НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
/Голуб Б.Л., доц., к.т.н. /

підпис ПІБ, вчене звання і ступінь

«___»___________2025 р

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Програмне забезпечення системи управління районними локаціями міста»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент / Вайганг Г.О./
Науковий ступень та вчене звання підпис ПІБ

Керівник бакалаврської кваліфікаційної роботи : / Семко О.В./
підпис ПІБ

Виконав: / Козаченко М.М./
підпис ПІБ

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп’ютерних наук

_________________________ / Голуб Б.Л., доцент, к.т.н /
підпис
“ ” 2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студенту Козаченку Миколі Миколайовичу

Спеціальність 121 «Інженерія програмного забезпечення»

1. Тема роботи: Програмне забезпечення системи управління районними локаціями міста

Затверджена наказом ректора НУБіП України № 2248 “С” від 16.12.2024

2. Термін подання завершеної роботи на кафедру 2025 . __. ___
рік, місяць, число

3. Вихідні дані до роботи: опис програмного забезпечення

4. Перелік питань що розглядаються:

1. Аналіз проблемної області.
2. Вибір та обґрунтування засобів для розробки системи.
3. Проектування інформаційної системи.
4. Висновки.

Керівник бакалаврської кваліфікаційної роботи_________ / Семко О.В./
підпис ініціали та прізвище

Завдання прийняв до виконання ____________________ / Козаченко М.М. /
підпис ініціали та прізвище

Дата отримання завдання ______.___.___
рік, місяць, число

Анотація

Дипломна робота присвячена розробці програмного забезпечення для системи управління районними локаціями міста. Моя головна мета була створити інструмент для автоматизаії обліку, адміністрування та візуалізації різних об'єктів міста. В роботі розглянув архітектуру самої системи, принципи функціонування та реалізував основні модулі, що забезпечують зручну взаємодію з даними міських локацій.

ABSTRACT

The thesis is devoted to the development of software for the management system of district locations of the city. My main goal was to create a tool for automating accounting, administration and visualization of various city objects. In the work, I examined the architecture of the system itself, the principles of functioning and implemented the main modules that provide convenient interaction with data of city locations.

ЗМІСТ

ВСТУП...5
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛСТІ... 8
1.1 Постановка задачі...8
1.2 Моделювання предметної області.. 10
1.3 Діаграма прецедентів... 11
1.5 Діаграма послідовності..16
1.6 Діаграма класів.. 19
1.6.1 Діаграма класів для кооперації: Додавання коментаря до локації........20
1.6.2 Діаграма класів для кооперації: Оцінювання локації...........................21
1.6.3 Діаграма класів для кооперації: Надсилання повідомлення в чаті....... 22
2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ...23
2.1 Загальні відомості про ER-діаграму...23
2.2 Побудова ER- діаграми...25
2.3 Вибір та обґрунтування СУБД...27
2.4 Створення БД..31
2.5 Додавання користувачів до БД.. 34
2.6 Модель даних фізичного рівня.. 36
3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕПЕЧЕННЯ...38
3.1 Вибір інструментарію для розробки програмного забезпечення......... 38
3.2 Діаграма пакетів... 41
3.3 Принципи роботи у середовищі візуальної розробки програм........... 42
4. ВПРОВАДЖЕННЯ СИСТЕМИ...45
4.1 Тестування системи... 45
4.2 Апаратні та технічі засоби...48
4.3 Опис роботи програми...51
ВИСНОВКИ..58
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.. 60
ДОДАТОК А База даних... 1
ДОДАТОК Б Код програми...1

ВСТУП

Сучасне управління міськими ресурсами та інфраструктурою вимагає використання інформаційних технологій для забезпечення ефективного планування, моніторингу та взаємодії між адміністрацією міста та його мешканцями. Одним із ключових елементів є організація та управління локаціями, які поділені на райони, що дозволяє оптимізувати доступ до інформації про міські об'єкти, такі як парки, транспортні вузли, торговельні центри чи культурні пам'ятки. У містах із розвиненою інфраструктурою виникає потреба в автоматизованих системах, які забезпечують зручний доступ до даних про локації, їхній стан, а також дозволяють мешканцям взаємодіяти з адміністрацією через коментарі, оцінки та повідомлення.

Управління міськими локаціями ускладнюється через велику кількість об'єктів, різноманітність їх типів і необхідність постійного оновлення інформації. Без автоматизації процеси збору, обробки та аналізу даних про локації потребують значних людських ресурсів і часу, що може призводити до затримок у прийнятті рішень, зниження якості обслуговування населення та зростання кількості помилок. Наприклад, ручне ведення обліку статусу об'єктів (активний, у ремонті чи запланований) або обробка відгуків громадян є трудомістким і схильним до людських помилок.

Автоматизація управління локаціями дозволяє вирішити ці проблеми, надаючи централізовану платформу для зберігання, обробки та представлення даних. Такі системи зменшують час, необхідний для пошуку інформації, забезпечують прозорість управління та підвищують рівень взаємодії між адміністрацією міста та його мешканцями. Крім того, впровадження автоматизованих систем сприяє економії ресурсів, зниженню ймовірності помилок і підвищенню ефективності роботи управлінського апарату.

