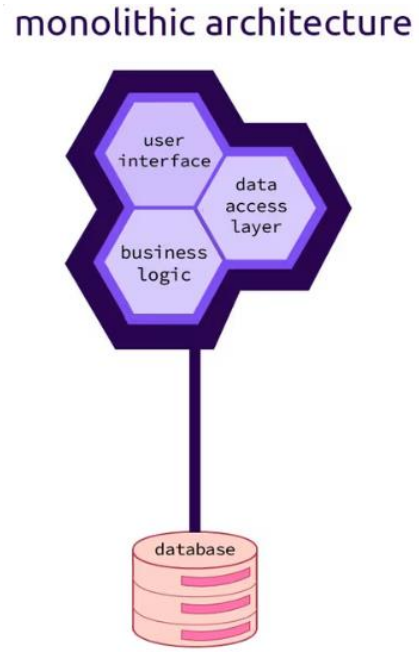

Рисунок 2.1 – Структура монолітної архітектури

У контексті сучасних тенденцій розробки програмного забезпечення монолітна архітектура залишається релевантною для певних типів проектів, особливо для стартапів та невеликих систем з обмеженими ресурсами, де швидкість розробки та простота підтримки є пріоритетними факторами. Проте для складних, масштабних систем з високими вимогами до гнучкості, масштабованості та надійності монолітна архітектура часто виявляється неоптимальним вибором у довгостроковій перспективі, поступаючись більш розподіленим архітектурним підходам [17].

Мікросервісна архітектура є сучасним підходом до проектування програмних систем, який передбачає декомпозицію додатку на набір невеликих, автономних сервісів, що взаємодіють між собою через мережеві протоколи. Кожен мікросервіс фокусується на реалізації конкретної бізнес-функції та може

розроблятися, розгортатися та масштабуватися незалежно від інших компонентів системи. Це принципово відрізняє мікросервісний підхід від монолітної архітектури, де всі функції інтегровані в єдину кодову базу. Структура мікросервісної архітектури зображена на рисунку 2.2

Мікросервісна архітектура надає низку суттєвих переваг для складних, масштабних систем. Насамперед, вона забезпечує високу масштабованість, дозволяючи незалежно нарощувати потужності лише тих сервісів, які потребують додаткових ресурсів. Це особливо важливо для систем з нерівномірним навантаженням на різні функціональні компоненти.

Стійкість до відмов є іншою ключовою перевагою мікросервісного підходу. Оскільки сервіси працюють незалежно, несправність одного компонента не призводить до загального збою системи. Правильно спроектована мікросервісна архітектура включає механізми відмовостійкості, такі як розривники мережі (circuit breakers), повторні спроби та резервні системи, що підвищують надійність системи в цілому.

Однак мікросервісна архітектура привносить свої специфічні виклики та складності. Розподілена природа таких систем значно ускладнює процеси налагодження, тестування та моніторингу. Відстеження запитів, що проходять через кілька сервісів, вимагає спеціалізованих інструментів розподіленого трасування [18].

Model-View-Controller, або MVC, є архітектурним патерном проектування, який розділяє додаток на три взаємопов'язані компоненти для відокремлення внутрішньої логіки від способів представлення та обробки інформації користувачем. Цей патерн був вперше описаний Трюгве Реенскаугом у 1979 році для платформи Smalltalk і з того часу став одним з найбільш впливових та широко використовуваних підходів до структурування програмного забезпечення, особливо для веб-додатків та додатків з графічним інтерфейсом користувача.

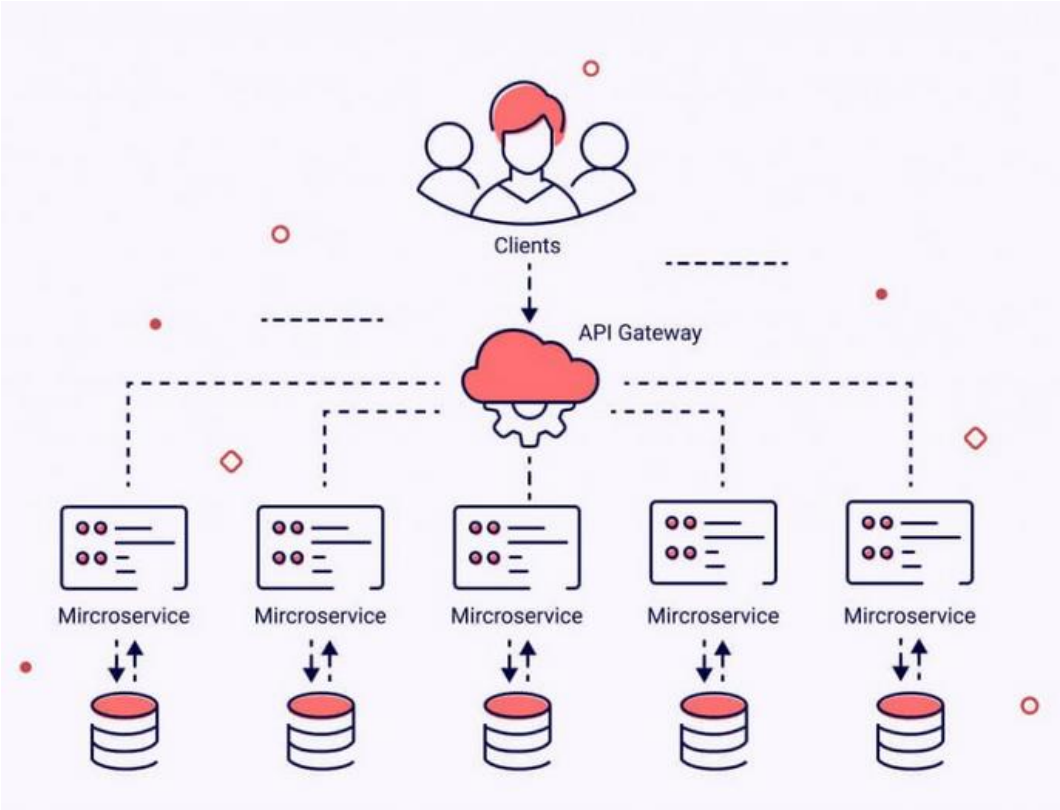

Рисунок 2.2 – Структура мікросервісної архітектури

У структурі MVC кожен з трьох компонентів має чітко визначену роль та відповідальність. Model (модель) представляє собою центральний компонент, який відповідає за управління даними, бізнес-логікою та правилами додатку. Він забезпечує доступ до даних та їх маніпуляцію, реагує на запити про зміну стану від контролера та повідомляє пов'язані з ним представлення про зміни в даних. Модель є незалежною від користувацького інтерфейсу і працює виключно з бізнес-доменом та даними системи. Структура MVC архітектури зображена на рисунку 2.3.

View (представлення) відповідає за відображення даних з моделі у формі, зручній для взаємодії з користувачем. Це може бути веб-сторінка, діаграма, таблиця або будь-який інший елемент інтерфейсу. Представлення отримує дані для відображення від моделі, але не має можливості безпосередньо змінювати її стан. Воно також може повідомляти контролеру про дії користувача, такі як натискання кнопок або введення даних.

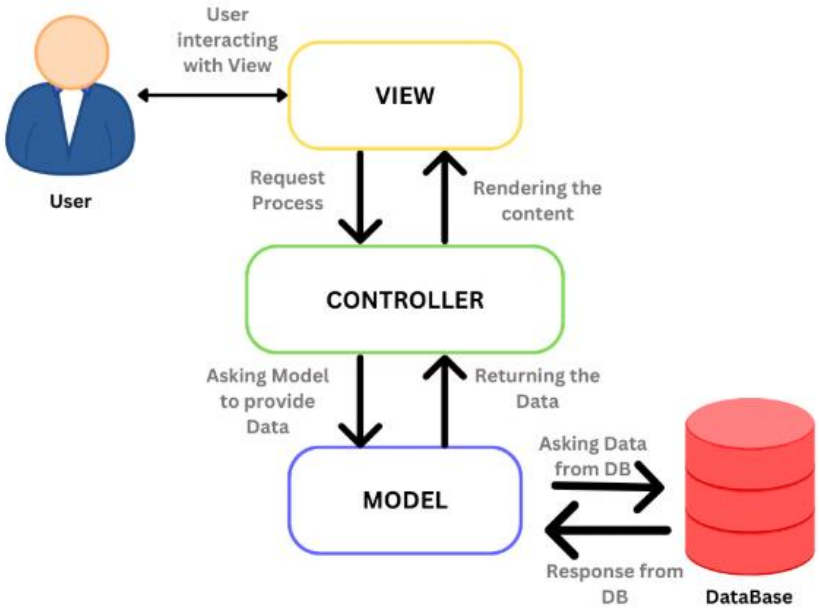

Рисунок 2.3 – Структура MVC архітектури

Controller (контролер) виступає в ролі посередника між моделлю та представленням. Він обробляє вхідні запити користувача, отримані від представлення, маніпулює моделлю відповідно до цих запитів та вибирає відповідне представлення для відображення оновлених даних. Контролер відіграє ключову роль у забезпеченні принципу розділення відповідальності, ізолюючи бізнес-логіку від інтерфейсу користувача.

У типовому сценарії взаємодії в MVC архітектурі користувач виконує дію через інтерфейс (представлення), ця дія перехоплюється контролером, який обробляє її, взаємодіє з моделлю для оновлення або отримання даних, і після цього вибирає відповідне представлення для відображення результатів операції. Така структура забезпечує чіткий потік даних та розділення відповідальності між компонентами системи.

Впровадження MVC приносить численні переваги для розробки програмного забезпечення. Насамперед, це покращена підтримуваність коду

завдяки чіткому розділенню відповідальності, що дозволяє розробникам легше розуміти, модифікувати та розширювати окремі компоненти без впливу на інші. Паралельна розробка стає можливою, оскільки різні команди можуть одночасно працювати над моделлю, представленням та контролером.

Незважаючи на численні переваги, MVC має свої обмеження та виклики. Одним з основних недоліків є потенційне ускладнення архітектури для простих додатків, де повне розділення на три компоненти може виявитися надмірним. Крім того, строге дотримання патерну MVC може призвести до збільшення кількості файлів та класів, що ускладнює навігацію по кодовій базі для нових розробників [19].

Клієнт-серверна архітектура є одним із фундаментальних архітектурних підходів у розробці розподілених інформаційних систем, який передбачає поділ функціональності між постачальниками послуг (серверами) та споживачами цих послуг (клієнтами). Ця архітектурна парадигма формує основу сучасного інтернету та більшості корпоративних інформаційних систем, забезпечуючи ефективний розподіл обчислювальних ресурсів та даних між різними компонентами мережі.

У своїй сутності клієнт-серверна архітектура ґрунтується на чіткому розподілі ролей. Сервер виступає централізованим постачальником ресурсів або послуг, які можуть включати доступ до даних, обчислювальні можливості, управління спільними ресурсами або виконання складних бізнес-операцій. Сервер пасивно очікує запитів від клієнтів, обробляє їх згідно з визначеними правилами та повертає результати. Він зазвичай працює безперервно, забезпечуючи доступність послуг у будь-який момент часу.

Клієнт, у свою чергу, представляє активну сторону взаємодії, яка ініціює запити до сервера для отримання необхідних ресурсів або виконання певних операцій. Клієнтська частина системи зазвичай зосереджена на забезпеченні інтерфейсу користувача, локальній обробці даних та представленні інформації в зручному для користувача форматі. Після завершення взаємодії з сервером

клієнт може продовжувати функціонувати автономно, використовуючи отримані дані, або завершити свою роботу до наступного сеансу.

Клієнт-серверна архітектура надає численні переваги, які роблять її привабливим вибором для багатьох типів систем. Централізоване управління даними є однією з ключових переваг, що забезпечує цілісність та узгодженість інформації. Всі модифікації даних відбуваються на сервері за єдиними правилами, що запобігає різночитанням та конфліктам, які могли б виникнути при розподіленому управлінні. Структура клієнт-серверної архітектури зображена на рисунку 2.4

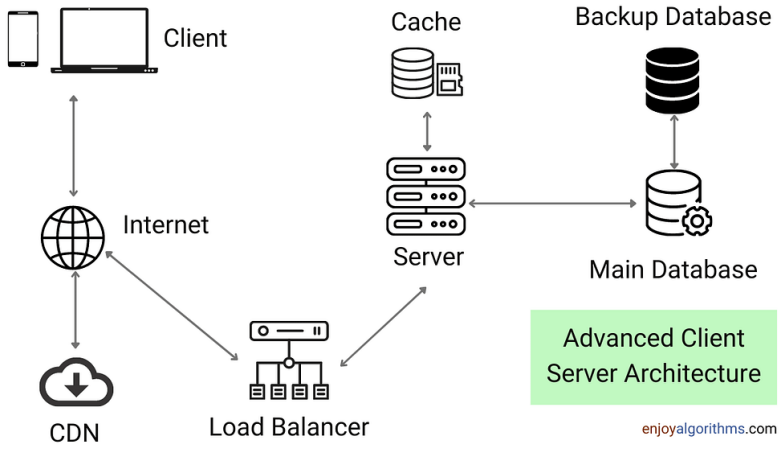

Рисунок 2.4 – Структура клієнт-серверної архітектури

Попри численні переваги, клієнт-серверна архітектура не позбавлена певних обмежень та викликів. Залежність від мережевого з'єднання є одним з найбільш очевидних обмежень, оскільки порушення зв'язку між клієнтом та сервером може призвести до недоступності системи. Для критичних додатків це вимагає впровадження механізмів роботи в автономному режимі та синхронізації даних при відновленні з'єднання.

Сервер може стати єдиною точкою відмови, що підвищує ризик недоступності всієї системи у випадку технічних проблем на серверній стороні. Для мінімізації цього ризику застосовуються різноманітні стратегії забезпечення

високої доступності, такі як кластеризація, резервне копіювання та автоматичне перемикання на резервні системи.

У контексті нашого програмного модуля клієнт-серверна архітектура представляється оптимальним вибором з огляду на специфіку завдань, які має вирішувати система, та наявні ресурси. Централізоване управління даними та бізнес-логікою на сервері забезпечить цілісність інформації та спростить процеси оновлення та підтримки. Клієнтська частина зосередиться на забезпеченні зручного та інтуїтивного інтерфейсу користувача, залишаючи ресурсомісткі операції обробки даних серверній стороні. Такий підхід також підвищує рівень безпеки, оскільки чутливі дані не зберігаються на клієнтському пристрої, а обробляються виключно на сервері.

Здатність клієнт-серверної архітектури до горизонтального масштабування дозволить ефективно реагувати на зростання навантаження без необхідності повного переосмислення архітектури. Крім того, розділення відповідальності між клієнтом та сервером сприятиме чіткій організації кодової бази та паралельній розробці різними командами, що прискорить процес впровадження нової функціональності [20]. Це також дозволяє більш гнучко розподіляти ресурси між фронтендом і бекендом, оптимізуючи продуктивність системи на різних рівнях.

На основі детального аналізу різних архітектурних підходів для нашого програмного модуля обрано клієнт-серверну архітектуру як оптимальне рішення, що найкраще відповідає поставленим вимогам та обмеженням. Ключовими факторами на користь цього вибору стали можливість централізованого управління даними, чіткий розподіл відповідальності між компонентами системи, висока масштабованість та безпека даних. На відміну від монолітної архітектури, клієнт-серверний підхід забезпечує більшу гнучкість при модифікації та оновленні окремих компонентів системи, а порівняно з мікросервісною архітектурою – пропонує простішу інфраструктуру та нижчі операційні витрати на початковому етапі розробки.

2.2 Розробка UML-діаграм розгортання та прецедентів програмного модуля оцінювання та рецензування відеоігор

Уніфікована мова моделювання (UML) є стандартизованою графічною нотацією, яка широко використовується в галузі програмної інженерії для візуалізації, специфікації, конструювання та документування програмних систем. UML-діаграми дозволяють розробникам створювати наочні моделі складних систем, що значно полегшує процес проектування, розробки та подальшого супроводу програмного забезпечення. Стандарт включає 14 типів діаграм, які поділяються на три основні категорії: структурні діаграми (показують статичну архітектуру системи), поведінкові діаграми (описують динамічну поведінку) та діаграми взаємодії (деталізують комунікацію між об'єктами).

Діаграми розгортання є одним із ключових типів структурних UML-діаграм, що відіграють важливу роль у процесі проектування програмного забезпечення. Вони зосереджуються на фізичній архітектурі системи та описують, яким чином програмні компоненти розподіляються на реальному обладнанні. На відміну від інших типів UML-діаграм, які здебільшого стосуються логічної структури програмного забезпечення, діаграми розгортання показують конкретне фізичне середовище, в якому система буде функціонувати. Діаграма розгортання програмного модуля зображена на рисунку 2.6

Діаграми прецедентів, також відомі як діаграми варіантів використання або Use Case діаграми, є одним із фундаментальних типів поведінкових UML-діаграм, що відіграють ключову роль на початкових етапах розробки програмного забезпечення. Вони представляють високорівневий погляд на систему з точки зору користувачів і слугують потужним інструментом для збору та документування функціональних вимог. Діаграма прецедентів програмного модуля зображена на рисунку 2.7.

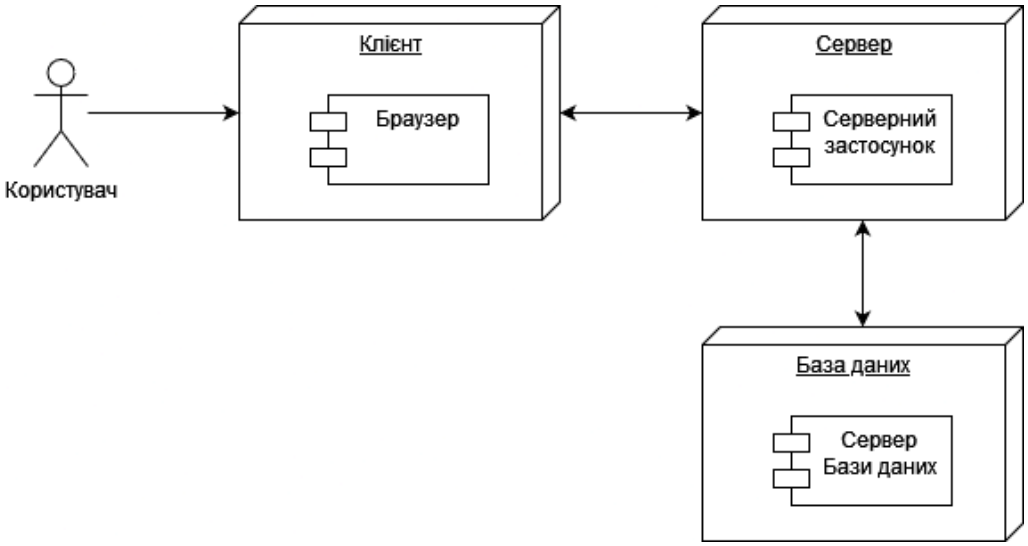

Рисунок 2.6 – Діаграма розгортання програмного модуля

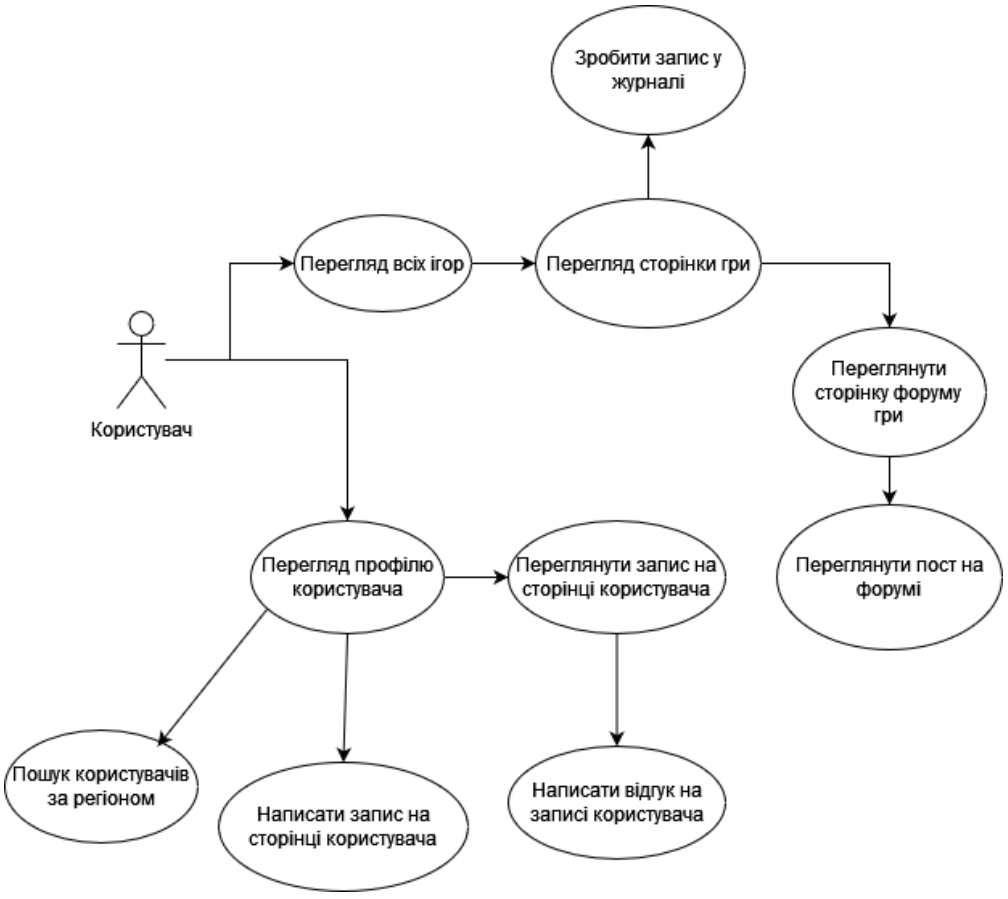

Рисунок 2.7 – Діаграма прецедентів програмного модуля

Взаємодія між акторами та прецедентами зображується за допомогою суцільних ліній, що називаються асоціаціями. Вони показують, які актори

ініціюють які прецеденти або беруть у них участь. Наприклад, асоціація між актором "Користувач" і прецедентом "Перегляд всіх ігор" вказує на те, що клієнт може виконати цю дію в системі.

2.3 Розробка бази даних для програмного модуля оцінювання та рецензування відеоігор

У сучасному світі програмних технологій ефективне керування даними є критичним фактором для успішної роботи будь-якого програмного забезпечення. Розробка бази даних – це фундаментальний етап створення програмного модуля, який визначає його продуктивність, масштабованість та надійність.

SQL бази даних залишаються одним із найпопулярніших та перевірених рішень для структурованого зберігання та обробки інформації. Їхні переваги включають потужну мову запитів, надійну підтримку цілісності даних та широкий спектр доступних систем управління базами даних (СУБД), таких як MySQL, PostgreSQL, Microsoft SQL Server та інші.

ER-діаграми (Entity-Relationship diagrams) – це графічне представлення моделі даних, яке широко використовується в процесі проектування баз даних. Вони дозволяють візуалізувати структуру даних, визначаючи основні сутності, атрибути та зв’язки між ними. Основна мета ER-діаграм – забезпечити чітке розуміння логіки побудови даних ще до етапу фізичної реалізації бази даних.

Кожна ER-діаграма складається з трьох компонентів: сутностей, які представляють об’єкти або поняття з реального світу; атрибутів, що описують властивості цих сутностей; та зв’язків, які відображають взаємодії між сутностями. Сутності зазвичай зображуються у вигляді прямокутників, атрибути – еліпсами, а зв’язки – ромбами

ER-діаграми також можуть містити додаткові елементи, такі як слабкі сутності або множинності зв’язків, які уточнюють характер взаємодії між

сутностями. Наприклад, зв’язок може бути один-до-багатьох, багато-до-багатьох або один-до-одного, і це важливо враховувати при проектуванні. Множинність визначає, скільки об'єктів однієї сутності може бути пов’язано з об’єктами іншої. Це дозволяє уникнути логічних помилок у побудові структури даних. Рисунок ER-діаграми програмного модуля зображена на рисунку 2.8.

За розробленою ER-діаграмою розробимо схему бази даних. Спроектована схема бази даних зображена на рисунку 2.9.

Наведена схема бази даних складається з таких таблиць: users, games, ratings, logs, forum_posts, forum_replies, user_posts, user_replies.

Для опису предметної області програмного модуля для оцінювання та рецензування відеоігор необхідні такі сутності: користувач, оцінки, ігра, журнал, пост на форумі, відповідь на пост форума, пост користувача, відповідь на пост користувача.

Виділимо характеристики зазначених сутностей:

- користувач: ім’я користувача, пароль, стать, країна, місто;
- гра: id гри, назва, категорія, опис, id обкладинки;
- оцінки: ім’я користувача, id гри, оцінка;
- журнал: ім’я користувача, id гри, дата запису, 100% проходження, переграш, відгук, оцінка запису, дата створення запису;
- пост на форумі: id поста форума, ім’я користувача, id гри, тема поста, текст поста, дата створення посту форума;
- відповідь на пост форума: id поста форума, ім’я користувача, текст відповіді на пост форума, дата створення відповіді на пост форума;
- пост користувача: id поста користувача, id автора, ім’я користувача, текст поста користувача, дата створення поста користувача;
- відповідь на пост користувача: id поста користувача, ім’я користувача, текст відповіді на пост користувача, дата створення відповіді на пост користувача.

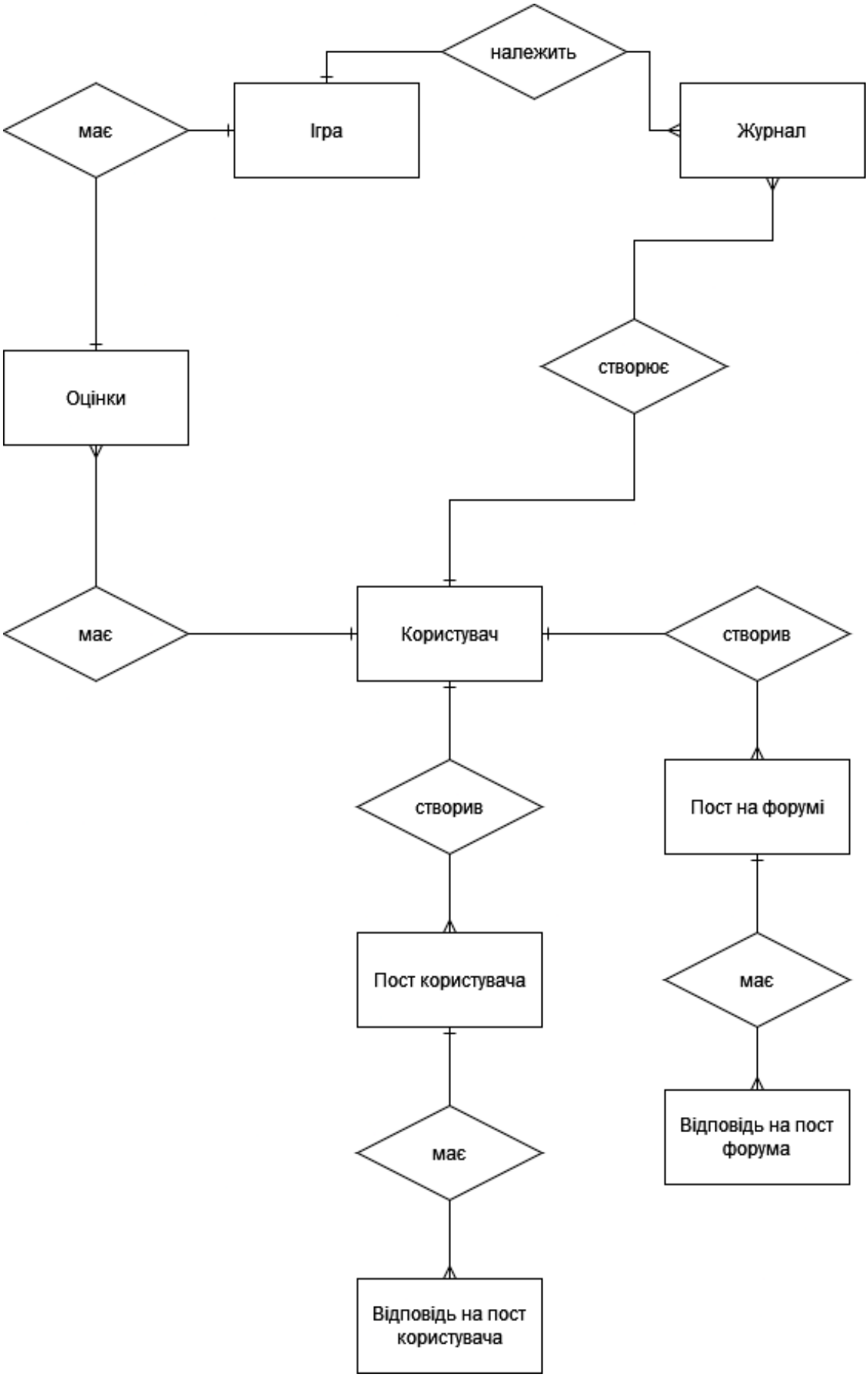

Рисунок 2.8 – ER-діаграма програмного модуля

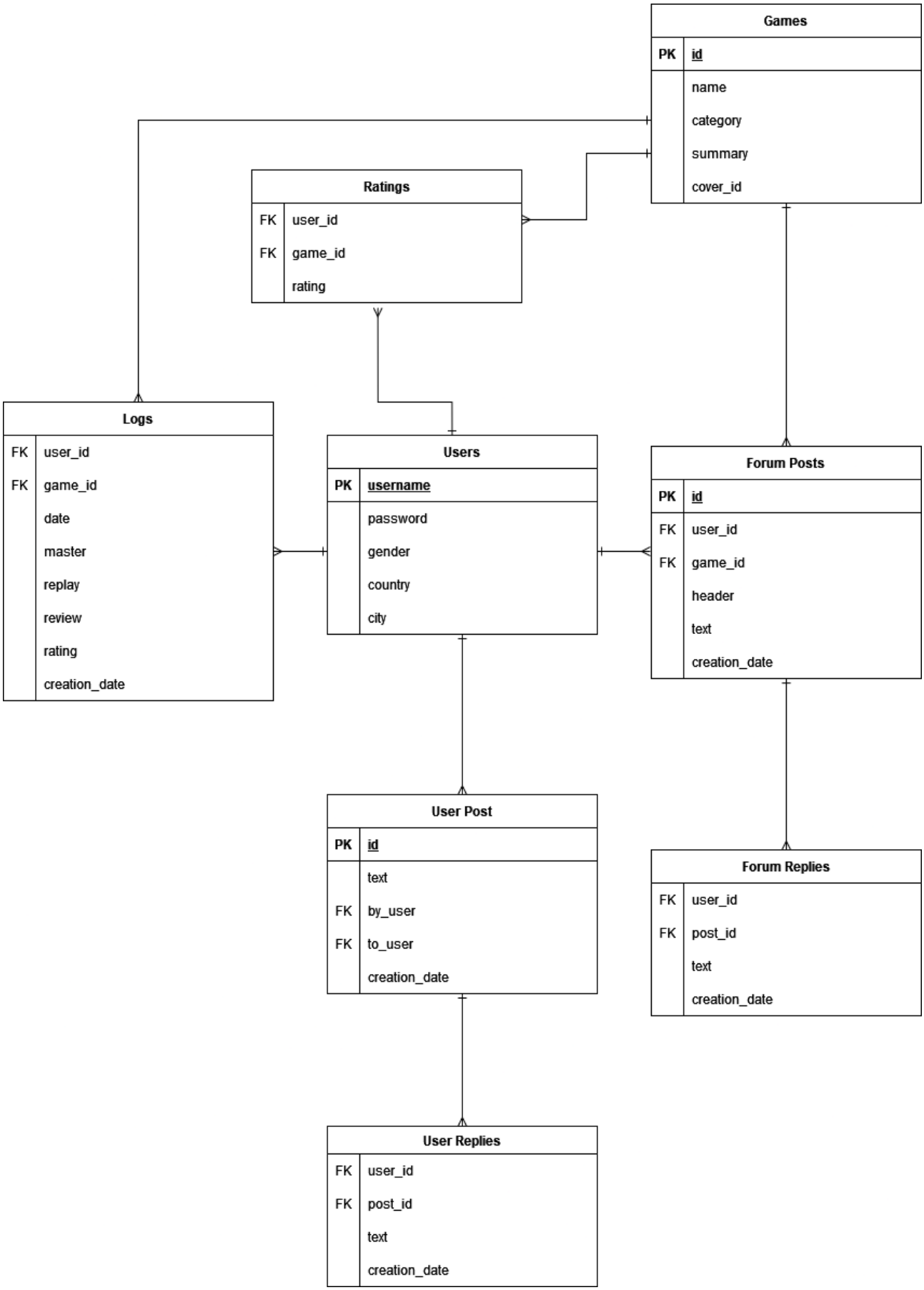

Рисунок 2.9 – Схема бази даних програмного модуля

Для визначення ступеню універсального відношення потрібно перерахувати всі атрибути, що описують сутності. Універсальне відношення для бази даних виглядає наступним чином: R (ім’я користувача, пароль, стать, країна, місто, id гри, назва, категорія, опис, id обкладинки, оцінка, дата запису, 100% проходження, переграш, відгук, оцінка запису, дата створення запису, id поста форума, тема поста, текст поста, дата створення посту форума, текст відповіді на пост форума, дата створення відповіді на пост форума, текст поста користувача, дата створення поста користувача, текст відповіді на пост користувача, дата створення відповіді на пост користувача).

Ступінь універсального відношення – 27.

Нормалізація бази даних є критично важливим процесом, що забезпечує ефективне зберігання та обробку даних шляхом усунення надлишковості та уникнення аномалій при оновленні, вставці або видаленні даних. Вона структурує дані в логічно зв’язані таблиці, що сприяє зменшенню дублювання та покращенню цілісності інформації. Завдяки нормалізації забезпечується узгодженість даних, спрощується супровід системи й підвищується її масштабованість, що особливо важливо для великих або динамічних інформаційних систем.

Відповідно до літератури відношення знаходиться в 1НФ, якщо на перетині кожного стовпця та кожного рядка знаходяться лише атомарні значення атрибутів і не містяться групи, що повторюються. Проаналізувавши таблиці, можна побачити, що відношення бази даних знаходяться в першій нормальній формі [21].

Відношення знаходиться в 2НФ, якщо виконуються обмеження 1НФ та кожен неключовий атрибут функціонально повно залежить від первинного ключа. Проаналізувавши розроблену базу даних, можна стверджувати, що відношення бази даних знаходяться у другій нормальній формі [21].

Для приведення відношення до 3НФ необхідно забезпечити виконання вимог другої нормальної форми, а також усунути транзитивні залежності

неключових атрибутів від первинного ключа. Це означає, що кожен неключовий атрибут повинен залежати лише від первинного ключа та не повинен мати залежностей від інших неключових атрибутів. Оскільки в процесі аналізу таблиць розробленої бази даних не було знайдено транзитивних залежностей неключових атрибутів, можна зробити висновок про відповідність відношень вимогам третьої нормальної форми [21].

2.4 Розробка алгоритму роботи програмного модуля для оцінювання та рецензування відеоігор

Для забезпечення повноцінного функціонування програмного модуля, призначеного для оцінювання та рецензування відеоігор, необхідно визначити послідовність дій користувача в межах типових сценаріїв використання. Програмний модуль повинен забезпечувати можливості від реєстрації та авторизації до створення записів, постів, відповідей, а також перегляду журналу та пошуку інших користувачів.

І випадок

Алгоритм у випадку коли користувач не зареєстрований складається з наступних кроків:

Крок 1. Запуск головного вікна;

Крок 2. Відображення форми для реєстрації;

Крок 3. Введення даних у форму;

Крок 4. Реєстрація користувача з заданими даними;

Крок 5. Відображення головного меню.

ІІ випадок

Алгоритм у випадку коли користувач не авторизований складається з наступних кроків:

Крок 1. Запуск головного вікна;

Крок 2. Відображення форми для авторизації;

Крок 3. Введення даних у форму;

Крок 4. Авторизація користувача з заданими даними;

Крок 5. Відображення головного меню.

III випадок

Алгоритм для створення запису у журналі користувача складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває список ігор;

Крок 3. Користувач відкриває сторінку з грою;

Крок 4. Користувач натискає кнопку для створення запису;

Крок 5. Відображення форми для створення запису;

Крок 6. Введення даних у форму;

Крок 7. Створення запису з заданими даними.

IV випадок

Алгоритм для створення посту на форумі складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває список ігор;

Крок 3. Користувач відкриває сторінку з грою;

Крок 4. Користувач відкриває форум гри;

Крок 5. Користувач натискає кнопку для створення посту на форумі;

Крок 6. Відображення форми для створення посту на форумі;

Крок 7. Введення даних у форму;

Крок 8. Створення посту форума з заданими даними.

V випадок

Алгоритм для створення відповіді на пост форума складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває список ігор;

Крок 3. Користувач відкриває сторінку з грою;

Крок 4. Користувач відкриває форум гри;

Крок 5. Користувач відкриває пост форума;

Крок 6. Користувач натискає кнопку для створення відповіді на пост форума;

Крок 7. Відображення форми для створення відповіді на пост форума;

Крок 8. Введення даних у форму;

Крок 9. Створення відповіді на пост форума;

VI випадок

Алгоритм для створення посту на сторінці користувача складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває сторінку з профілем;

Крок 3. Користувач натискає кнопку для створення посту користувача;

Крок 4. Відображення форми для створення посту користувача;

Крок 5. Введення даних у форму;

Крок 6. Створення посту користувача;

VII випадок

Алгоритм для створення відповіді на пост користувача складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває сторінку з профілем;

Крок 3. Користувач відкриває пост користувача;

Крок 4. Користувач натискає кнопку для створення відповіді на пост користувача

Крок 5. Відображення форми для створення відповіді на пост користувача

Крок 6. Введення даних у форму;

Крок 7. Створення відповіді на пост користувача

VIII випадок

Алгоритм для відкриття журналу користувача складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває сторінку з профілем;

Крок 3. Користувач відкриває сторінку з журналом;

IX випадок

Алгоритм для відкриття пошуку користувачів по геолокації складається з наступних кроків:

Крок 1. Користувач відкриває програмний модуль та якщо авторизований, переходить до головної сторінки;

Крок 2. Користувач відкриває сторінку з профілем

Крок 3. Користувач відкриває сторінку пошуку користувачів по геолокації.