Метою бакалаврської кваліфікаційної роботи є розробка програмного забезпечення системи управління районними локаціями міста, що забезпечує зручне зберігання, обробку та представлення інформації про міські об'єкти, а

також підтримує взаємодію користувачів із системою через оцінки, коментарі та чат із менеджерами. Система має бути простою у використанні, доступною для різних категорій користувачів (мешканців і адміністраторів) та адаптованою до реальних потреб міста.

Одним із ключових елементів системи є база даних, яка забезпечує структуроване зберігання інформації про райони, локації, користувачів, їхні взаємодії (коментарі, оцінки, повідомлення) та історію переглядів. Моделювання бази даних є важливим етапом розробки, оскільки від її структури залежить ефективність роботи системи в цілому. Розробка системи передбачає створення серверної частини для обробки запитів і клієнтської частини для зручної взаємодії користувачів.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- вивчити предметну область управління міськими локаціями та зібрати вимоги до системи;
- проаналізувати існуючі рішення у сфері автоматизації міського управління;
- сформулювати функціональні та нефункціональні вимоги до програмного продукту;
- розробити моделі бізнес-процесів за допомогою UML-діаграм;
- спроектувати загальний алгоритм роботи системи;
- провести аналіз засобів розробки та обрати оптимальний інструментарій;
- розробити структуру бази даних для зберігання інформації про райони, локації та взаємодію користувачів;
- реалізувати базу даних засобами SQLite;
- розробити серверну частину системи з використанням Flask для обробки запитів;

- створити API для взаємодії клієнтської частини з сервером;
- провести тестування розробленої системи для забезпечення її стабільності та відповідності вимогам.

У ході розробки основна увага приділяється створенню програмного продукту, який дозволяє зручно зберігати, обробляти, редагувати та переглядати інформацію про локації, а також забезпечує швидкий доступ до даних у зрозумілому вигляді. Система має спростити роботу адміністраторів шляхом автоматизації управління локаціями та забезпечити мешканцям можливість залишати відгуки, оцінювати об’єкти та звертатися до менеджерів через чат. Таким чином, розроблене програмне забезпечення сприятиме підвищенню ефективності управління міськими ресурсами та покращенню взаємодії між адміністрацією міста та його мешканцями.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛСТІ

1.1 Постановка задачі

Система управління районними локаціями міста призначена для спрощення роботи з інформацією про міські об'єкти (локації) для адміністраторів (менеджерів) та забезпечення зручного доступу до даних для звичайних користувачів. Система має допомогти в організації, обліку та управлінні інформацією про різноманітні локації, такі як парки, пам'ятки, транспортні вузли, торгові центри тощо, з урахуванням їх географічного розташування в межах районів міста.

Основні цілі програми:

- спростити управління інформацією про локації для менеджерів міста;
- забезпечити користувачам зручний доступ до актуальної інформації про міські об'єкти;
- створити базу даних районів і локацій з можливістю фільтрації та пошуку;
- вести облік взаємодії користувачів з локаціями (оцінки, коментарі, перегляди).

Процес управління локаціями пов'язаний з певними труднощами, зокрема:

- складністю централізованого зберігання та оновлення інформації про різноманітні об'єкти;
- відсутністю зручного інструменту для аналізу та фільтрації даних про локації за районами, типами чи статусом;
- необхідністю взаємодії між адміністраторами та користувачами для обробки запитів і відгуків.

Хоча розроблене програмне забезпечення не забезпечує повної автоматизації всіх процесів управління міськими локаціями, воно надає зручний інтерфейс та функціонал для виконання основних завдань з адміністрування та перегляду інформації.

Серед функцій, які виконуватиме розроблена інформаційна система, слід виділити такі:

- управління каталогом районів і локацій (додавання, редагування, видалення);
- збереження інформації про локації, включаючи їх координати, тип, статус і описи;
- підтримка взаємодії користувачів через коментарі, оцінки та чат з менеджерами;
- збереження історії переглядів локацій користувачами;
- авторизація та розмежування доступу для менеджерів і звичайних користувачів.

Підсумовуючи, система управління районними локаціями міста має бути простим у використанні програмним продуктом, зрозумілим як для менеджерів, так і для кінцевих користувачів. Вона повинна бути адаптованою до потреб користувачів, забезпечуючи зручний доступ до інформації та інтуїтивно зрозумілий інтерфейс.

У системі вся інформація зберігається в базі даних, що дозволяє отримувати актуальні дані в реальному часі. Система має містити таку інформацію:

- відомості про райони (назви, межі, кольорове кодування);
- відомості про локації (назви, типи, координати, статуси, описи);
- відомості про користувачів та їхню взаємодію (коментарі, оцінки, повідомлення).

Система має забезпечувати:

- введення, редагування та видалення даних про райони та локації;
- обмеження доступу через авторизацію (логін і пароль);
- фільтрацію та пошук локацій за різними критеріями (район, тип, статус, ключові слова);
- збереження історії взаємодії користувачів з локаціями;
- чат між користувачами та менеджерами для оперативної комунікації.

Програма має відзначатися простотою, зручністю та зрозумілістю концепцій управління інформацією про міські локації.

1.2 Моделювання предметної області

Моделювання предметної області є ключовим етапом у розробці програмного забезпечення для управління районними локаціями міста [1]. На цьому етапі створюється фундамент для подальшого проектування системи, що дозволяє уникнути помилок і забезпечити її відповідність поставленим вимогам. Витрати на моделювання можуть складати значну частину загального бюджету проекту, оскільки саме на цьому етапі виявляються та виправляються недоліки, які на пізніших етапах можуть призвести до значних витрат або навіть провалу проекту. Адекватна модель предметної області допомагає чітко визначити структуру, зв’язки та функціональні особливості системи, що є основою для її успішної реалізації.

Інформаційна модель системи управління районними локаціями міста являє собою сукупність бази даних, серверної логіки та клієнтських додатків, об’єднаних у єдину систему. Ця модель включає такі ключові компоненти, як інформація про райони міста, локації, користувачів, їхні взаємодії (коментарі, оцінки, повідомлення) та історію переглядів. При проектуванні структури моделі необхідно враховувати взаємозв’язки між цими компонентами, їхні

характеристики та правила функціонування системи. Чим складніша система, тим важливіше створити несуперечливу та збалансовану модель, яка буде зрозумілою для всіх учасників розробки.

Для моделювання предметної області системи управління районними локаціями міста використано уніфіковану мову моделювання UML (Unified Modeling Language), яка є визнаним стандартом у світі розробки програмного забезпечення. UML дозволяє створювати чіткі та наочні моделі, які використовуються системними аналітиками, розробниками, тестувальниками та іншими фахівцями. Ця мова підтримує специфікацію, візуалізацію, проектування та документування компонентів системи, що робить її ідеальним інструментом для створення концептуальних і логічних моделей. На ринку доступні численні UML-орієнтовані інструменти, такі як Enterprise Architect, StarUML або Visual Paradigm, які автоматизують процес моделювання та полегшують створення діаграм.

Основним принципом моделювання предметної області є принцип абстрагування, який передбачає включення до моделі лише тих елементів системи, які є критично важливими для її функціонування. Наприклад, у даній системі акцент зроблено на структурі районів, типах локацій, ролях користувачів (менеджери та звичайні користувачі) та їхніх взаємодіях, тоді як другорядні деталі, такі як внутрішні технічні показники реалізації, ігноруються на етапі моделювання. Іншим важливим принципом є багатомодельність, який вказує на те, що жодна окрема модель не може повною мірою описати всі компоненти складної системи. Для системи управління локаціями створюються різні діаграми, які відображають статичну структуру, поведінку та взаємодії компонентів, що дозволяє отримати повне уявлення про систему.

Підсумовуючи, UML є універсальною мовою моделювання, яка відіграє важливу роль у процесі розробки програмного забезпечення для управління районними локаціями міста [2]. Вона використовує графічні позначення для створення абстрактних моделей, які називаються UML-діаграмами. Ці діаграми дозволяють зобразити систему з різних точок зору, включаючи її статичну

структуру, поведінку та взаємодії між компонентами [3]. UML не є мовою програмування, а інструментом для моделювання, який допомагає розробникам створювати чіткі та зрозумілі представлення системи, що сприяє ефективному проектуванню та реалізації.

1.3 Діаграма прецедентів

Діаграма прецедентів (use case diagram) є однією з діаграм UML, що застосовується на початковому етапі проектування інформаційної системи для моделювання її функціональності на концептуальному рівні [4]. Вона дозволяє чітко визначити взаємодію між зовнішніми учасниками (екторами) та системою, а також окреслити основні дії, які система виконує для досягнення певних результатів.

Основні елементи діаграми прецедентів включають:

- Межа системи – прямокутник, що позначає межі модельованої системи, відокремлюючи її від зовнішнього середовища. У UML 2 ця межа розглядається як суб’єкт або контекст системи.
- Ектор (actor) – елемент, що представляє ролі користувачів або зовнішніх систем, які взаємодіють із системою.
- Прецедент – елемент, що відображає конкретні дії або набір дій, які система виконує для досягнення результату, що має значення для ектора.
- Сценарій – конкретна послідовність взаємодій між ектором і системою, що описує окремий випадок використання.
- Відношення – зв’язки між екторами і прецедентами або між самими прецедентами, які відображають взаємодію чи залежності.

Діаграма прецедентів призначена для створення моделі функціонування системи в її оточенні. Вона включає екторів, прецеденти, обмежені межею системи, асоціації між екторами та прецедентами, а також відношення

включення (include) і розширення (extend) між прецедентами. Її головна мета – забезпечити спільне розуміння функціональності системи для клієнтів, користувачів і розробників.

При створенні діаграми прецедентів аналітик прагне:

- Чітко відокремити систему від її зовнішнього середовища.

- Визначити екторів, їхню взаємодію з системою та очікувані функції.

- Сформувати глосарій предметної області, що описує ключові поняття, пов’язані з функціоналом системи.

Діаграма починається з текстового опису функціональних вимог, при цьому нефункціональні вимоги (наприклад, продуктивність чи безпека) не включаються до моделі прецедентів і оформлюються окремо. Кожен прецедент описує дії, що виникають під час взаємодії ектора з системою, не деталізуючи, як саме ці дії реалізовані.