На основі визначених сценаріїв взаємодії користувача з системою було розроблено послідовні алгоритми, що охоплюють як базову навігацію в модулі (реєстрація, авторизація), так і ключові функції – створення записів, постів, відповідей, а також перегляд журналу та пошук користувачів. Ці алгоритми

відображають логіку користувацького шляху, забезпечують інтуїтивність у взаємодії з інтерфейсом та створюють фундамент для реалізації функціональних компонентів системи. Алгоритм роботи програмного модуля для оцінювання та рецензування відеоігор наведено на рисунку 2.10

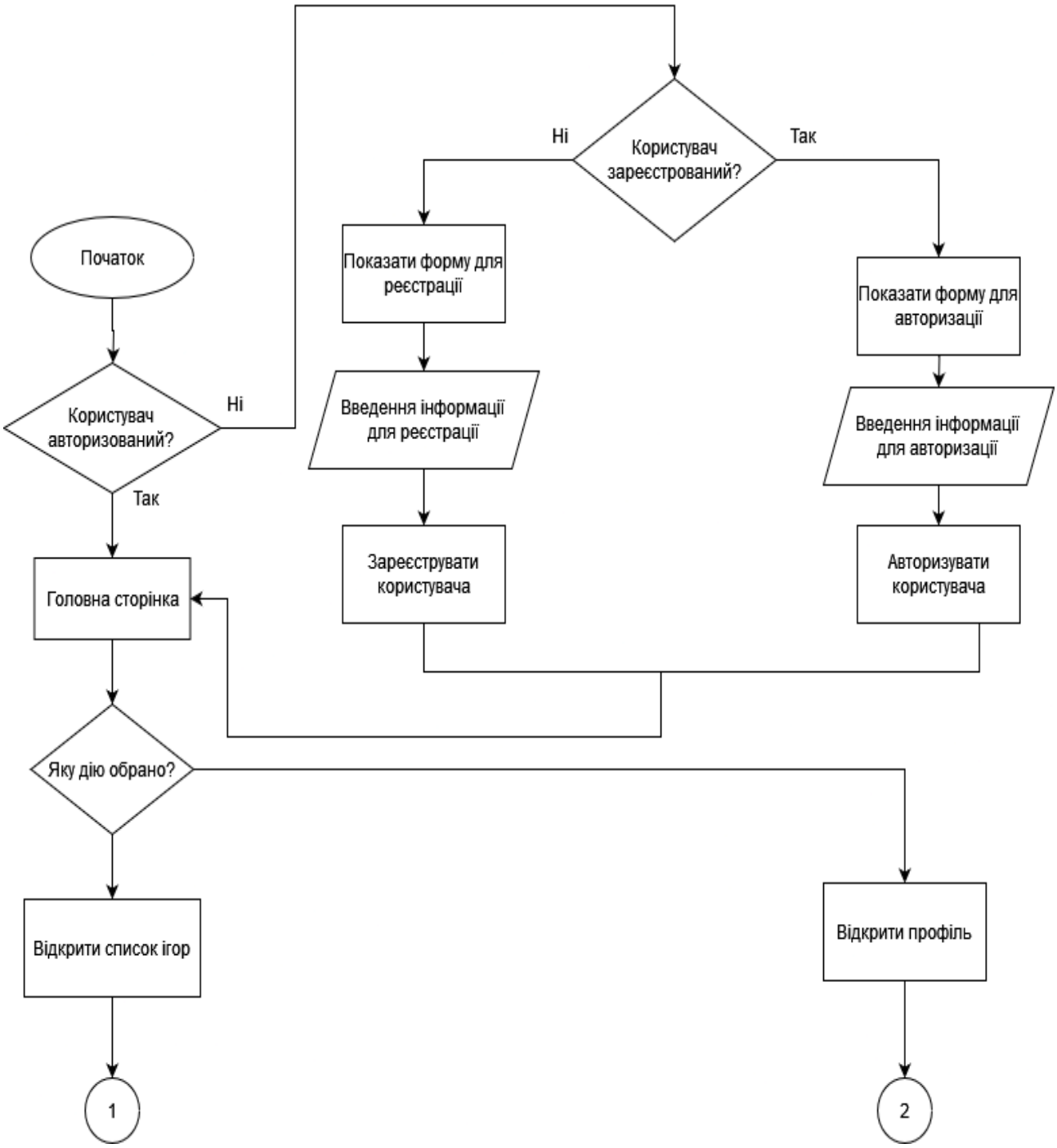

Рисунок 2.10 – Алгоритм роботи програмного модуля для оцінювання та рецензування відеоігор

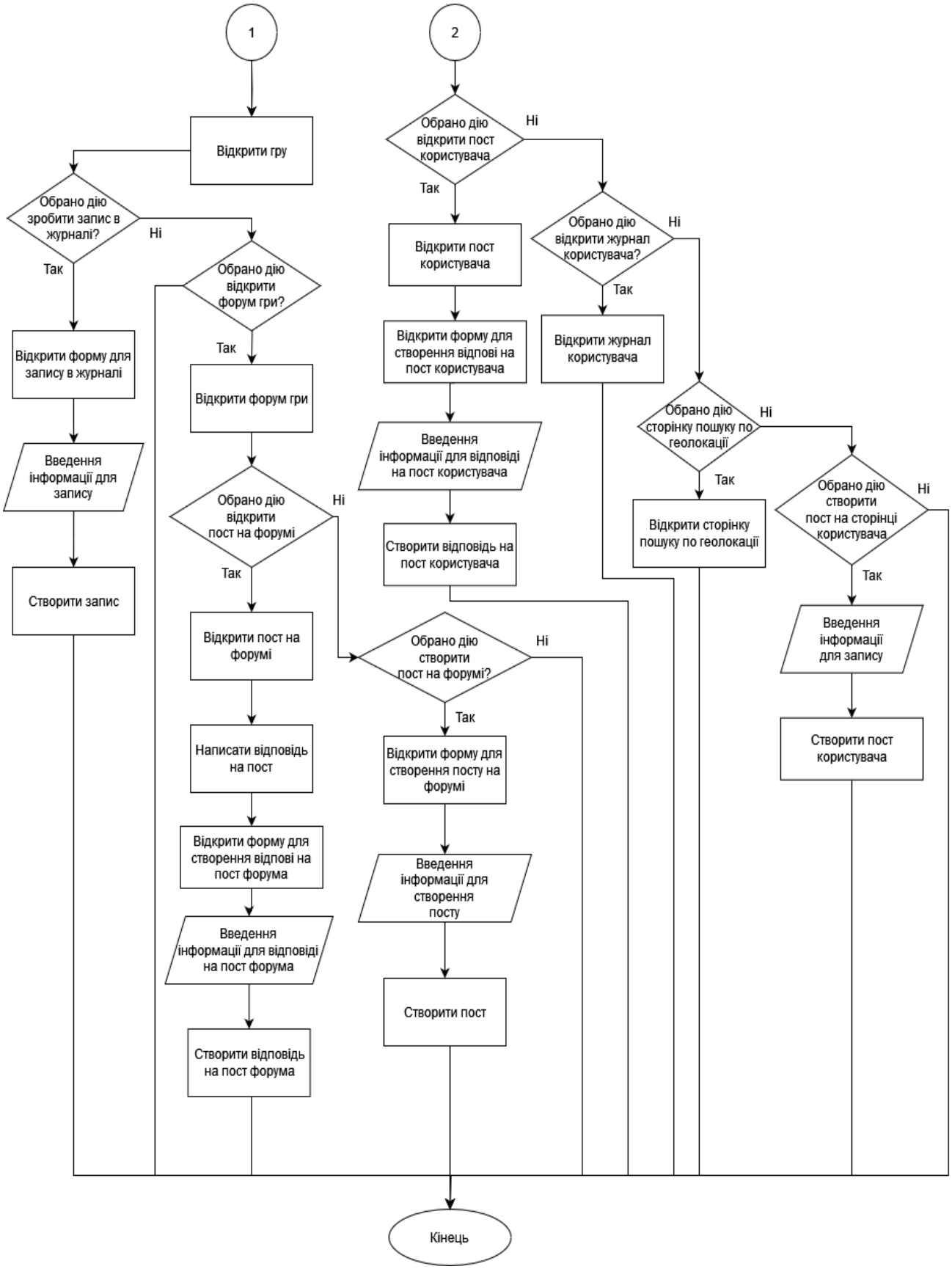

Рисунок 2.10 – Продовження

2.5 Висновок до розділу 2

Здійснено аналіз можливих архітектурних підходів щодо створення програмного модуля оцінювання та рецензування відеоігор, що дозволило обґрунтовано обрати клієнт-серверну архітектуру як найбільш оптимальну для забезпечення масштабованості, зручності підтримки та розподілу функціональності між клієнтською та серверною частинами.

Розроблено діаграми розгортання та прецедентів, що надали змогу візуалізувати загальну структуру системи та взаємодію користувачів з програмним модулем. Розроблена діаграма прецедентів передбачає реалізацію відсутніх в аналогах функцій (створення записів у журналі, створення постів на форумі та відповідей на них, створення постів на сторінках користувачів та відповідей на них, здійснення пошуку користувачів за географічною локацією).

Створено ER-діаграму та схему бази даних програмного модуля оцінювання та рецензування відеоігор, які відображають структуру зберігання даних, у тому числі сутності, їхні атрибути та взаємозв’язки. Розроблена схема бази даних дозволяє реалізувати додаткові вище зазначені функції, що відсутні в аналогах.

Розроблено алгоритм роботи програмного модуля оцінювання та рецензування відеоігор, який визначає послідовність дій під час взаємодії з модулем. Даний алгоритм відрізняється від існуючих наявністю додаткових блоків. Це дозволяє реалізувати такі додаткові функції як: створення записів у журналі, створення постів на форумі та відповідей на них, створення постів на сторінках користувачів та відповідей на них, здійснення пошуку користувачів за географічною локацією.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОГРАМНОГО МОДУЛЯ ОЦІНЮВАННЯ ТА РЕЦЕНЗУВАННЯ ВІДЕОІГОР

3.1 Обґрунтування вибору мови програмування

Під час розробки програмного модуля важливим кроком є вибір мови програмування, яка відповідатиме вимогам проекту щодо функціональності, гнучкості, швидкості розробки та сумісності з іншими технологіями. У межах реалізації проекту було прийнято рішення використовувати клієнт-серверну архітектуру, яка дозволяє незалежно розробляти інтерфейс користувача (клієнтську частину) та серверну логіку. Це дає можливість обрати оптимальні мови програмування для кожного з компонентів, зважаючи на специфіку їхніх задач. Для цього будуть розглянуті такі мови як Python, Java, C# та JavaScript.

Для розробки клієнтської частини розглядалися такі мови, як JavaScript, TypeScript, Dart (через Flutter Web), та Python (з використанням Brython або Pyodide). Попри те, що Python є дуже зручною мовою для швидкого прототипування, його підтримка у веб-браузері залишається обмеженою та значно поступається продуктивністю нативному JavaScript. Варіанти, які дозволяють запускати Python у браузері (Brython, Pyodide), є експериментальними та додають суттєвого навантаження на сторінку через завантаження інтерпретатора та повільну взаємодію з DOM.

Dart у поєднанні з Flutter Web пропонує гарну продуктивність і сучасний підхід до побудови UI, проте має меншу екосистему та вимагає від розробника більше ресурсів для налаштування середовища та оптимізації. Крім того, розмір зібраного JavaScript-коду у Flutter Web часто є значно більшим, що впливає на швидкість завантаження сторінки.

TypeScript є розширенням JavaScript з підтримкою статичної типізації. Хоча його використання додає зручності при масштабних проєктах, у нашому випадку обсяг клієнтської логіки не є настільки складним, щоб обґрунтувати

додаткові витрати часу на конфігурацію TypeScript-оточення. З огляду на це, було обрано JavaScript – як єдину мову, яка виконується у веббраузерах без додаткових обгорток, має величезну екосистему та дозволяє швидко створювати інтерфейси з використанням сучасних фреймворків (наприклад, React чи Vue).

Для серверної частини розглядалися мови Python, JavaScript (Node.js), PHP, Java та Go. PHP, хоча і широко використовується у веб-розробці, має застарілу архітектуру та вимагає більше ресурсів на обробку кожного запиту через модель "запуску з нуля" на кожен HTTP-запит. Крім того, PHP поступається сучасним мовам у плані зручності роботи з асинхронними операціями.

Node.js (JavaScript на сервері) – популярне рішення, особливо коли використовується JavaScript і на клієнті. Проте модель асинхронного програмування через callback-и або проміси іноді ускладнює підтримку складної логіки, а продуктивність Node.js у сценаріях з високим навантаженням часто поступається мовам зі скомпільованим виконанням, таким як Go.

Python – дуже зручна мова, з великою кількістю фреймворків (Flask, Django), але інтерпретована природа мови та відсутність вбудованої багатопоточності у стандартній реалізації (через Global Interpreter Lock) створює певні обмеження у масштабованості без додаткових рішень (наприклад, Gunicorn + Uvicorn + async).

Java – потужна, високопродуктивна мова, але її екосистема є складною для налаштування, а розгортання програм займає більше часу. Java добре підходить для великих корпоративних систем, проте для легшого та швидшого розгортання REST-серверу цей варіант був надлишковим.

У підсумку, Go (Golang) був обраний як основна серверна мова завдяки своїй простоті, високій продуктивності, підтримці конкурентного виконання через легкі горутини, низькому споживанню ресурсів, та простій компіляції в єдиний виконуваний файл. Go добре підходить для побудови високопродуктивних веб-серверів і REST API з мінімальними зовнішніми залежностями, що значно спрощує процес розгортання і супроводу.

3.2 Обгрунтування вибору середовища розробки

Вибір середовища розробки (IDE або редактора коду) є важливим етапом у проєктуванні та реалізації програмного забезпечення, оскільки безпосередньо впливає на ефективність розробника, швидкість роботи з кодом та можливості масштабування проєкту. У процесі порівняння було розглянуто кілька популярних рішень: Visual Studio Code, Atom, Sublime Text та WebStorm.

Atom був створений компанією GitHub як відкритий, кросплатформений редактор коду, орієнтований на максимальну кастомізацію. Завдяки використанню фреймворку Electron, Atom дозволяв розробникам змінювати практично будь-який аспект інтерфейсу та функціональності за допомогою HTML, CSS та JavaScript.

Однією з ключових переваг Atom була його розширюваність. Користувачі мали доступ до великої кількості пакетів, які додавали нові функції, підтримку мов програмування та інтеграцію з різними інструментами.

Проте з часом Atom почав втрачати популярність через проблеми з продуктивністю, особливо при роботі з великими проєктами. Крім того, після придбання GitHub компанією Microsoft, розвиток Atom був зупинений, і з грудня 2022 року офіційна підтримка редактора припинилася [22].

Sublime Text – це високопродуктивний текстовий редактор, відомий своєю швидкістю та стабільністю. Він підтримує велику кількість мов програмування, має функції автодоповнення, підсвічування синтаксису, згортання коду та потужну систему пошуку.

Редактор також відзначається високим рівнем кастомізації, дозволяючи користувачам налаштовувати теми, комбінації клавіш та встановлювати плагіни для розширення функціональності.

Проте Sublime Text є умовно безкоштовним: хоча базова версія доступна безкоштовно, періодично з'являються нагадування про необхідність придбання ліцензії. Крім того, деякі функції, такі як інтеграція з системами контролю версій

чи підтримка сучасних фреймворків, потребують додаткової конфігурації або встановлення плагінів [23].

WebStorm – це комерційне середовище розробки від компанії JetBrains, спеціалізоване на розробці JavaScript, TypeScript та інших вебтехнологій. Воно пропонує широкий спектр функцій "з коробки", включаючи інтелектуальне автодоповнення, рефакторинг, налагодження та інтеграцію з системами контролю версій.

Однією з переваг WebStorm є глибока інтеграція з популярними фреймворками та бібліотеками, такими як React, Angular та Vue.js, що дозволяє ефективно працювати з сучасними вебзастосунками.

Однак WebStorm є платним продуктом, що може бути обмеженням для студентів або розробників, які шукають безкоштовні рішення. Крім того, його функціональність може бути надмірною для невеликих проєктів або початківців [24].

Visual Studio Code (VS Code) – це безкоштовний, кросплатформений редактор коду від Microsoft, який поєднує легкість текстового редактора з функціональністю повноцінного IDE. Він підтримує велику кількість мов програмування, має вбудовану інтеграцію з Git, термінал, налагоджувач та систему розширень.

Завдяки великій спільноті та активному розвитку, VS Code пропонує широкий вибір розширень для підтримки різних фреймворків, бібліотек та інструментів. Його гнучкість дозволяє налаштовувати середовище під конкретні потреби проєкту.

Після детального аналізу популярних середовищ розробки можна зробити висновок, що Visual Studio Code є оптимальним вибором для реалізації сучасних веб-застосунків. Його безкоштовність, кросплатформеність, широка підтримка мов програмування та фреймворків, а також активна спільнота розробників роблять його універсальним інструментом як для початківців, так і для досвідчених програмістів [25].

3.3 Обґрунтування вибору СУБД

У процесі розробки програмного модуля постала необхідність у виборі системи управління базами даних, яка б забезпечувала надійне зберігання інформації, високу продуктивність та можливість подальшого масштабування. Було розглянуто кілька поширених рішень: MySQL, SQLite, Microsoft SQL Server та PostgreSQL.

MySQL є однією з найпопулярніших реляційних СУБД, що відзначається швидкою роботою та широкою підтримкою спільноти. Вона добре підходить для проєктів із типовою структурою даних і невеликою складністю запитів. Проте при зростанні вимог до гнучкості запитів та складності логіки MySQL виявляється менш зручною у використанні.

SQLite — це легковагова СУБД, що зберігає всі дані у вигляді одного файлу і не потребує окремого серверного середовища. Вона оптимальна для невеликих локальних застосунків або прототипів. Її основним обмеженням є недостатня ефективність у багатокористувацьких сценаріях та при значному навантаженні.

Microsoft SQL Server — це функціонально потужна СУБД, що забезпечує високу надійність і гнучкість у роботі з даними. Вона активно використовується в корпоративному середовищі, має зручні засоби адміністрування та розробки. Водночас, її встановлення та початкове налаштування є доволі складним процесом, що потребує більше часу та ресурсів у порівнянні з іншими рішеннями.

За підсумками аналізу було обрано PostgreSQL як оптимальне рішення. Вона є вільною у використанні, підтримує складні запити та структури, добре працює з великими обсягами даних і відповідає вимогам щодо масштабованості та надійності. PostgreSQL поєднує функціональність, гнучкість і стабільність, що робить її доцільним вибором для реалізації даного проєкту.

3.4 Розробка клієнтської та серверної частин

У межах реалізації програмного модуля застосовується архітектура «клієнт-сервер», яка передбачає поділ системи на дві основні компоненти: клієнтську частину (інтерфейс користувача) та серверну частину (логіка обробки даних і взаємодія з базою даних). Такий підхід дозволяє розвивати та тестувати кожен компонент окремо, забезпечуючи гнучкість, масштабованість і зручність супроводу.

Клієнтська частина реалізується мовою JavaScript і відповідає за динамічну взаємодію з користувачем, а також за надсилання HTTP-запитів до серверної частини. Серверна частина, реалізована мовою Go, приймає запити, виконує бізнес-логіку та здійснює доступ до бази даних.

Для зберігання даних використовується система керування базами даних PostgreSQL. З метою спрощення розгортання та ізоляції бази даних від решти системи, вона запускається у вигляді окремого Docker-контейнера. Такий підхід дозволяє легко налаштувати середовище бази даних, уникати конфліктів із вже встановленими СКБД на хості та спрощує процес розгортання на різних машинах. Крім того, використання контейнера полегшує резервне копіювання, відновлення даних та масштабування в майбутньому.

Серверна частина застосунку взаємодіє з контейнером PostgreSQL через вказані параметри з'єднання (хост, порт, користувач, пароль і назва бази), що конфігуруються у зовнішньому файлі або через змінні середовища. Така модель дозволяє зберігати гнучкість у конфігурації й підтримувати чисту архітектуру проєкту.

Для розробки клієнтської частини застосунку було обрано React – одну з найпопулярніших бібліотек JavaScript для побудови користувацьких інтерфейсів. React забезпечує компонентну архітектуру, що дозволяє розділити інтерфейс на незалежні, повторно використовувані частини (компоненти), кожна з яких управляє власним станом. Завдяки віртуальному DOM, React ефективно оновлює

лише ті частини сторінки, які зазнали змін, що покращує продуктивність інтерфейсу.

Окрім основного React, у проєкті використовується React Router – офіційна бібліотека для організації маршрутизації в React-застосунках. Вона дозволяє створювати багатосторінкові інтерфейси без необхідності перезавантаження сторінки (SPA – Single Page Application). React Router керує URL-адресами та відображає відповідні компоненти залежно від поточного шляху, забезпечуючи інтуїтивну навігацію для користувача.