Між прецедентами можуть існувати залежності, які зображуються пунктирними стрілками. Основними типами залежностей є:

- Відношення включення (include) – вказує, що один прецедент використовує поведінку іншого як свою складову частину. Прецедент, що включається, не може використовуватися окремо і є абстрактним. На діаграмі це позначається пунктирною стрілкою зі стереотипом «include», спрямованою від базового прецеденту до включеного.

- Відношення розширення (extend) – використовується для моделювання необов’язкової поведінки системи, яка виконується за певних умов. Пунктирна стрілка зі стереотипом «extend» спрямована від розширюючого прецеденту до базового.

Опис екторів подано в таблиці 1.

Таблиця 1.1

Опис екторів

Ектор	Короткий опис
Користувач	Особа, яка переглядає локації, залишає коментарі, оцінює їх і спілкується з менеджерами.
Менеджер	Особа, яка керує даними про локації, модерує коментарі та відповідає користувачам.

Діаграма прецедентів (рис. 1.1) для системи управління районними локаціями міста включає такі основні прецеденти:

- Для користувача: перегляд районів і локацій, пошук локацій за параметрами, оцінка локацій, коментування, спілкування в чаті з менеджером.
- Для менеджера: створення, редагування, видалення локацій, модерація коментарів, відповіді користувачам у чаті.

Відношення включення застосовується, наприклад, до авторизації, яка є обов'язковою для оцінки чи коментування локацій. Відношення розширення може стосуватися необов'язкових дій, таких як модерація коментарів менеджером за певних умов (наприклад, при скаргах користувачів).

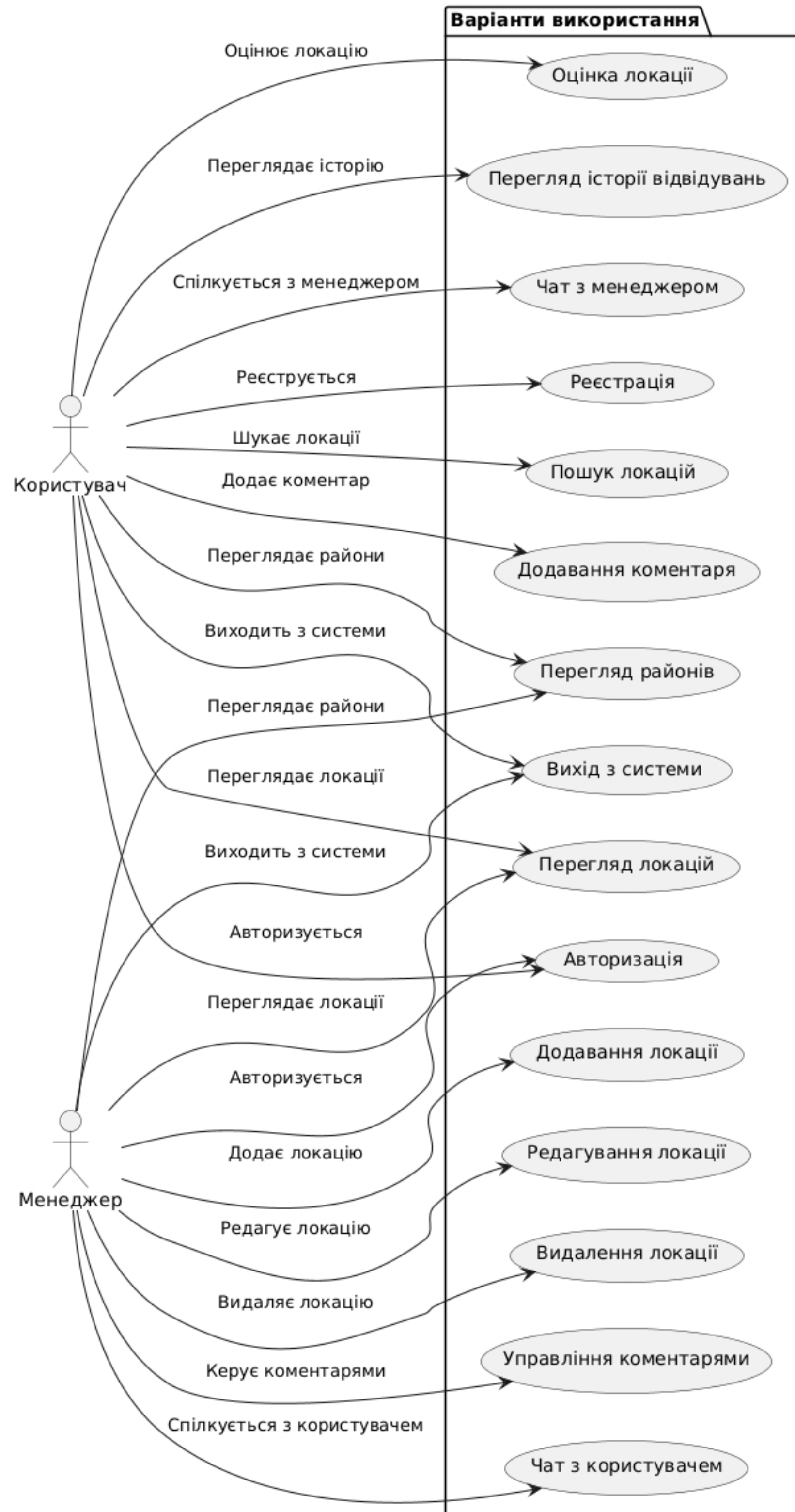

Рис. 1.1 Діаграма прецедентів

1.5 Діаграма послідовності

Діаграма послідовності (sequence diagram) є інструментом для графічного зображення порядку взаємодії об’єктів у системі управління районними локаціями міста протягом часу [6]. Вона відображає, як користувачі (актори, такі як звичайні користувачі або менеджери) взаємодіють із компонентами програми під час реалізації певних сценаріїв використання, а також як компоненти системи взаємодіють між собою. У контексті системи управління районними локаціями діаграма послідовності дозволяє формалізувати сценарії використання, такі як перегляд локацій, додавання коментарів чи оцінок, надсилання повідомлень у чаті тощо.

Діаграма допомагає на ранніх етапах розробки визначити всі взаємодіючі компоненти системи, такі як клієнтський інтерфейс (наприклад, веб-інтерфейс), сервер (описаний у файлі server.py), база даних (locations.db) та її таблиці (districts, locations, users, comments, ratings, messages, view_history). Вона також описує потоки повідомлень між цими компонентами, які згодом трансформуються у виклики методів об’єктів у коді. Це дозволяє створити модель подій, які підтримуються та обробляються системою.

На діаграмі послідовності зображуються лише ті об’єкти, які беруть безпосередню участь у взаємодії для реалізації конкретного сценарію використання, наприклад, авторизація користувача чи додавання нової локації менеджером. Зв’язки з іншими об’єктами, які не беруть участі у сценарії, не відображаються. Вертикальна вісь діаграми неявно представляє час, що дозволяє візуалізувати послідовність передачі повідомлень між об’єктами.

Лінія життя об’єкта (object lifeline) – це пунктирна вертикальна лінія, яка позначає період існування об’єкта в системі. Наприклад, об’єкт користувача (users), локації (locations) чи сервера існує протягом усього сценарію взаємодії. Лінія життя показує, коли об’єкт активний і готовий обробляти повідомлення.

Повідомлення – це сигнали, які ініціюють виконання певної дії об’єктом, що їх отримує. У системі управління районними локаціями повідомленнями є, наприклад, HTTP-запити до сервера (GET /api/locations, POST /api/auth/login) або SQL-запити до бази даних для отримання чи оновлення даних. Повідомлення

зображуються горизонтальними стрілками з назвою дії, наприклад, “отримати список локацій” чи “додати коментар”. Вони впорядковані за часом: повідомлення, розташовані вище на діаграмі, виконуються раніше за ті, що нижче.

Діаграма послідовності має два виміри. Перший – горизонтальний, де зліва направо розташовані лінії життя об’єктів, таких як клієнт (користувач або менеджер), сервер, база даних та її таблиці. Об’єкти можуть перебувати в стані активності (наприклад, сервер обробляє запит) або очікування (база даних чекає SQL-запит). Для позначення активності використовується фокус управління – прямокутник на лінії життя, який показує період, коли об’єкт виконує дію. Наприклад, при отриманні запиту POST /api/locations сервер активується для обробки даних і звернення до таблиці locations.

Другий вимір – вертикальна часова вісь, спрямована зверху вниз, де верхня частина діаграми відповідає початковому моменту часу. У системі управління районними локаціями взаємодія реалізується через повідомлення, такі як запит користувача на отримання даних про локацію, виклик сервером SQL-запиту до таблиці locations чи збереження оцінки в таблиці ratings. Масштаб часу на діаграмі не вказується, оскільки важлива лише відносна послідовність подій.

Наприклад, у сценарії “Додавання коментаря до локації” діаграма послідовності (рис. 1.3) включатиме такі об’єкти: користувач, клієнтський інтерфейс, сервер, база даних (таблиці users, locations, comments). Користувач надсилає повідомлення через інтерфейс (POST /api/locations/<id>/comments), сервер перевіряє авторизацію в таблиці users, підтверджує існування локації в таблиці locations, додає коментар до таблиці comments і повертає підтвердження клієнту. Лінії життя цих об’єктів тривають протягом усього сценарію, а фокуси управління відображають активність сервера та бази даних під час обробки запиту.