Нижче наведено приклад базової конфігурації маршрутизатора для програмного модуля, який визначає основні маршрути застосунку, включаючи сторінки гри, профілю, форуму, журналу користувача, пошуку та автентифікації:

```
createRoot(document.querySelector('body')!).render(
  <StrictMode>
    <BrowserRouter>
      <Routes>
        <Route element={<Layout />} >
          <Route path='/' element={<App />} />
          <Route path='/game/:id' element={<Game />} />
          <Route path='/game/:id/forum' element={<Forum />} />
          <Route path='/games' element={<Games />} />
          <Route path='/forum/:id' element={<ForumPost />} />
          <Route path='/profile/:username' element={<Profile
key={location.pathname}/>} />
          <Route path='/profile/:username/journal' element={<Journal />} />
          <Route path='/profile/post/:id' element={<UserPost />} />
          <Route path='/geo_search' element={<GeoSearch />} />
        </Route>
        <Route path='/sign_in' element={<SignIn />} />
        <Route path='/sign_up' element={<SignUp />} />
```

```
      </Routes>
    </BrowserRouter>
    <Toaster position='bottom-right' />
  </StrictMode>
)
```

У реалізації серверної частини програмного модуля було використано Gin – високопродуктивний, мінімалістичний веб-фреймворк для мови Go. Gin забезпечує зручний API для побудови RESTful-застосунків, дозволяючи легко обробляти HTTP-запити, працювати з маршрутами, middleware та JSON-даними. Його синтаксис інтуїтивно зрозумілий, а внутрішня архітектура оптимізована для максимальної швидкодії, що робить його особливо ефективним для вебсервісів, які обробляють велику кількість одночасних запитів.

Однією з ключових переваг Gin є підтримка middleware – функцій, які виконуються на кожному запиті до певного маршруту або до всієї групи маршрутів. Це дозволяє централізовано реалізувати функціональність на кшталт логування, авторизації, обробки помилок тощо. Наприклад, у цьому проєкті middleware використовується для перевірки дійсності JWT-токенів, що забезпечує захист приватних маршрутів.

Gin також надає вбудовану підтримку для автоматичного парсингу JSON-тіл запитів та форм, обробки параметрів маршруту, генерації відповідей у форматі JSON, роблячи розробку серверного API швидкою та ефективною. Завдяки цим можливостям, розробник може зосередитися на реалізації бізнес-логіки, не витрачаючи надмірних зусиль на ручну обробку запитів.

У порівнянні з більш "важкими" фреймворками (на зразок Echo або Revel), Gin зберігає мінімалістичність і простоту, залишаючись водночас достатньо гнучким для складних вебзастосунків. Його активна спільнота та добре задокументований інтерфейс також сприяють зручності використання на практиці.

Для захисту маршрутизованих ресурсів у серверній частині застосунку реалізовано систему аутентифікації на основі JSON Web Token (JWT). Такий підхід дозволяє без використання стану сесій зберігати дані про автентифікованого користувача на стороні клієнта у вигляді токена, який автоматично надсилається у запитах до захищених маршрутів. JWT зручний для RESTful-архітектури та має широке поширення у сучасній веб-розробці.

Серверна частина застосунку реалізує набір RESTful endpoint'ів, які забезпечують основний функціонал платформи: аутентифікацію, роботу з ігровими рецензіями, журналами, форумом та користувацькими даними. Всі маршрути поділяються на публічні та захищені – останні потребують авторизації за допомогою JWT. Зокрема, маршрути /sign-up та /sign-in дозволяють користувачеві зареєструватися та увійти до системи. Захищені POST-запити, як-от /post_review або /post_forum_post, відповідають за створення нового контенту – рецензій, дописів у форумі, відповідей тощо. GET-запити, такі як /fetch_game, /fetch_reviews, /fetch_user, забезпечують отримання інформації з бази даних: даних про ігри, користувачів, записи в журналі або пости. Окрему роль виконують допоміжні маршрути, наприклад, /fetch_users_by_geo для пошуку користувачів за місцем проживання або /fetch_average_rating для обчислення середньої оцінки гри. Така структура забезпечує масштабованість і логічне розділення функціоналу між клієнтською та серверною частинами. Загалом існують наступні точки

Токени створюються у функції GenerateJWT, де вказуються стандартні поля (claims): ім’я користувача, дата створення (iat), термін дії (exp) та ідентифікатор видавника (iss). Токен підписується за допомогою алгоритму HMAC-SHA256 і секретного ключа, що гарантує неможливість його підробки без знання цього ключа. Код функції GenerateJWT виглядає наступним чином:

```
func GenerateJWT(username string) (string, error) {
   claims := jwt.MapClaims{
      "username": username,
```

```
        "exp": time.Now().Add(1 * time.Hour).Unix(),
        "iat": time.Now().Unix(),
        "iss": "loggd",
    }

    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

    signedToken, err := token.SignedString(jwtSecret)
    if err != nil {
        return "", err
    }

    return signedToken, nil
}
```

Для перевірки автентичності токена використовується функція VerifyJWT, яка перевіряє правильність підпису, термін дії, а також відповідність поля iss заданому значенню. Крім того, реалізовано middleware-функція AuthMiddleware для вебфреймворку Gin, яке автоматично перевіряє наявність дійсного JWT-токена в cookie з назвою auth. У випадку успішної перевірки, в контекст запиту додається ім’я користувача, що дає змогу іншим обробникам отримати доступ до автентифікованої інформації.

Реєстрація нового користувача здійснюється у функції handleSignUp. Після базової валідації введених даних (мінімальна довжина логіна та пароля, відсутність порожніх значень) пароль хешується з використанням bcrypt – сучасного алгоритму хешування, рекомендованого для зберігання паролів. Отриманий хеш зберігається в базі даних разом із іншими атрибутами користувача. Завдяки цьому, навіть у разі витоку бази, зловмисники не матимуть доступу до реальних паролів. Функція handleSignUp виглядає наступним чином:

```
func handleSignUp(c *gin.Context) {
```

```
	var requestBody SignUpBody

	if err := c.BindJSON(&requestBody); err != nil {
		c.Status(http.StatusBadRequest)
	}

	if len(requestBody.Username) < 3 || len(requestBody.Password) < 6 {
		c.JSON(400, gin.H{
			"message": "username or password doesn't match needed length",
		})
	}

	if requestBody.Username == "" || requestBody.Password == "" {
		c.JSON(400, gin.H{
			"message": "username or password is empty",
		})
	}
	fmt.Printf("username: %s; password: %s; country: %s; city: %s\n",
requestBody.Username, requestBody.Password, requestBody.Country,
requestBody.City)
	hashedPassword, err :=
bcrypt.GenerateFromPassword([]byte(requestBody.Password), bcrypt.DefaultCost)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error in hashing password",
		})
	}
```

```
	_, err = db.Exec("INSERT INTO users (username, password, gender, country, city) VALUES ($1, $2, $3, $4)", requestBody.Username, hashedPassword, requestBody.Gender, requestBody.Country, requestBody.City)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error inserting a user in database",
		})
	}
}
```

Для побудови візуального інтерфейсу клієнтської частини було обрано компонентну бібліотеку shadcn/ui, яка поєднує у собі гнучкість та естетику Tailwind CSS з готовими, стилізованими компонентами інтерфейсу. Shadcn/ui є сучасним рішенням для React-проєктів і базується на бібліотеці Radix UI, забезпечуючи високий рівень доступності та кросбраузерної сумісності.

Використання готових компонентів у вебзастосунках – це не лише спосіб прискорити розробку, а й запорука уніфікованого вигляду застосунку, що відповідає сучасним дизайнерським стандартам. Компоненти бібліотеки shadcn легко стилізуються, підтримують тему світлого і темного інтерфейсу, та дозволяють розробникам фокусуватися на логіці, а не на деталях верстки. Зокрема, інтеграція таких елементів як діалогові вікна, селектори, чекбокси та календарі – істотно спрощується завдяки їхній модульній структурі.

Одним із функціональних елементів клієнтської частини є можливість створити запис у журналі гри (log), який відкривається у вигляді модального вікна. Нижче наведено приклад інтеграції діалогу з бібліотеки shadcn/ui для цієї задачі. Діалог відкривається при натисканні кнопки Log, що обгорнута у компонент DialogTrigger, а сама форма знаходиться всередині LogDialog.

```
<Dialog>
  <DialogTrigger>
    <div>Log</div>
```

```
        </DialogTrigger>
        <LogDialog gameName={game.name} gameID={params.id as string}></LogDialog>
        </Dialog>
```

Компонент LogDialog реалізує форму для створення нового запису гри. Він містить такі елементи:

- Заголовок і опис діалогу, які інформують користувача про контекст запису (назва гри).
- Календар вибору дати, реалізований через Popover з Calendar, що дозволяє встановити дату проходження гри.
- Оцінка гри від 0 до 10, яка реалізована через Select з динамічним оновленням значення.
- Додаткові прапорці: "100%" (master) та "Replay" (replay), що дозволяють позначити рівень проходження гри.
- Текстове поле для рецензії, в якому користувач може залишити враження від гри.
- Кнопка відправки, яка знаходиться в DialogFooter і надсилає форму на сервер для збереження.

У підсумку, було обгрунтовано вибір бібліотек таких як React, Gin та shadcn/ui та описано розробку основних компонентів програми.

3.5 Тестування програмного модуля для оцінювання та рецензування відеоігор

Тестування програмного модуля для оцінювання та рецензування відеоігор є важливим етапом перевірки його працездатності, стабільності та відповідності функціональним вимогам.

Для реєстрації користувачу потрібно вказати логін, пароль, гендер та за бажанням країну та місто проживання користувача. Для входу в систему

потрібно лише логін та пароль. Форма для реєстрації зображено на рисунку 3.4, а для входу на рисунку 3.5.

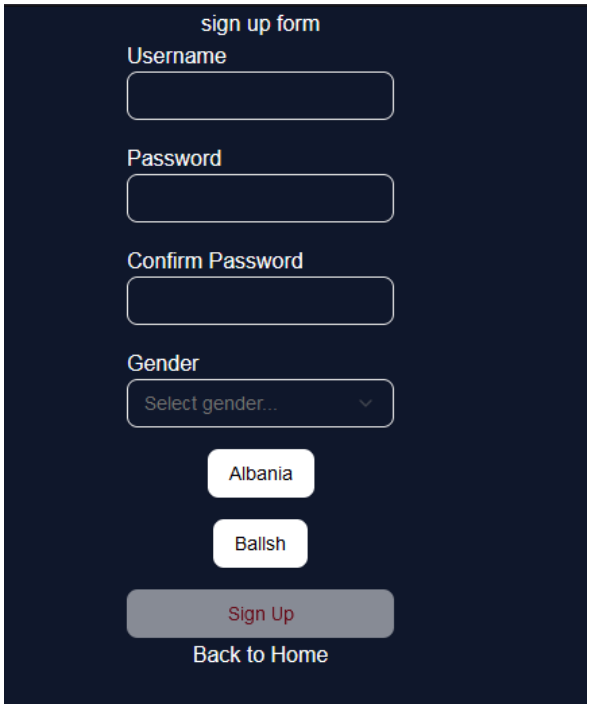

Рисунок 3.4 – Форма для реєстрації користувача

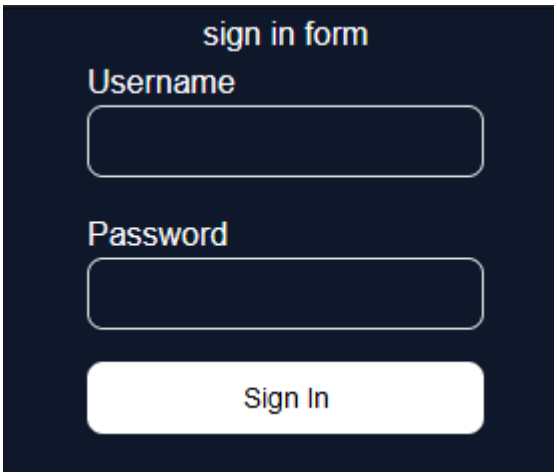

Рисунок 3.5 – Форма для аутентифікації користувача

Після входу в систему відображається головне вікно програмного модуля де користувач може натиснути кнопку Browse для того щоб відкрити список ігор з якими хоче взаємодіяти користувач. Головне вікно зображено на рисунку 3.6 та вікно вибору ігор зображено на рисунку 3.7.

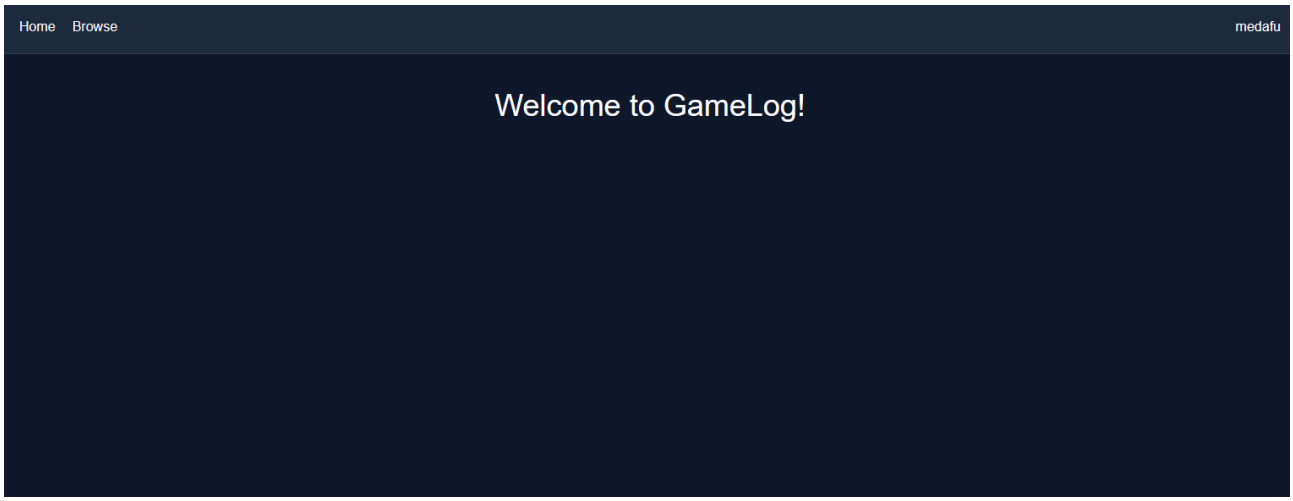

Рисунок 3.6 – Головне меню програмного модуля для оцінювання та рецензування відеоігор

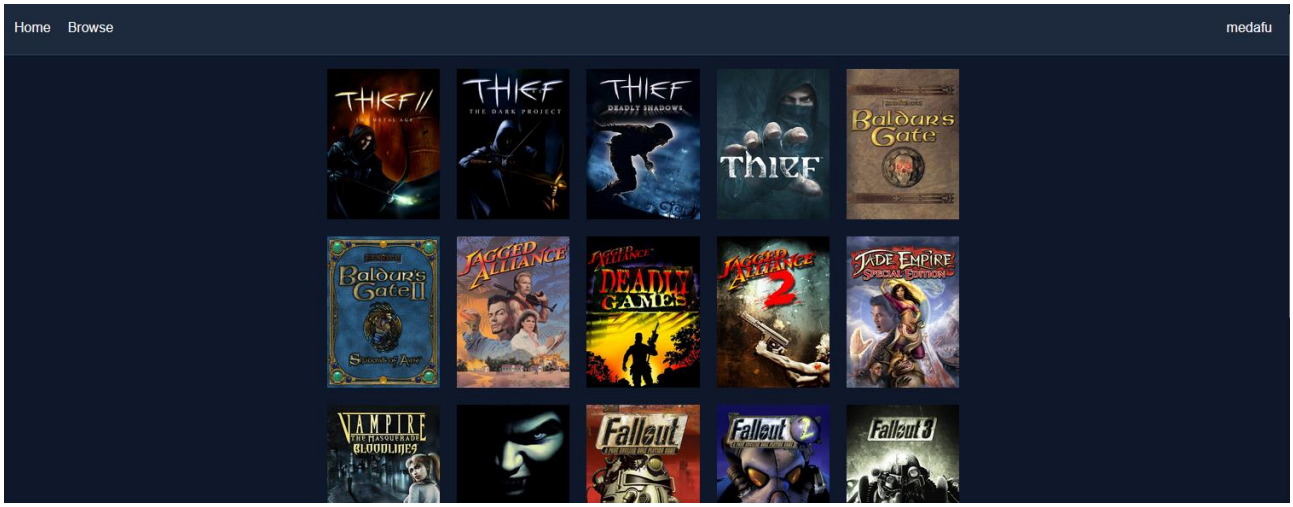

Рисунок 3.7 – Вікно вибору ігор

Після обрання гри ми потрапляємо на сторінку гри де можемо побачити середній рейтинг гри, відгуки, перейти на форум гри або створити запис для цієї гри. Сторінку гри зображено на рисунку 3.8

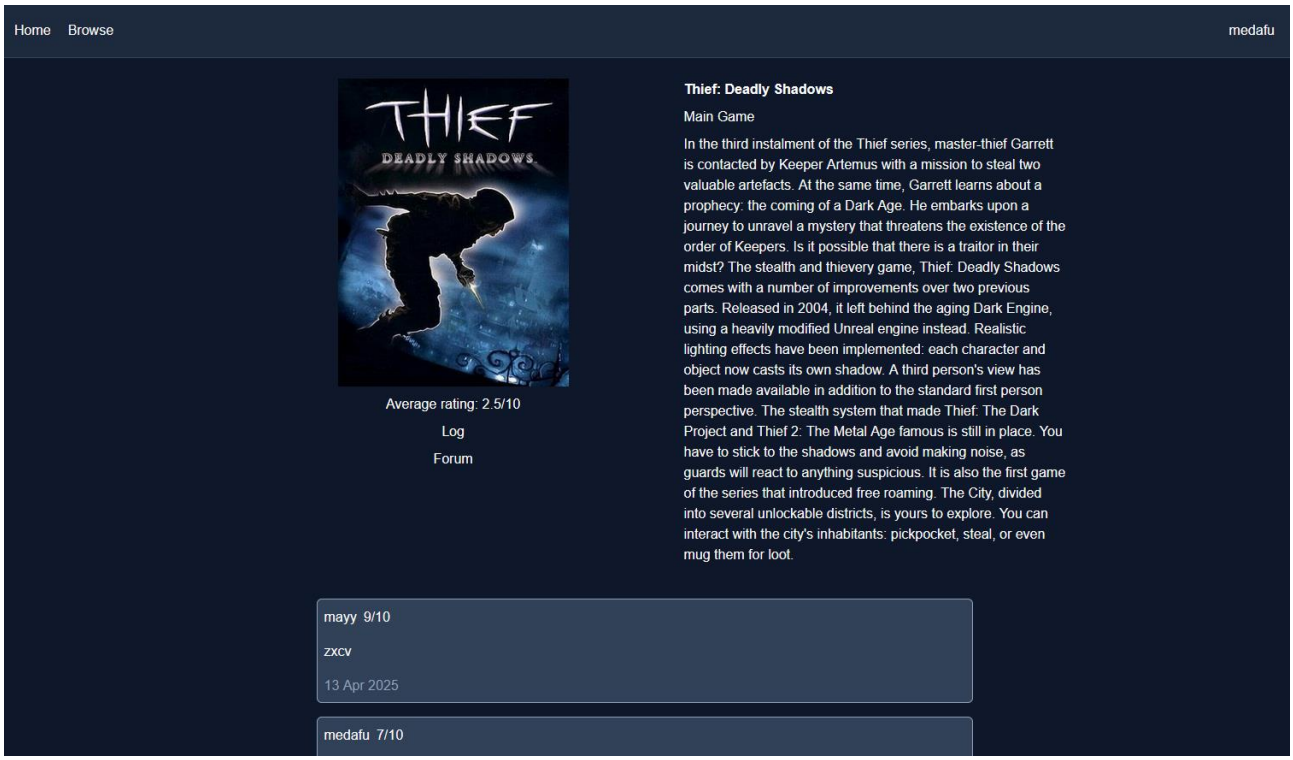

Рисунок 3.8 – Сторінка гри

Якщо було обрано створити запис то з’явиться діалог для створення запису, де користувач повинен ввести дату, оцінку, поставити прапорці чи була гра пройдена на 100 відсотків і чи це повторне проходження та за бажанням написати відгук. Форма для створення запису зображена на рисунку 3.9.