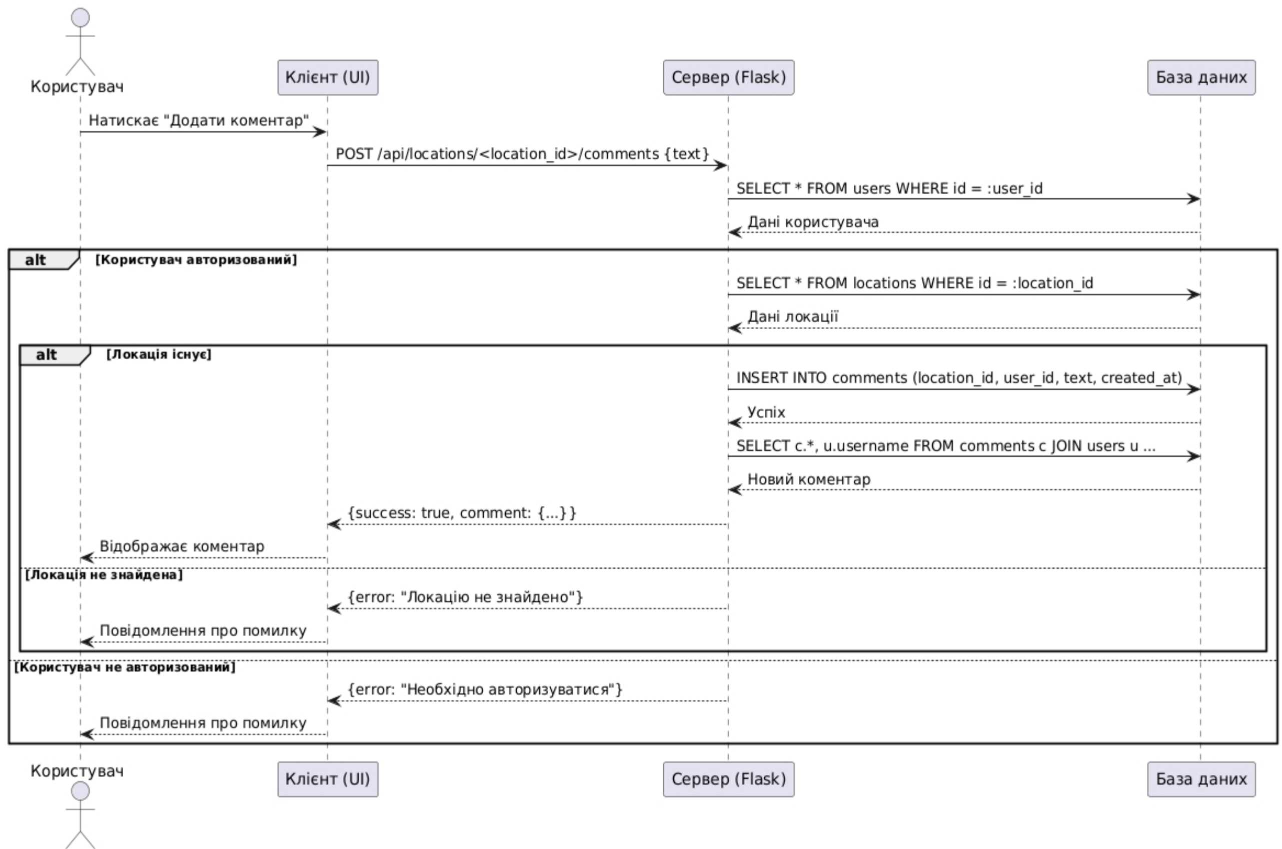

Рис. 1.3 Діаграма послідовності

1.6 Діаграма класів

Діаграма класів (рис. 1.4) є статичною UML-діаграмою, яка відображає структуру системи через класи, їх атрибути, методи та взаємозв'язки (асоціації, узагальнення, залежності). У контексті системи управління районними локаціями міста діаграма класів описує ключові об'єкти системи, такі як District, Location, User, Comment, Rating, Message та ViewHistory, а також їхні взаємодії. Вона забезпечує основу для реалізації серверної логіки та взаємодії з базою даних, допомагаючи розробникам зрозуміти структуру програмного коду