Рисунок 3.9 – Форма для створення запису

Якщо було обрано перейти на сторінку форуму гри то нас зустріне вибір зі створених тем. Також є форма для створення нового посту на форумі де користувачу потрібно вказати заголовок та текст поста. Сторінка з форумом гри зображена на рисунку 3.10

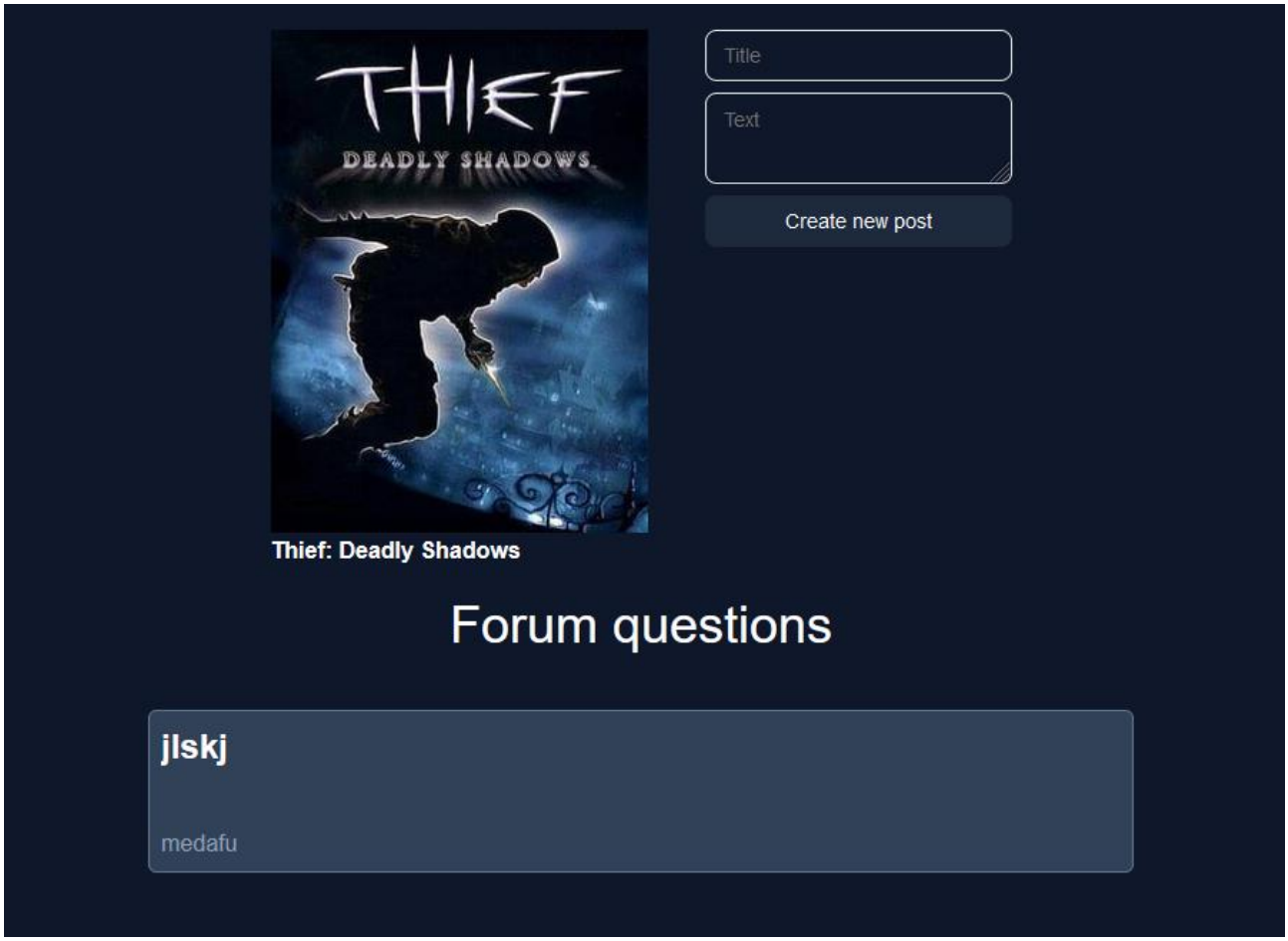

Рисунок 3.10 – Сторінка форуму гри

Користувач може відкрити конкретний пост на форумі і побачити сторінку цього поста. На ній є відповіді інших користувачів до теми на форумі, також можна залишити свою відповідь заповнивши текстове поле внизу. Сторінку з постом на форумі зображено на рисунку 3.11.

Повертаючись на головну сторінку програмного модуля ми можемо зайти на сторінку свого профілю та побачити інформацію про свій профіль. На сторінці профілю також можна побачити перелік постів користувача, перейти на сторінку

с журналом користувача, форму для створення посту користувача. Сторінку з профілем користувача зображено на рисунку 3.12.

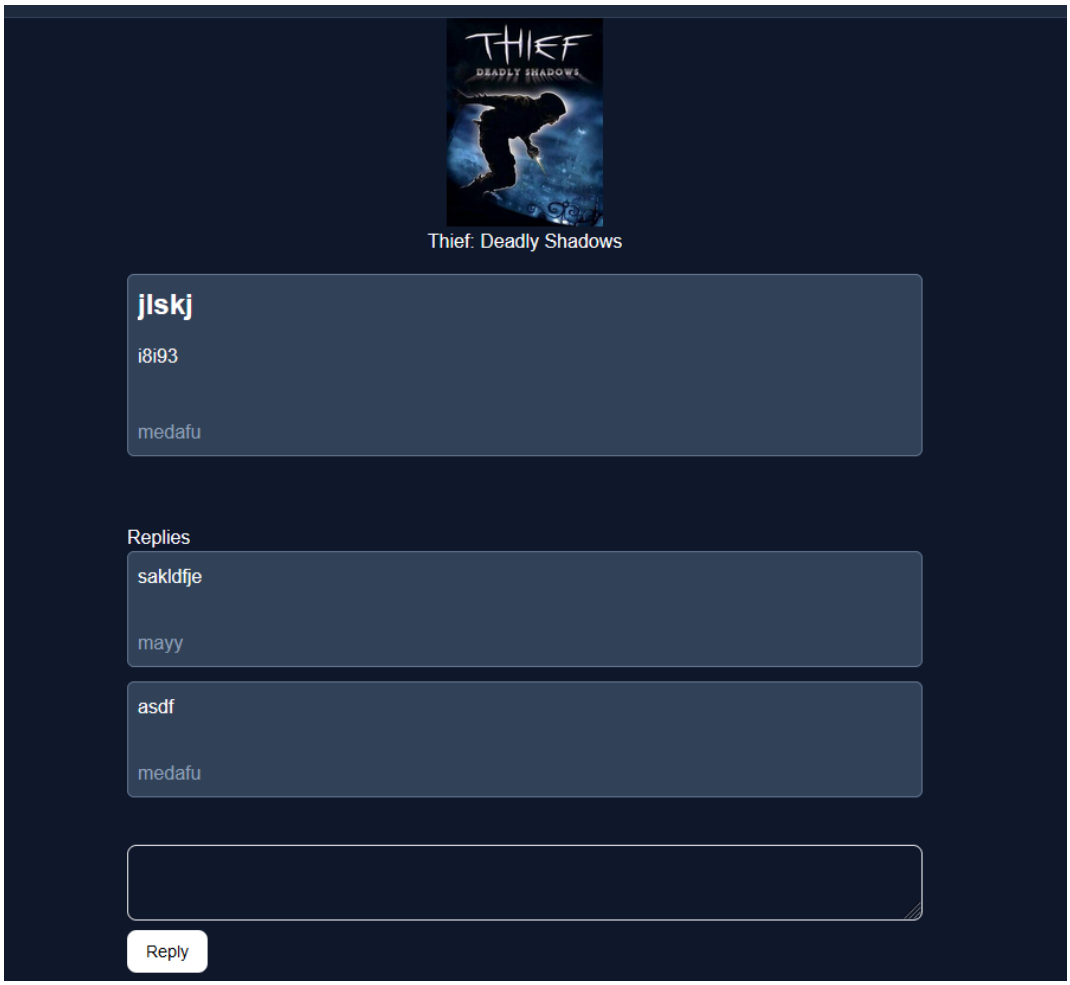

Рисунок 3.11 – Сторінка поста на форумі

Якщо користувач обере пошук інших користувачів за країном, та опціонально за містом, то буде відкрита сторінка зі списком користувачів. Сторінка з пошуком користувачів за країною показана на рисунку 3.13, а сторінку з пошуком по країні та місту на рисунку 3.14.

Якщо користувач обере відкрити пост на сторінці користувача, то він побачить повний текст посту та відповіді на пост, також користувач може використати текстове поле щоб створити відповідь. Сторінку з постом на сторінці користувача зображено на рис 3.15.

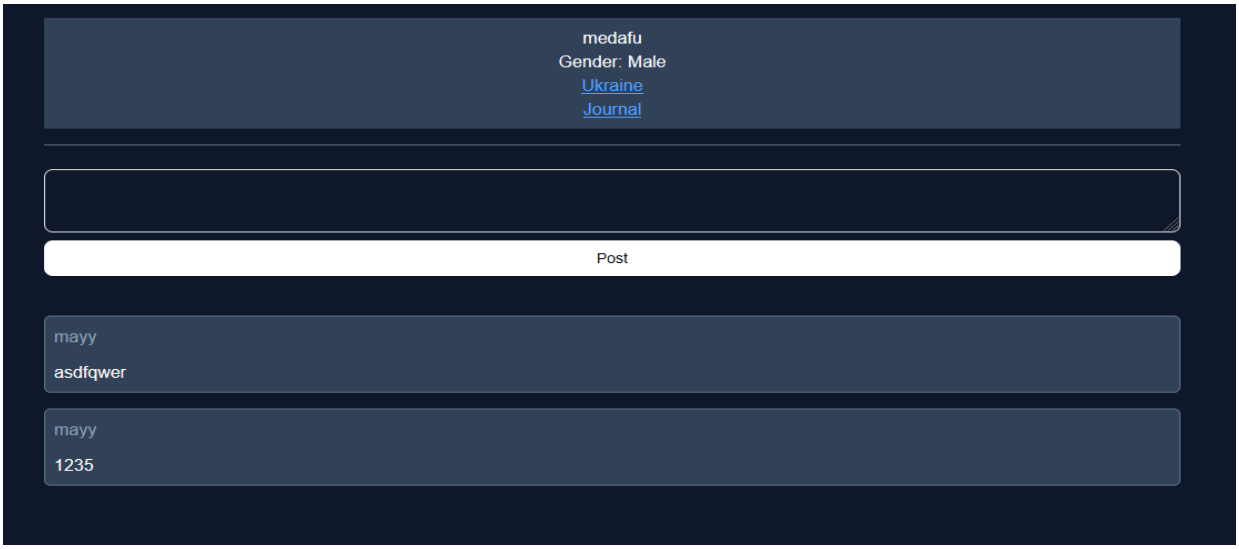

Рисунок 3.12 – Профіль користувача

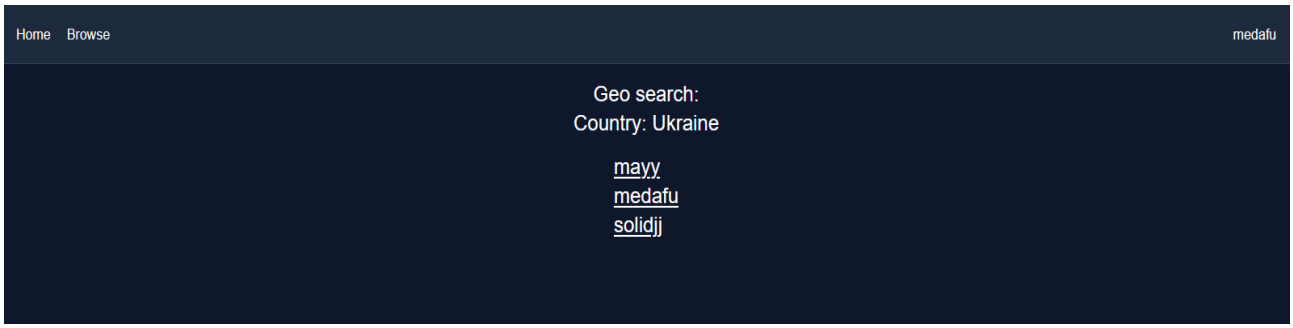

Рисунок 3.13 – Сторінка з пошуком користувачів за країною

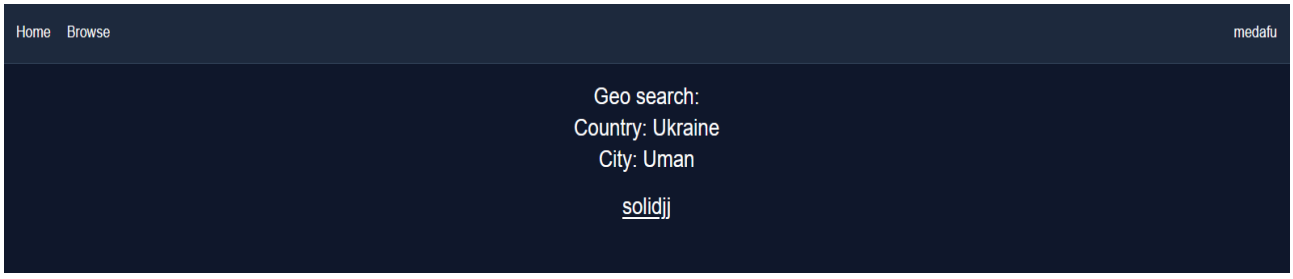

Рисунок 3.14 – Сторінка з пошуком користувачів за країною та містом

Якщо користувач зайде на сторінку журналу користувача то він побачить записи користувача. Сторінка журналу користувача зображено на рис. 3.16.

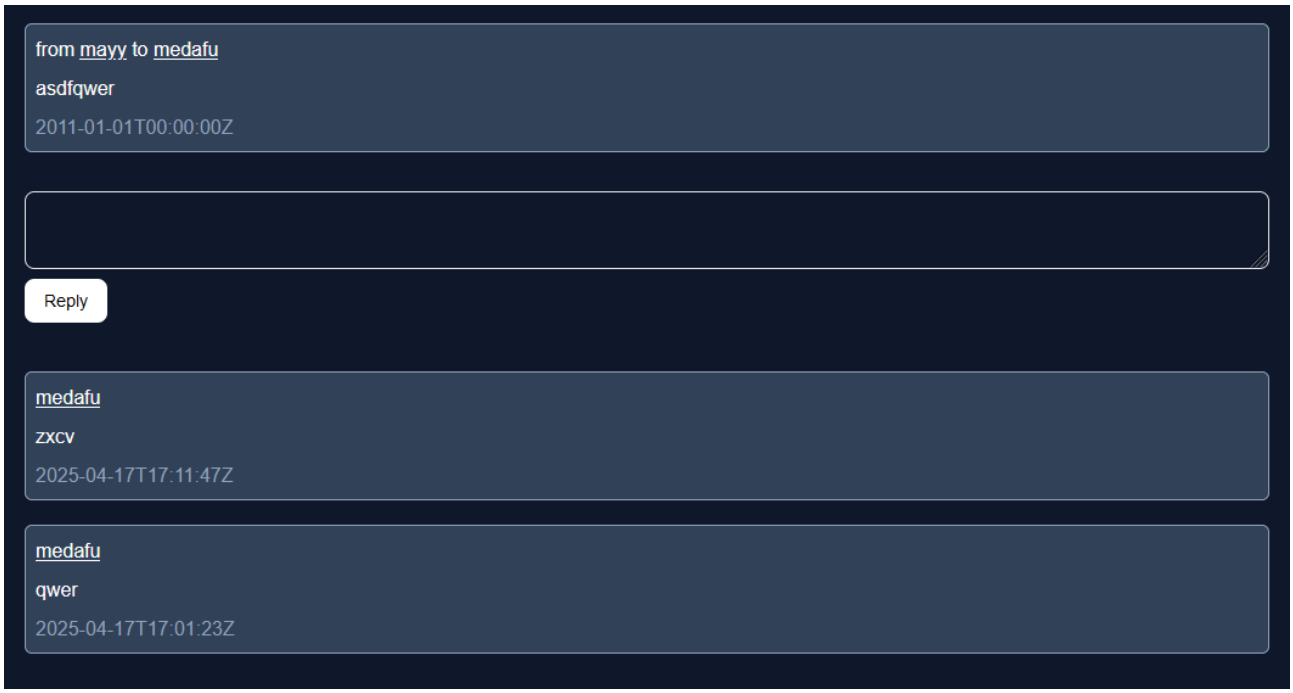

Рисунок 3.15 – Сторінка посту на сторінці користувача

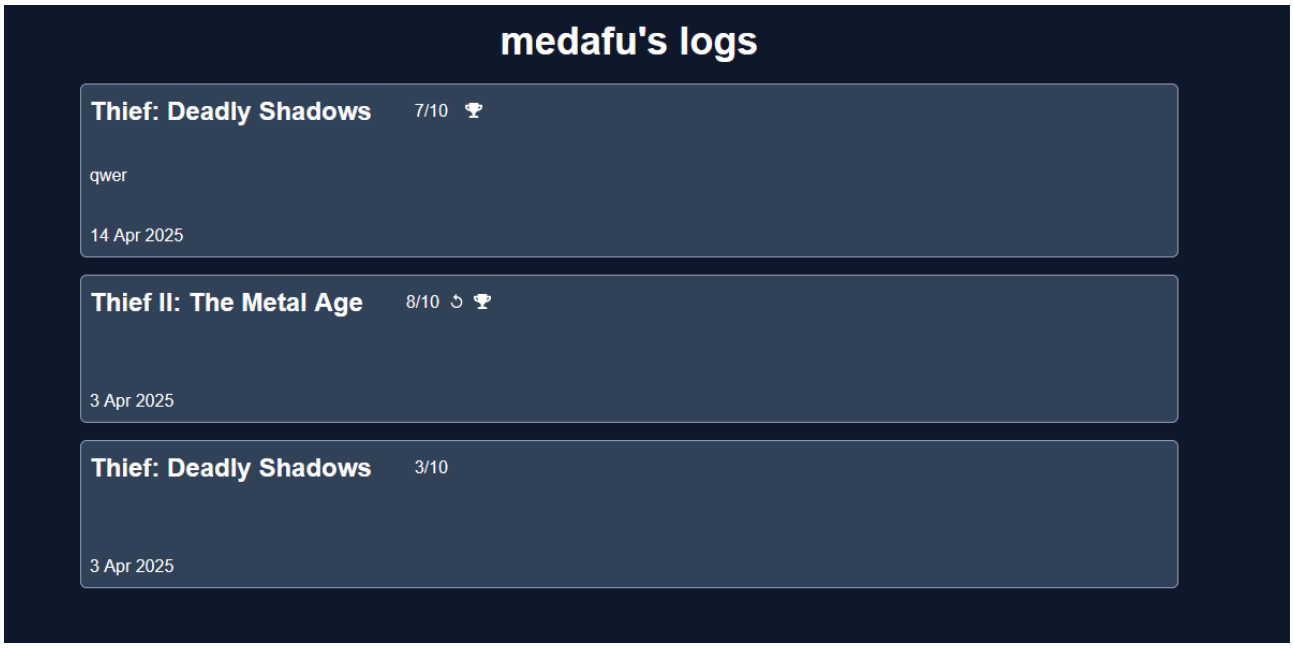

Рисунок 3.16 – Сторінка журналу користувача

У процесі тестування програмного модуля було створено 5 користувачів кожен з яких створив декілька записів у журналі різним іграм, було створено 3 поста на форумі, декілька відповідей на пост форума, 3 поста на сторінках користувачів та декілька відповідей на пости на сторінці користувачів,

перевірено пошук користувачів по геолокації з різними налаштуваннями країни та міста.

3.5 Висновок до розділу 3

Обґрунтовано вибір мов програмування для реалізації клієнтської та серверної частин програмного модуля. Проаналізовано переваги та недоліки популярних мов програмування для кожної частини архітектури, після чого вибір було зроблено на користь JavaScript для клієнта та Go для серверної логіки завдяки їхній продуктивності, зручності та активній підтримці спільноти.

Обґрунтовано вибір середовища розробки. Було порівняно чотири популярні редактори коду: Visual Studio Code, Atom, Sublime Text та WebStorm. Зважаючи на широку підтримку розширень, інтеграцію з системами керування версіями, зручність роботи з JavaScript та Go, а також високу продуктивність, було обрано Visual Studio Code як основне середовище розробки.

Розроблено клієнтську та серверну частини застосунку. Обґрунтовано вибір бібліотек та фреймворків React, React Router та shadcn/ui для клієнта, а також Gin та JWT для сервера. Розроблено програмний код для взаємодії між клієнтом, сервером і базою даних PostgreSQL. Програмний модуль передбачає виконання розширених функцій таких як можливість створення записів у журналі, створення постів на сторінці форума гри або відповіді на них, створення постів на сторінці користувача або відповіді на них, пошуку користувачів за вказаним країною та/або містом.

Здійснено тестування програмного модуля, що дозволило переконатися в його коректній роботі, перевірити обробку запитів, функціональність інтерфейсу, правильність збереження та отримання даних. Також було успішно протестована реалізацію додаткових функцій. Результати тестування підтвердили працездатність розробленого програмного модуля оцінювання та рецензування відеоігор.

ВИСНОВКИ

Поставлені перед бакалаврською кваліфікаційною роботою задачі виконано в повному обсязі, а саме [26]:

1. Аналіз предметної області оцінювання та рецензування відеоігор.
2. Огляд та порівняльний аналіз програмних засобів оцінювання та рецензування відеоігор.
3. Розробка UML-діаграм розгортання та прецедентів програмного модуля.
4. Розробка загального алгоритму роботи програмного модуля.
5. Програмна реалізація та тестування модуля оцінювання та рецензування відеоігор.
6. Розробка інструкції користувача.

Здійснено аналіз предметної області, що дозволило обґрунтувати доцільність створення програмного модуля оцінювання та рецензування відеоігор.

Огляд та порівняльний аналіз популярних аналогів, таких як Backloggd, OpenCritic та інші, показав відсутність єдиної платформи, яка б поєднувала всі необхідні функції для повноцінної взаємодії користувачів навколо тематики відеоігор. Зокрема, жоден із розглянутих сервісів не забезпечує одночасно можливості створення персонального профілю, публікації власних рецензій, ведення журналу проходжень ігор, участі в обговореннях на форумі та здійснення пошуку інших користувачів за географічним розташуванням.

Розроблено UML-діаграму прецедентів, що описують основні сценарії взаємодії користувача з системою та UML-діаграму розгортання програмного модуля, яка відображає архітектуру клієнт-серверної взаємодії. Розроблена діаграма прецедентів має блоки що реалізують усі необхідні функції.

Розроблено загальний алгоритм роботи програмного модуля. Даний алгоритм відрізняється від існуючих наявністю додаткових блоків. Це дозволяє

реалізувати такі додаткові функції як: створення записів у журналі, створення постів на форумі та відповідей на них, створення постів на сторінках користувачів та відповідей на них, здійснення пошуку користувачів за географічною локацією.

Здійснено програмну реалізацію модуля для платформи оцінювання та рецензування відеоігор. Обґрунтовано вибір технічних засобів шляхом порівняльного аналізу мов програмування, середовищ і бібліотек. Клієнтську частину реалізовано на JavaScript з використанням React і бібліотеки shadcn/ui, серверну – на Go з фреймворком Gin. Розробка здійснювалася у Visual Studio Code. У результаті реалізовано всі основні компоненти функціональності: реєстрацію та авторизацію, створення рецензій, дописів на форумі, журналів проходження тощо.

Здійснено тестування програмного модуля, що довело коректну роботу усіх його процесів, а саме авторизації та реєстрації, створення записів у журналі, створення постів на форумі, створення відповіді на пост форума, створення постів на сторінці користувача, створення відповіді на пост користувача, пошук користувачів за географічним розташуванням.

Отже, всі поставлені задачі було виконано, а мету роботи досягнуто.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Лікнарович В. В. Обґрунтування доцільності створення програмного модуля для оці-нювання та рецензування відеоігор/ В. В. Лікнарович, І. Р. Арсенюк // Молодь в науці: дослідження, проблеми, перспективи (МН-2025), 15.10.24 – 14.06.25 : збірник матеріалів. – Вінниця: ВНТУ, 2025. – URL: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/viewFile/24695/20414

2. Gaming Industry Report 2025: Market Size & Trends [Електронний ресурс]. – Режим доступу: https://www.blog.udonis.co/mobile-marketing/mobile-games/gaming-industry

3. Gamer Demographics: 2024 Game-Changing Statistics Worth Checking [Електронний ресурс]. – Режим доступу: https://playtoday.co/blog/stats/gamer-demographics/

4. Metacritic [Електронний ресурс] – Режим доступу: https://www.metacritic.com/

5. OpenCritic [Електронний ресурс] – Режим доступу: https://opencritic.com/

6. HowLongToBeat [Електронний ресурс] – Режим доступу: https://howlongtobeat.com/

7. Steam [Електронний ресурс] – Режим доступу: https://howlongtobeat.com/

8. Epic Games Store [Електронний ресурс] – Режим доступу: https://howlongtobeat.com/

9. GOG Galaxy [Електронний ресурс] – Режим доступу: https://www.gog.com/galaxy

10. Discord [Електронний ресурс] – Режим доступу: https://discord.com/

11. PlayStation Network [Електронний ресурс] – Режим доступу: https://www.playstation.com/en-us/playstation-network/

12. Xbox Live [Електронний ресурс] – Режим доступу: https://www.xbox.com/en-US/live

13. RAWG [Електронний ресурс] – Режим доступу: https://rawg.io/

14.IGN [Електронний ресурс] – Режим доступу: https://www.ign.com/

15.What Are The 10 Most Common Software Architecture Patterns? [Електронний доступ] – Режим доступу: https://medium.com/@the_nick_morgan/what-are-the-10-most-common-software-architecture-patterns-faa4b26e8808

16.Microservices vs monolith: Which architecture is the best choice for your business? [Електронний доступ] – Режим доступу: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/

17.Про архітектуру додатків [Електронний доступ] – Режим доступу: https://foxminded.ua/arkhitektura-zastosunku/

18.What Is Microservices Architecture? (+The Why and How) [Електронний доступ] – Режим доступу: https://learn.g2.com/microservices-architecture

19.MVC Architecture - System Design [Електронний доступ] – Режим доступу: https://www.geeksforgeeks.org/mvc-architecture-system-design/

20.Client-Server Architecture - System Design [Електронний доступ] – Режим доступу: https://www.geeksforgeeks.org/client-server-architecture-system-design/

21.Нормалізація схем баз даних [Електронний доступ] – Режим доступу: https://web.posibnyky.vntu.edu.ua/fitki/10savchuk_organizaciya_bazdanih_znan/gl_43.html

22.Atom https://atom-editor.cc/

23.Sublime Text [Електронний доступ] – Режим доступу: https://www.sublimetext.com/

24.WebStorm [Електронний доступ] – Режим доступу: https://www.jetbrains.com/webstorm/

25.Visual Studio Code [Електронний доступ] – Режим доступу: https://code.visualstudio.com/download

26.Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 122 «Комп'ютерні науки» [Електронний ресурс] / уклад.: А.А. Яровий, О.В. Сілагін, І.В. Богач. Вінниця: ВНТУ 2023.(PDF,54с.).URL:https://iq.vntu.edu.ua/method/getfile.php?fname=124285.pdf&x=1&card_id=46522&id=124285

Додаток А (обов'язковий) ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Програмний модуль для платформи оцінювання та рецензування відеоігор

Тип роботи: бакалаврська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра комп'ютерних наук
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 3,05 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Яровий А.А., зав. каф. КН _____
(прізвище, ініціали, посада) (підпис)

Колесницький О.К., проф. каф КН _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Озеранський В.С.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Арсенюк І.Р.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Лікнарович В.В.
(підпис) (прізвище, ініціали)

Додаток Б (обов’язковий) ЛІСТИНГ ПРОГРАМИ

```
package main

import (
	"fmt"
	"log"
	"math"
	"net/http"
	"server/jwt"
	"strconv"
	"time"

	"github.com/gin-gonic/gin"
	"golang.org/x/crypto/bcrypt"

	"database/sql"

	_ "github.com/lib/pq"
)

var db *sql.DB

type User struct {
	Username string
	Gender   int
	Country  string
	City     string
}

type SignUpBody struct {
	Username string `json:"username"`
	Password string `json:"password"`
	Country  string `json:"country"`
	City     string `json:"city"`
	Gender   int    `json:"gender"`
}

type Game struct {
	Name     string
	Category string
	Summary  string
	CoverId  string
}
```

```
type Games struct {
	Id      int
	Name    string
	CoverId string
}

type Review struct {
	GameId int    `json:"game_id"`
	Date   string `json:"date"`
	Master bool   `json:"master"`
	Replay bool   `json:"replay"`
	Text   string `json:"review_text"`
	Rating int    `json:"rating"`
}

type ForumPost struct {
	Id     int    `json:"id"`
	UserId string `json:"user_id"`
	GameId int    `json:"game_id"`
	Header string `json:"header"`
	Text   string `json:"text"`
}

type ForumReply struct {
	PostId int    `json:"postID"`
	Text   string `json:"text"`
	UserId string `json:"user_id"`
}

type Log struct {
	UserId       string `json:"user_id"`
	LogDate      string `json:"date"`
	Master       bool   `json:"master"`
	Replay       bool   `json:"replay"`
	Review       string `json:"review"`
	GameId       int    `json:"game_id"`
	Rating       int    `json:"rating"`
	CreationDate string `json:"creation_date"`
}

type UserPost struct {
	PostId       int    `json:"id"`
	ByUser       string `json:"by_user"`
```

```
	ToUser	string `json:"to_user"`
	Text	string `json:"text"`
	CreationDate string `json:"creation_date"`
}

type UserReply struct {
	ReplyId	int	`json:"id"`
	UserId	string `json:"user_id"`
	Text	string `json:"text"`
	CreationDate string `json:"creation_date"`
	PostId	int	`json:"post_id"`
}

func nullStringToString(ns sql.NullString) string {
   if ns.Valid {
      return ns.String
   }
   return ""
}

func handleSignUp(c *gin.Context) {
	var requestBody SignUpBody

	if err := c.BindJSON(&requestBody); err != nil {
		c.Status(http.StatusBadRequest)
	}

	if len(requestBody.Username) < 3 || len(requestBody.Password) < 6 {
		c.JSON(400, gin.H{
			"message": "username or password doesn't match needed length",
		})
	}

	if requestBody.Username == "" || requestBody.Password == "" {
		c.JSON(400, gin.H{
			"message": "username or password is empty",
		})
	}

	fmt.Printf("username: %s; password: %s; country: %s; city: %s\n",
requestBody.Username, requestBody.Password, requestBody.Country,
requestBody.City)
```

```
	hashedPassword, err := bcrypt.GenerateFromPassword([]byte(requestBody.Password), bcrypt.DefaultCost)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error in hashing password",
		})
	}

	_, err = db.Exec("INSERT INTO users (username, password, gender, country, city) VALUES ($1, $2, $3, $4)", requestBody.Username, hashedPassword, requestBody.Gender, requestBody.Country, requestBody.City)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error inserting a user in database",
		})
	}
}

func handleSignIn(c *gin.Context) {
	var requestBody SignUpBody

	if err := c.BindJSON(&requestBody); err != nil {
		c.JSON(400, gin.H{
			"message": "couldn't bind json in go",
		})
	}

	if requestBody.Username == "" || requestBody.Password == "" {
		c.JSON(400, gin.H{
			"message": "username or password is empty",
		})
	}

	fmt.Printf("trying to sign in\nusername: %s\npassword: %s\n", requestBody.Username, requestBody.Password)

	var dbPassword string

	if err := db.QueryRow("SELECT password FROM users WHERE username = $1", requestBody.Username).Scan(&dbPassword); err != nil {
		if err == sql.ErrNoRows {
			c.JSON(400, gin.H{
				"message": "user not found in db",
			})
```

```
		} else {
			log.Fatal("failed to query a row:", err)
		}
	}

	if err := bcrypt.CompareHashAndPassword([]byte(dbPassword),
[]byte(requestBody.Password)); err != nil {
		c.JSON(400, gin.H{
			"message": "wrong credentials",
		})
		return
	}

	token, err := jwt.GenerateJWT(requestBody.Username)

	if err != nil {
		log.Fatal("error generating token:", err)
	}

	c.SetCookie("auth", token, 36000, "/", "localhost", false, false)

	c.JSON(200, gin.H{
		"message": "user was found and authenticated",
	})
}

func fetchUser(c *gin.Context) {
	username, exists := c.Get("username")
	if !exists {
		c.JSON(http.StatusInternalServerError, gin.H{
			"message": "username was not found",
		})
		return
	}

	c.JSON(http.StatusOK, gin.H{
		"username": username,
	})
}

func fetchUserWithParams(c *gin.Context) {
	username, ok := c.GetQuery("username")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
```

```
                "message": "coudn't get username from get request",
            })
        return
        }

        var user User
    var country, city sql.NullString

        if err := db.QueryRow("SELECT username, gender, country, city FROM users
WHERE username = $1", username).Scan(&user.Username, &user.Gender,
&country, &city); err != nil {
            c.AbortWithStatusJSON(400, gin.H{
                "message": "no user found",
            })
        fmt.Println(err)
        return
        }

    user.Country = nullStringToString(country)
    user.City = nullStringToString(city)

        c.JSON(200, gin.H{
            "username": user.Username,
            "gender":   user.Gender,
            "country":  user.Country,
            "city":     user.City,
        })
}

func fetchGame(c *gin.Context) {
        gameID, ok := c.GetQuery("gameID")
        if !ok {
            c.AbortWithStatusJSON(400, gin.H{
                "message": "coudn't get gameid from get request",
            })
        }

        var game Game

        if err := db.QueryRow("SELECT name, category, summary, cover_id FROM
games WHERE id = $1", gameID).Scan(&game.Name, &game.Category,
&game.Summary, &game.CoverId); err != nil {
            c.AbortWithStatusJSON(400, gin.H{
                "message": "no game found",
```

```
			})
		}

		c.JSON(200, gin.H{
			"name":     game.Name,
			"category": game.Category,