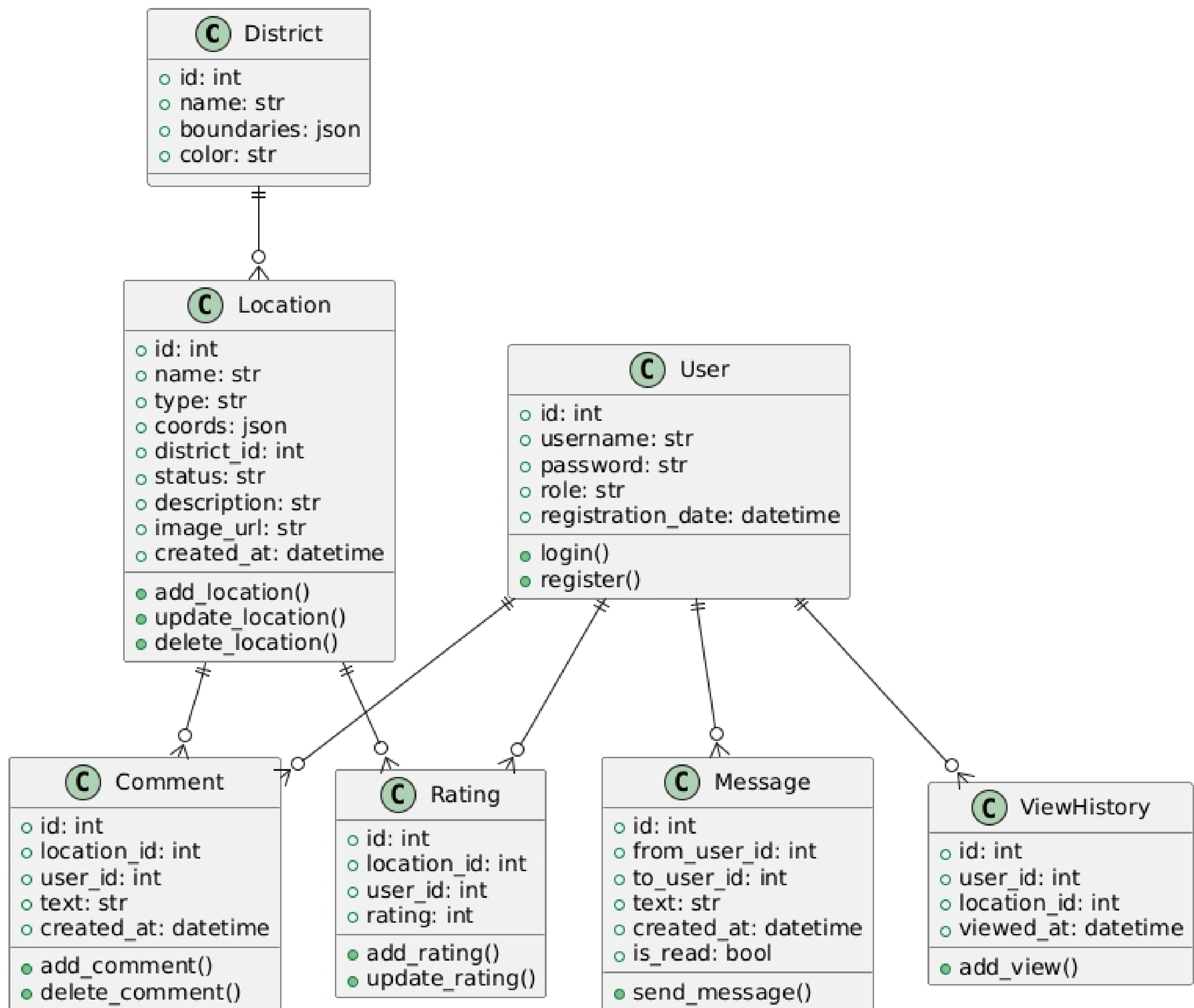

Рис. 1.4. Діаграма класів

1.6.1 Діаграма класів для кооперації: Додавання коментаря до локації

Діаграма класів для кооперації «Додавання коментаря до локації» (рис. 1.5) відображає класи, задіяні у процесі створення нового коментаря до локації користувачем. Вона включає класи User, Location, Comment та їхні взаємозв'язки, необхідні для виконання цієї операції. Діаграма показує, як користувач через метод add_comment() додає коментар, який пов'язується з певною локацією через атрибут location_id та з користувачем через user_id. Це забезпечує чітке розуміння структури та взаємодій для реалізації функціоналу коментування в системі

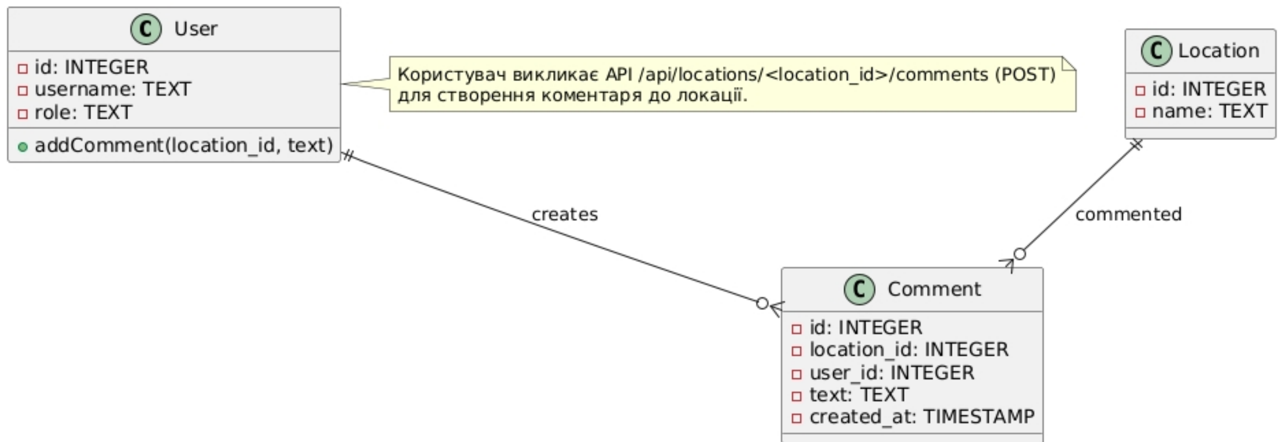

Рис. 1.5. Діаграма класів для кооперації «Додавання коментаря до локації»

1.6.2 Діаграма класів для кооперації: Оцінювання локації

Діаграма класів для кооперації «Оцінювання локації» (рис. 1.6) описує класи, які беруть участь у процесі виставлення оцінки локації користувачем. Вона включає класи User, Location, Rating та їхні зв'язки, необхідні для збереження оцінки. Клас User через метод add_rating() дозволяє користувачу оцінити локацію, яка пов'язана з оцінкою через location_id, а оцінка асоціюється з користувачем через user_id. Діаграма допомагає зрозуміти структуру даних для реалізації функціоналу оцінювання.