			"summary":  game.Summary,
			"coverID":  game.CoverId,
		})
}

func handleSubmitReview(c *gin.Context) {
		var review Review

		if err := c.BindJSON(&review); err != nil {
			c.JSON(400, gin.H{
				"message": "couldn't bind json in go",
			})
		}

		username, exists := c.Get("username")

		fmt.Println(review)
		fmt.Println(username)

		if !exists {
			c.AbortWithStatusJSON(400, gin.H{
				"message": "couldn't find user",
			})
		}

		t := time.Now()
		formatted := t.Format("2006-01-02 15:04:05")
		_, err := db.Exec("INSERT INTO logs (user_id, game_id, master, replay, date,
review, rating, creation_date) VALUES ($1, $2, $3, $4, $5, $6, $7, $8)", username,
review.GameId, review.Master, review.Replay, review.Date, review.Text,
review.Rating, formatted)
		if err != nil {
			c.JSON(400, gin.H{
				"message": "error inserting a log in database",
			})
		}
```

```
	_, err = db.Exec("INSERT INTO ratings (user_id, game_id, rating) VALUES
($1, $2, $3) ON CONFLICT (user_id, game_id) DO UPDATE SET rating =
EXCLUDED.rating", username, review.GameId, review.Rating)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error upserting rating",
		})
		return
	}

	c.JSON(200, gin.H{"message": "review submitted successfully"})
}

func fetchGameCount(c *gin.Context) {
	var count int
	err := db.QueryRow("SELECT COUNT(*) FROM games").Scan(&count)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error fetching count of games from database",
		})
	}

	c.JSON(200, gin.H{
		"count": count,
	})
}

func fetchGames(c *gin.Context) {
	page, ok := c.GetQuery("page")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "coudn't get page from get request",
		})
	}

	page_int, err := strconv.Atoi(page)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error converting string page to int",
		})
	}

	rows, err := db.Query("SELECT id, name, cover_id FROM games ORDER
BY id ASC LIMIT 50 OFFSET $1", (page_int-1)*50)
```

```
	if err != nil {
		if err == sql.ErrNoRows {
			c.JSON(400, gin.H{
				"message": "couldn't find games in db",
			})
		} else {
			log.Fatal("failed to query a row:", err)
		}
	}
	defer rows.Close()

	var games []Games
	for rows.Next() {
		var g Games
		if err := rows.Scan(&g.Id, &g.Name, &g.CoverId); err != nil {
			log.Fatal("failed scanning value in array:", err)
		}
		games = append(games, g)
	}

	c.JSON(http.StatusOK, games)
}

func fetchReviews(c *gin.Context) {
	gameID, ok := c.GetQuery("gameID")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "coudn't get gameid from get request",
		})
	}

	rows, err := db.Query("SELECT user_id, date, creation_date, master, replay,
review, rating FROM logs WHERE review IS NOT NULL AND review <> '' AND
game_id = $1 ORDER BY date ASC", gameID)
	if err != nil {
		if err == sql.ErrNoRows {
			c.JSON(400, gin.H{
				"message": "couldn't find reviews in db",
			})
		} else {
			log.Fatal("failed to query a row:", err)
		}
	}
	defer rows.Close()
```

```
	var reviews []Log
	for rows.Next() {
		var r Log
		if err := rows.Scan(&r.UserId, &r.LogDate, &r.CreationDate,
&r.Master, &r.Replay, &r.Review, &r.Rating); err != nil {
			log.Fatal("failed scanning value in array:", err)
		}
		reviews = append(reviews, r)
	}

	c.JSON(http.StatusOK, reviews)
}

func handleForumPost(c *gin.Context) {
	username, exists := c.Get("username")
	if !exists {
		c.JSON(http.StatusInternalServerError, gin.H{
			"message": "username was not found",
		})
		return
	}

	var forumPost ForumPost

	if err := c.BindJSON(&forumPost); err != nil {
		c.Status(http.StatusBadRequest)
	}

	t := time.Now()
	formatted := t.Format("2006-01-02 15:04:05")
	_, err := db.Exec("INSERT INTO forum_posts (user_id, header, text,
creation_date, game_id) VALUES ($1, $2, $3, $4, $5)", username,
forumPost.Header, forumPost.Text, formatted, forumPost.GameId)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error inserting a forum post in database",
		})
	}
}

func fetchForumPosts(c *gin.Context) {
	gameID, ok := c.GetQuery("gameID")
	if !ok {
```

```
		c.AbortWithStatusJSON(400, gin.H{
			"message": "coudn't get gameid from get request",
		})
	}

	rows, err := db.Query("SELECT id, user_id, game_id, header, text FROM
forum_posts WHERE game_id = $1 ORDER BY creation_date DESC", gameID)
	if err != nil {
		if err == sql.ErrNoRows {
			c.JSON(400, gin.H{
				"message": "couldn't find games in db",
			})
		} else {
			log.Fatal("failed to query a row:", err)
		}
	}
	defer rows.Close()

	var forumPosts []ForumPost
	for rows.Next() {
		var f ForumPost
		if err := rows.Scan(&f.Id, &f.UserId, &f.GameId, &f.Header, &f.Text);
err != nil {
			log.Fatal("failed scanning value in array:", err)
		}
		forumPosts = append(forumPosts, f)
	}

	c.JSON(http.StatusOK, forumPosts)
}

func fetchForumPost(c *gin.Context) {
	postID, ok := c.GetQuery("postID")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "coudn't get gameid from get request",
		})
	}

	var forumPost ForumPost

	if err := db.QueryRow("SELECT id, user_id, header, text, game_id FROM
forum_posts WHERE id = $1", postID).Scan(&forumPost.Id, &forumPost.UserId,
&forumPost.Header, &forumPost.Text, &forumPost.GameId); err != nil {
```

```
			if err == sql.ErrNoRows {
				c.JSON(400, gin.H{
					"message": "post not found in db",
				})
			} else {
				log.Fatal("failed to query a row:", err)
			}
		}

		c.JSON(200, forumPost)
}

func handleForumReply(c *gin.Context) {
		username, exists := c.Get("username")
		if !exists {
			c.JSON(http.StatusInternalServerError, gin.H{
				"message": "username was not found",
			})
			return
		}

		var forumReply ForumReply

		if err := c.BindJSON(&forumReply); err != nil {
			c.Status(http.StatusBadRequest)
		}

		fmt.Println(forumReply)

		t := time.Now()
		formatted := t.Format("2006-01-02 15:04:05")
		_, err := db.Exec("INSERT INTO forum_replies (user_id, text, creation_date,
post_id) VALUES ($1, $2, $3, $4)", username, forumReply.Text, formatted,
forumReply.PostId)
		if err != nil {
			c.JSON(400, gin.H{
				"message": "error inserting a forum post in database",
			})
		}
}

func fetchForumReplies(c *gin.Context) {
		postID, ok := c.GetQuery("postID")
		if !ok {
```

```
			c.AbortWithStatusJSON(400, gin.H{
				"message": "coudn't get gameid from get request",
			})
		}

		rows, err := db.Query("SELECT user_id, text FROM forum_replies WHERE
post_id = $1 ORDER BY creation_date DESC", postID)
		if err != nil {
			if err == sql.ErrNoRows {
				c.JSON(400, gin.H{
					"message": "couldn't find games in db",
				})
			} else {
				log.Fatal("failed to query a row:", err)
			}
		}
		defer rows.Close()

		var forumReplies []ForumReply
		for rows.Next() {
			var f ForumReply
			if err := rows.Scan(&f.UserId, &f.Text); err != nil {
				log.Fatal("failed scanning value in array:", err)
			}
			forumReplies = append(forumReplies, f)
		}

		c.JSON(http.StatusOK, forumReplies)
}

func fetchLogs(c *gin.Context) {
		userID, ok := c.GetQuery("userID")
		if !ok {
			c.AbortWithStatusJSON(400, gin.H{
				"message": "coudn't get gameid from get request",
			})
		}

		rows, err := db.Query("SELECT date, rating, master, replay, review, game_id,
creation_date FROM logs WHERE user_id = $1 ORDER BY date DESC", userID)
		if err != nil {
			if err == sql.ErrNoRows {
				c.JSON(400, gin.H{
					"message": "couldn't find logs in db",
```

```
                })
        } else {
                log.Fatal("failed to query a row:", err)
        }
    }
    defer rows.Close()

    var logs []Log
    for rows.Next() {
        var l Log
        if err := rows.Scan(&l.LogDate, &l.Rating, &l.Master, &l.Replay,
&l.Review, &l.GameId, &l.CreationDate); err != nil {
            log.Fatal("failed scanning value in array:", err)
        }
        l.UserId = userID
        logs = append(logs, l)
    }

    c.JSON(http.StatusOK, logs)
}

func fetchUserPosts(c *gin.Context) {
    userID, ok := c.GetQuery("userID")
    if !ok {
        c.AbortWithStatusJSON(400, gin.H{
            "message": "coudn't get gameid from get request",
        })
    }

    rows, err := db.Query("SELECT id, text, by_user, creation_date FROM
user_posts WHERE to_user = $1 ORDER BY creation_date DESC", userID)
    if err != nil {
        if err == sql.ErrNoRows {
            c.JSON(400, gin.H{
                "message": "couldn't find user_posts in db",
            })
        } else {
            log.Fatal("failed to query a row:", err)
        }
    }
    defer rows.Close()

    var posts []UserPost
    for rows.Next() {
```

```
		var p UserPost
		if err := rows.Scan(&p.PostId, &p.Text, &p.ByUser, &p.CreationDate);
err != nil {
			log.Fatal("failed scanning value in array:", err)
		}
		posts = append(posts, p)
	}

	c.JSON(http.StatusOK, posts)
}

func fetchUserPost(c *gin.Context) {
	postID, ok := c.GetQuery("postID")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "couldn't get postID from request",
		})
	}

	fmt.Println(postID)
	var post UserPost
	if err := db.QueryRow("SELECT id, text, by_user, to_user, creation_date
FROM user_posts WHERE id = $1", postID).Scan(&post.PostId, &post.Text,
&post.ByUser, &post.ToUser, &post.CreationDate); err != nil {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "no user post found",
		})
		return
	}

	c.JSON(200, post)
}

func handleUserReply(c *gin.Context) {
	username, exists := c.Get("username")
	if !exists {
		c.JSON(http.StatusInternalServerError, gin.H{
			"message": "username was not found",
		})
		return
	}

	var userReply UserReply
```

```
	if err := c.BindJSON(&userReply); err != nil {
		c.Status(http.StatusBadRequest)
	}

	fmt.Printf("user_id: %v, post_id: %v, text: %v\n", username, userReply.PostId,
userReply.Text)
	t := time.Now()
	formatted := t.Format("2006-01-02 15:04:05")
	_, err := db.Exec("INSERT INTO user_replies (user_id, post_id, text,
creation_date) VALUES ($1, $2, $3, $4)", username, userReply.PostId,
userReply.Text, formatted)
	if err != nil {
		c.JSON(400, gin.H{
			"message": "error inserting a user reply in database",
		})
	}
}

func fetchUserReplies(c *gin.Context) {
	postID, ok := c.GetQuery("postID")
	if !ok {
		c.AbortWithStatusJSON(400, gin.H{
			"message": "coudn't get postID from get request",
		})
	}

	rows, err := db.Query("SELECT id, user_id, post_id, creation_date, text
FROM user_replies WHERE post_id = $1 ORDER BY creation_date DESC",
postID)
	if err != nil {
		if err == sql.ErrNoRows {
			c.JSON(400, gin.H{
				"message": "couldn't find user_posts in db",
			})
		} else {
			log.Fatal("failed to query a row:", err)
		}
	}
	defer rows.Close()

	var replies []UserReply
	for rows.Next() {
		var r UserReply
```

```
			if err := rows.Scan(&r.ReplyId, &r.UserId, &r.PostId, &r.CreationDate,
&r.Text); err != nil {
				log.Fatal("failed scanning value in array:", err)
			}
			replies = append(replies, r)
		}

		c.JSON(http.StatusOK, replies)
}

func fetchAverageRating(c *gin.Context) {
		gameID, ok := c.GetQuery("gameID")
		if !ok {
			c.AbortWithStatusJSON(400, gin.H{
				"message": "coudn't get gameid from get request",
			})
			return
		}

		gameIDint, err := strconv.Atoi(gameID)
		if err != nil {
			c.JSON(400, gin.H{
				"message": "error converting gameID to int",
			})
			return
		}

		var count int
		if err := db.QueryRow("SELECT COUNT(rating) FROM ratings WHERE
game_id = $1", gameIDint).Scan(&count); err != nil {
			if err == sql.ErrNoRows {
				c.JSON(400, gin.H{
					"message": "couldn't find ratings for gameid",
				})
				return
			} else {
				log.Fatal("failed to query a row:", err)
			}
		}

		rows, err := db.Query("SELECT rating FROM ratings WHERE game_id =
$1", gameIDint)
		if err != nil {
			if err == sql.ErrNoRows {
```

```
                c.JSON(400, gin.H{
                        "message": "couldn't find ratings in db",
                })
        } else {
                log.Fatal("failed to query a row:", err)
        }
    }
    defer rows.Close()

    var ratings []int
    for rows.Next() {
        var r int
        if err := rows.Scan(&r); err != nil {
                log.Fatal("failed scanning value in array:", err)
        }
        ratings = append(ratings, r)
    }

    sum := 0
    for _, v := range ratings {
        sum += v
    }

    var avg_rating float64 = float64(sum) / float64(count)
    if math.IsNaN(avg_rating) {
        avg_rating = 0
    }

    c.JSON(http.StatusOK, gin.H{
        "average": avg_rating,
    })
}

func fetchUsersByGeo(c *gin.Context) {
    onlyCountry := false

    country, ok := c.GetQuery("country")
    if !ok {
        c.AbortWithStatusJSON(400, gin.H{
            "message": "couldn't get country from request",
        })
        return
    }
```

```
	city, ok := c.GetQuery("city")
	if !ok {
		onlyCountry = true
	}

	var rows *sql.Rows
	var err error
	if !onlyCountry {
		rows, err = db.Query("SELECT username FROM users WHERE
country = $1 AND city = $2", country, city)
		if err != nil {
			log.Fatal("failed to query a row:", err)
		}
	} else {
		rows, err = db.Query("SELECT username FROM users WHERE
country = $1", country)
		if err != nil {
			log.Fatal("failed to query a row:", err)
		}
	}
	defer rows.Close()

	var users []string
	for rows.Next() {
		var u string
		if err := rows.Scan(&u); err != nil {
			log.Fatal("failed scanning value in array:", err)
		}
		users = append(users, u)
	}

	c.JSON(http.StatusOK, users)
}

func main() {
	connStr := "host=localhost port=5432 user=postgres password=5457
dbname=test sslmode=disable"
	var err error
	db, err = sql.Open("postgres", connStr)
	if err != nil {
		fmt.Println("connection to db has failed")
		log.Fatal(err)
	}
```

```
	r := gin.Default()
	r.POST("/sign-up", handleSignUp)
	r.POST("/sign-in", handleSignIn)
	r.POST("/post_review", jwt.AuthMiddleware(), handleSubmitReview)
	r.POST("/post_forum_post", jwt.AuthMiddleware(), handleForumPost)
	r.POST("/post_forum_reply", jwt.AuthMiddleware(), handleForumReply)
	r.POST("/post_user_reply", jwt.AuthMiddleware(), handleUserReply)
	r.GET("/fetch_user", jwt.AuthMiddleware(), fetchUser)
	r.GET("/fetch_user_params", fetchUserWithParams)
	r.GET("/fetch_user_posts", fetchUserPosts)
	r.GET("/fetch_user_post", fetchUserPost)
	r.GET("/fetch_user_replies", fetchUserReplies)
	r.GET("/fetch_logs", fetchLogs)
	r.GET("/fetch_reviews", fetchReviews)
	r.GET("/fetch_game", fetchGame)
	r.GET("/fetch_games", fetchGames)
	r.GET("/fetch_forum_posts", fetchForumPosts)
	r.GET("/fetch_forum_post", fetchForumPost)
	r.GET("/fetch_forum_replies", fetchForumReplies)
	r.GET("/fetch_game_count", fetchGameCount)
	r.GET("/fetch_average_rating", fetchAverageRating)
	r.GET("/fetch_users_by_geo", fetchUsersByGeo)

	r.Run(":8080")
}
```

Додаток В (обов'язковий) ІЛЮСТРАТИВНА ЧАСТИНА

ІЛЮСТРАТИВНА ЧАСТИНА

Програмний модуль для платформи оцінювання та рецензування відеоігор

Виконав: студент 4 курсу, групи 1КН-21б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)

Лікнарович В. В.
(прізвище та ініціали)

Керівник: к.ф.-м.н., доцент каф. КН
Арсенюк І. Р.
(прізвище та ініціали)

«03» червня 2025 р.

Вінниця ВНТУ - 2025 рік

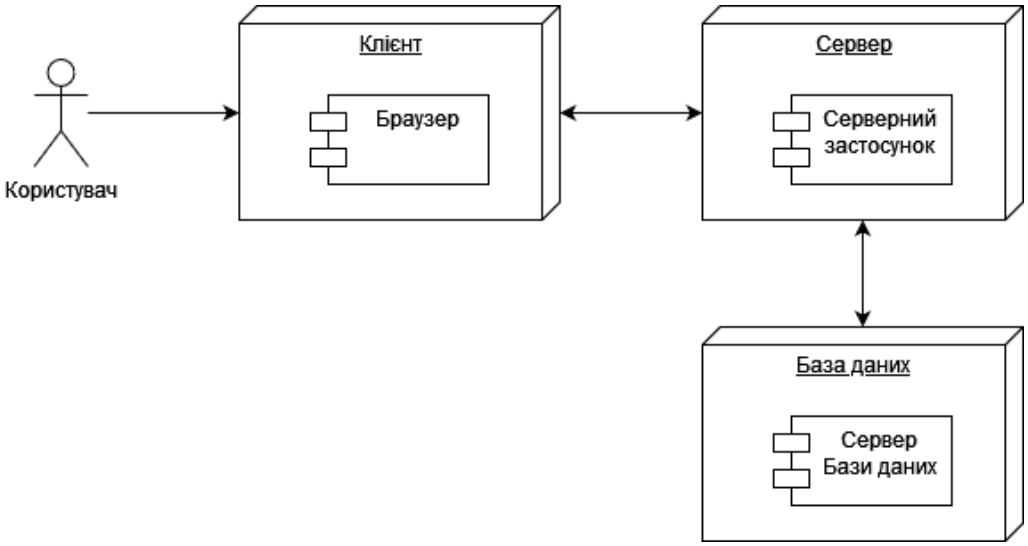

Рисунок В.1 – Діаграма розгортання програмного модуля

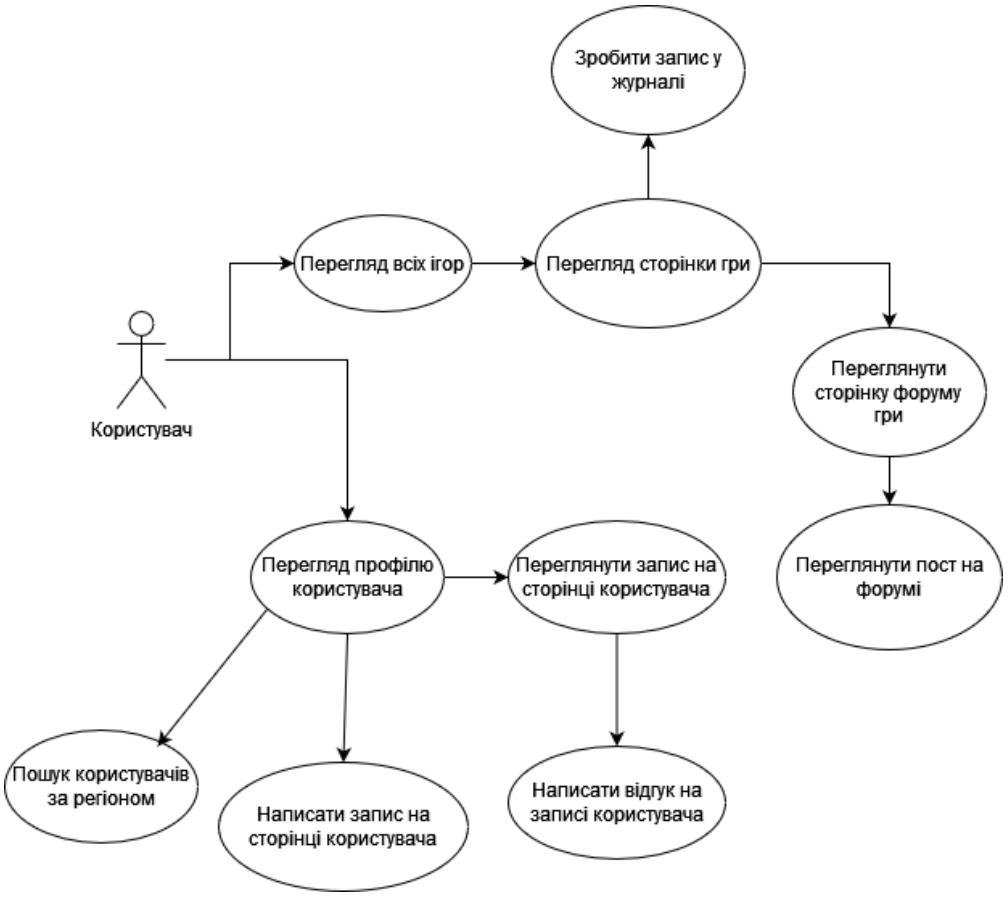

Рисунок В.2 – Діаграма прецедентів програмного модуля

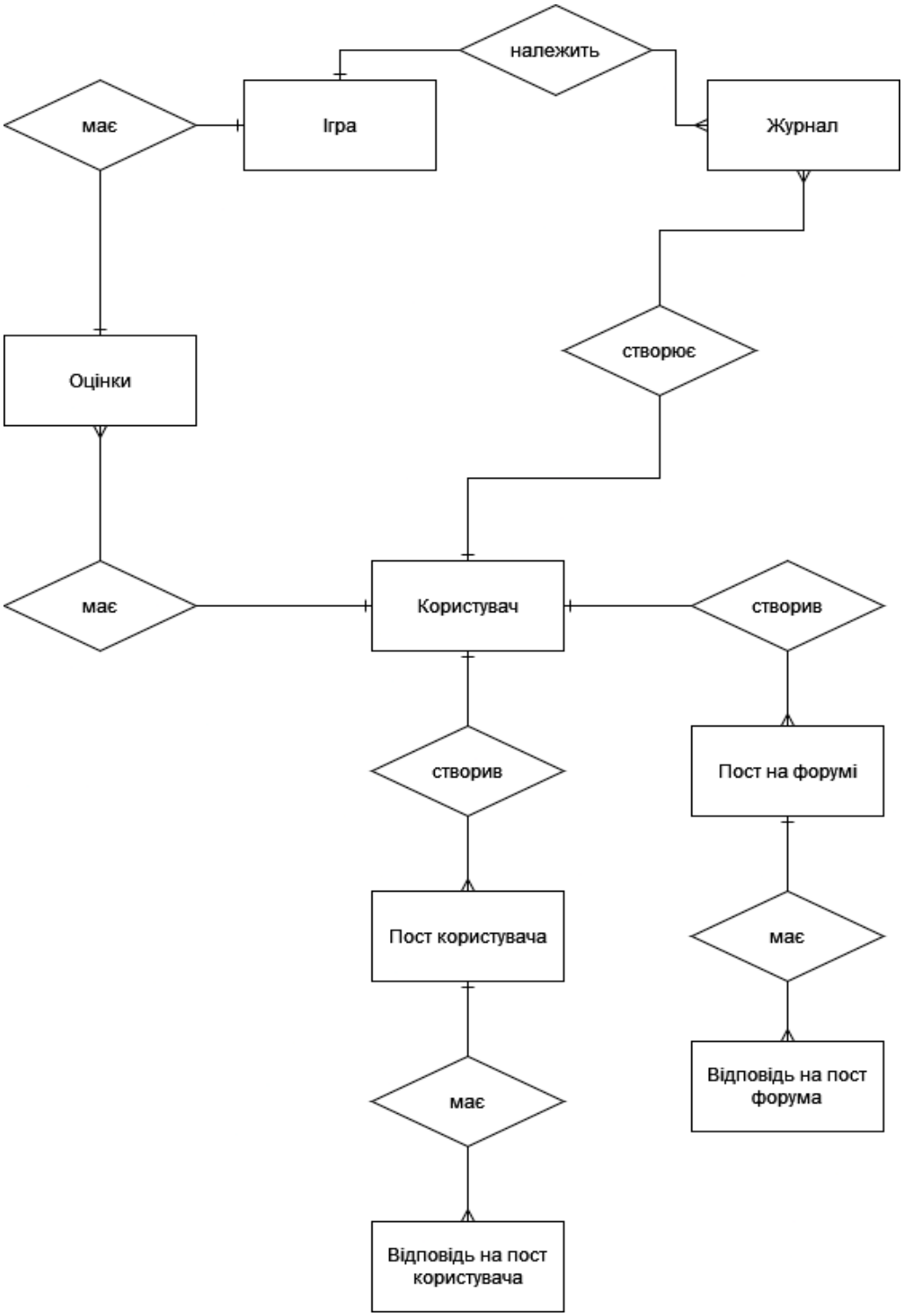

Рисунок В.3 – ER-діаграма програмного модуля

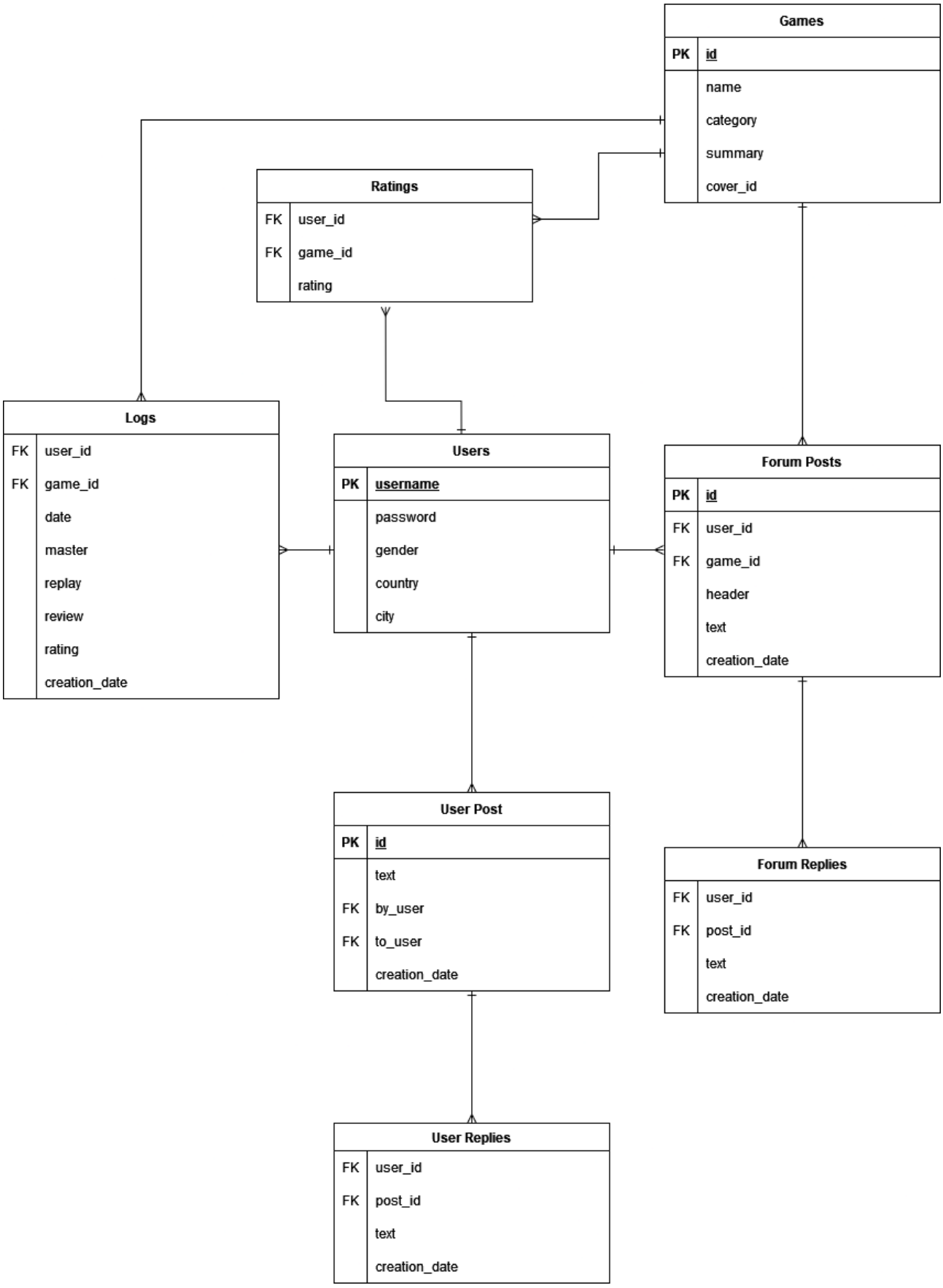

Рисунок В.4 – Схема бази даних програмного модуля

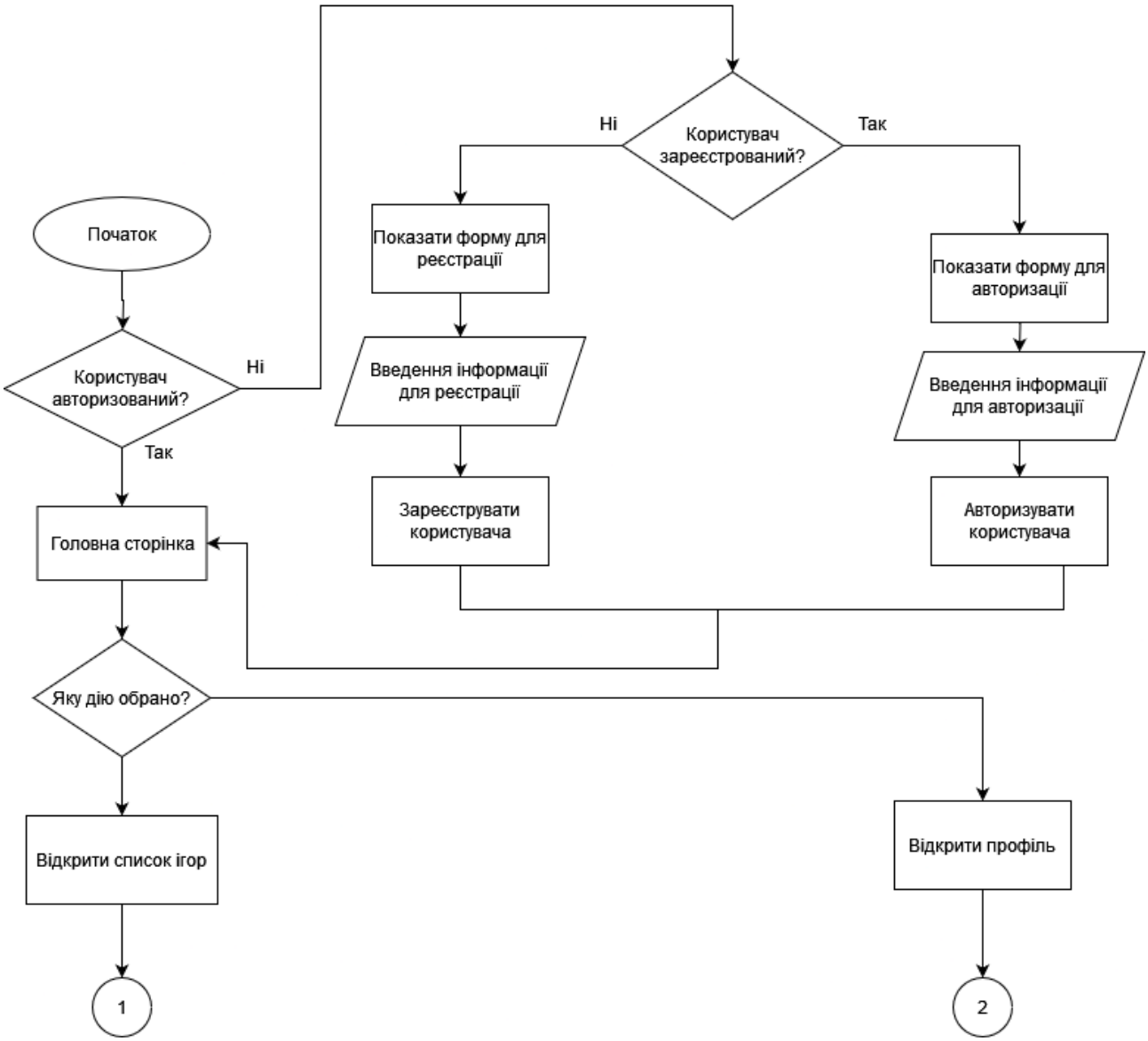

Рисунок В.5 – Алгоритм роботи програмного модуля для оцінювання та рецензування відеоігор

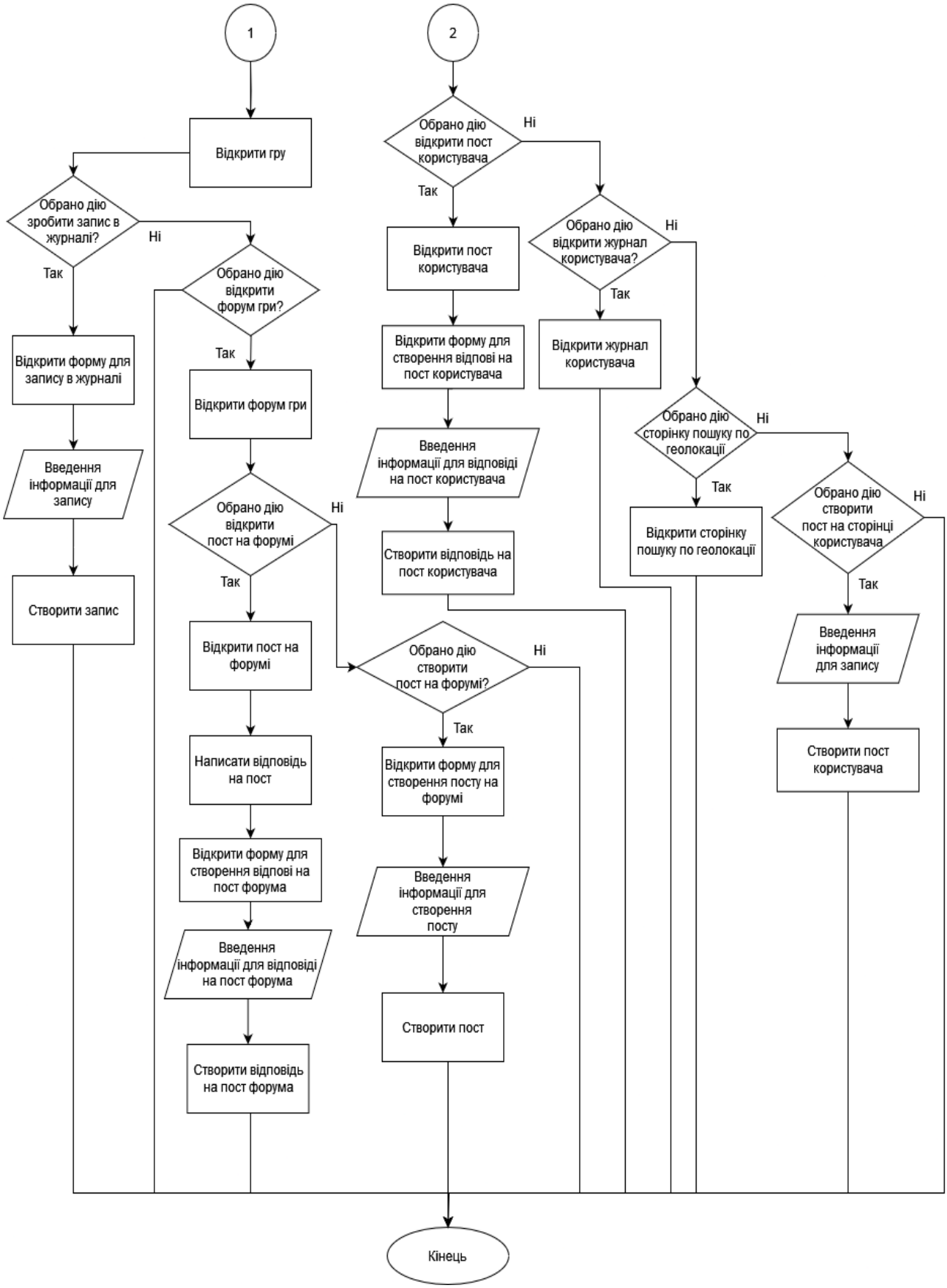

Рисунок В.5 – Продовження

Додаток Г (довідниковий) ІНСТРУКЦІЯ КОРИСТУВАЧА

Для взаємодії з функціями програмного модуля потрібно спочатку зареєструватись використовуючи форму для реєстрації. Після реєстрації/авторизації з’явиться головне вікно де користувач може натиснути кнопку Browse для того щоб відкрити список ігор з якими хоче взаємодіяти користувач. Вікно вибору ігор зображено на рисунку Г.1.

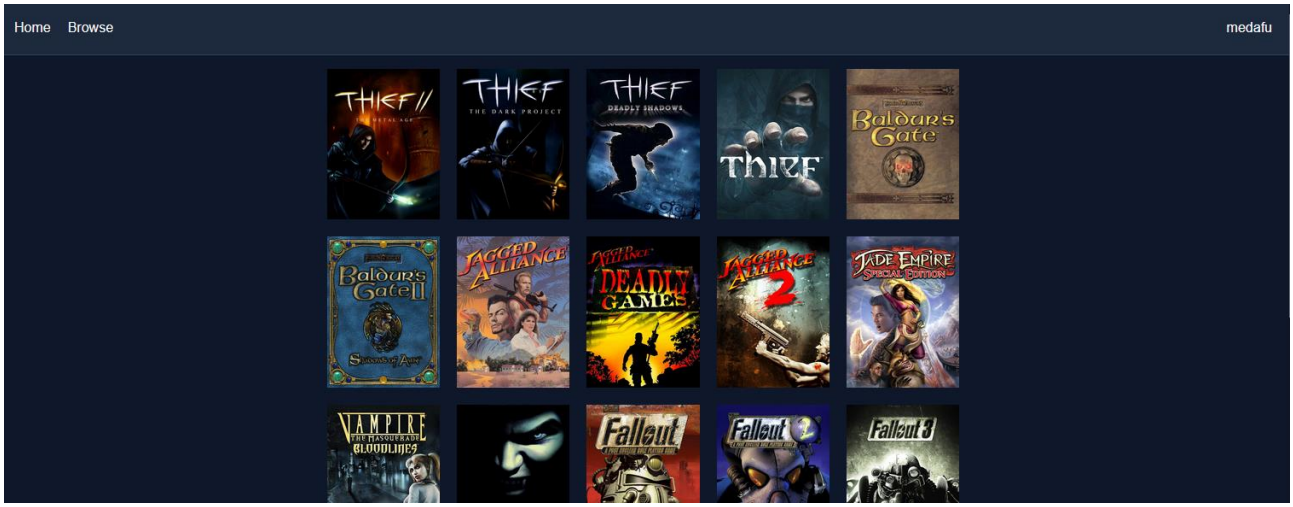

Рисунок Г.1 – Вікно вибору ігор

Після обрання гри ми потрапляємо на сторінку гри де можемо створити запис або відкрити форум. Для створення запису натисніть кнопку «Log» після чого з’явиться форма для створення запису. Форма для створення запису зображена на рисунку Г.2.

На сторінці форума гри користувач може створити новий пост на форумі або відкрити вже існуючий. Для створення нового посту користувач може використати форму на сторінці форума. Форма для створення нового посту на форумі гри зображено на рисунку Г.3.

Повертаючись на головний екран, можемо відкрити сторінку профіля користувача. На цій сторінці можемо побачити інформацію про користувача,

пости під сторінкою користувача та перейти на сторінку пошуку інших користувачів по заданій країні/місту.

Рисунок Г.2 – Форма для створення запису

Рисунок Г.3 – Форма для створення нового посту на форумі

Для створення посту для користувача потрібно використати текстове поле на сторінці користувача після чого натиснути кнопку Post. Якщо користувач хоче написати відповідь на пост на сторінці користувача, то він натискає на обраний пост після чого відкриється сторінка поста користувача де можна знайти текстове поле для написання відповіді та кнопка відправки відповіді «Reply». Сторінка поста користувача зображена на рисунку Г.4.

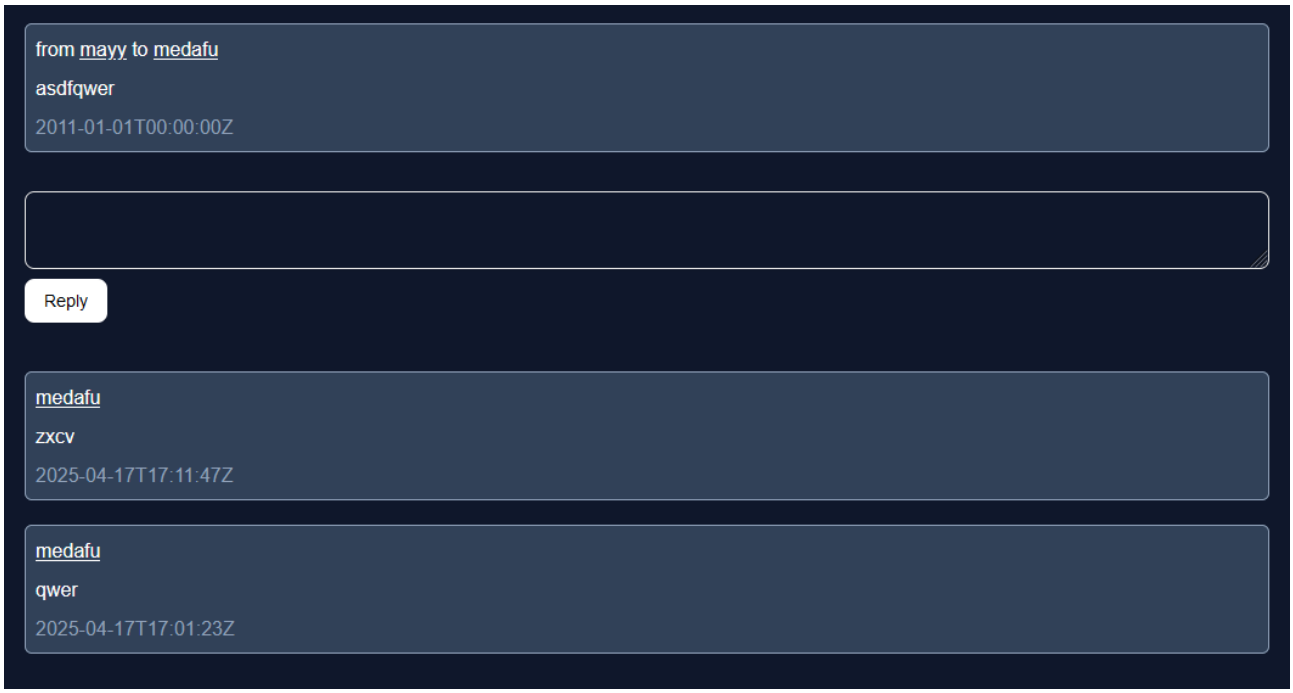

Рисунок Г.4 – Сторінка поста користувача

Для того щоб відкрити сторінку пошуку користувачів за країною та містом на сторінці профілю користувача натисніть на країну яку вказав користувач. Після чого відкриється сторінка списку людей зі спільною країною. Сторінка пошуку користувачів за країною та містом зображена на рисунку Г.5.

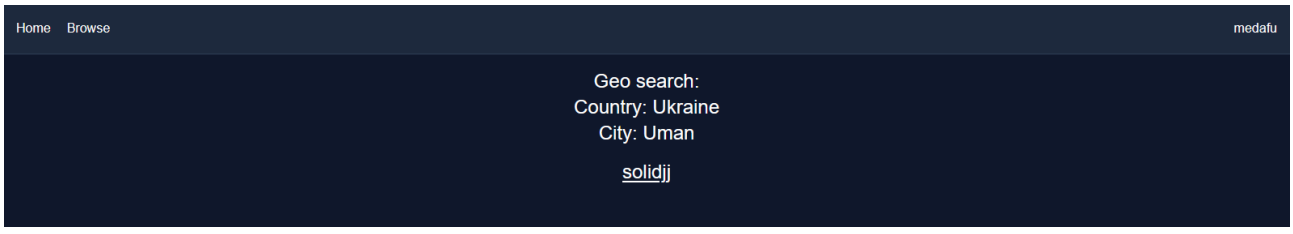

Рисунок Г.5 – Сторінка пошуку користувачів за країною та містом