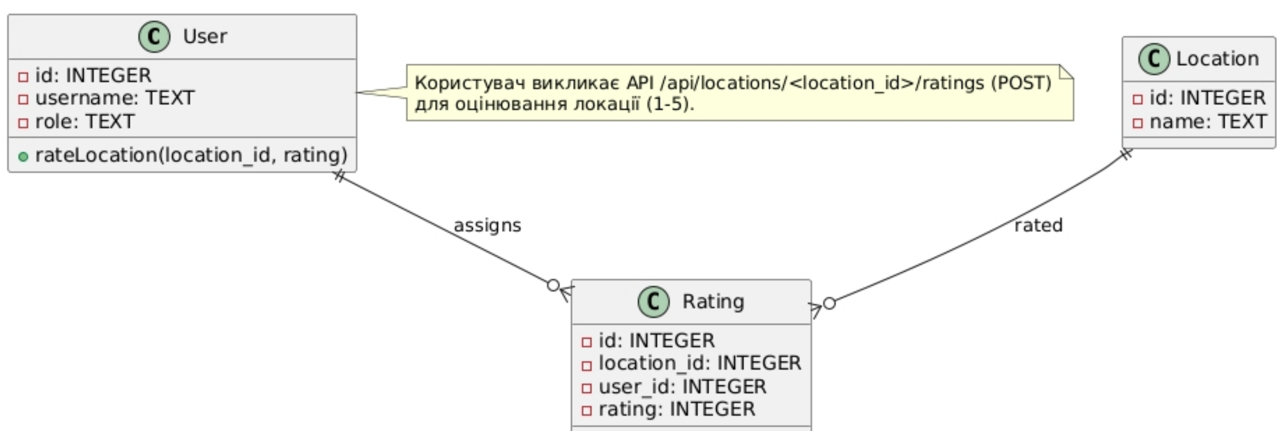

Рис. 1.6. Діаграма класів для кооперації «Оцінювання локації»

1.6.3 Діаграма класів для кооперації: Надсилання повідомлення в чаті

Діаграма класів для кооперації «Надсилання повідомлення в чаті» (рис. 1.7) відображає класи, задіяні у процесі надсилання повідомлення між користувачем і менеджером. Вона включає класи User, Message та їхні зв’язки, необхідні для реалізації чату. Клас User через метод send_message() дозволяє відправляти повідомлення, які пов’язуються з відправником через from_user_id та одержувачем через to_user_id. Діаграма ілюструє структуру для забезпечення комунікації в системі

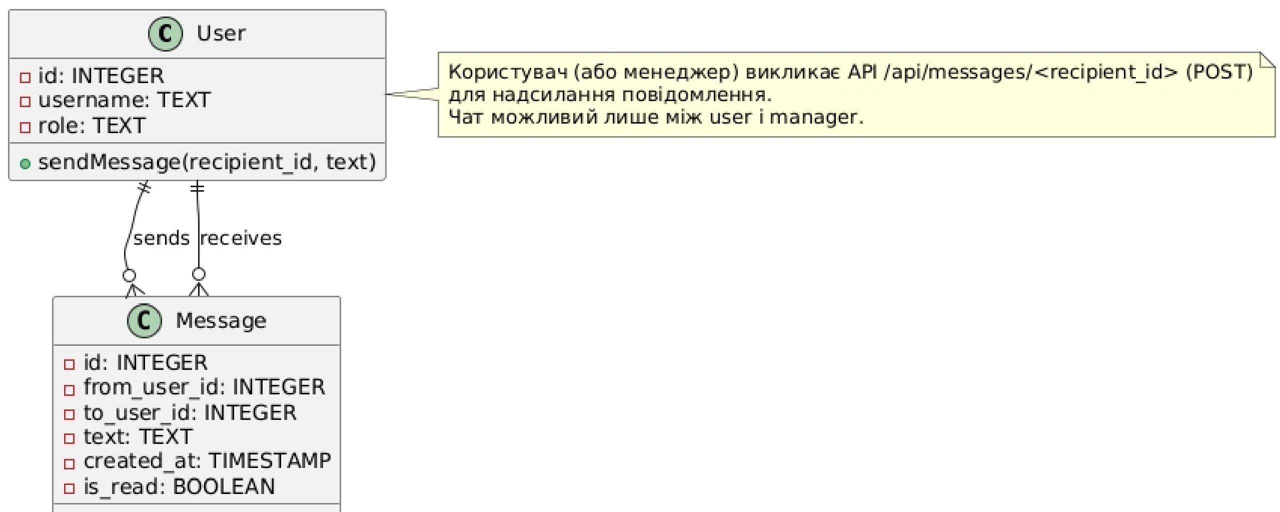

Рис. 1.7. Діаграма класів для кооперації «Надсилання повідомлення в чаті»

2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ

2.1 Загальні відомості про ER-діаграму

Перед початком розробки програмного забезпечення системи управління районними локаціями міста необхідно детально проаналізувати вимоги до системи. База даних має бути спроектованою таким чином, щоб вона відповідала потребам користувачів, була зрозумілою та не містила надлишкових даних. Для ефективного проектування бази даних використовуються семантичні моделі даних, серед яких найпоширенішою є ER-модель (Entity-Relationship model).

ER-модель – це інструмент для концептуального моделювання бази даних, який спрощує процес її створення [7]. Вона дозволяє графічно представити структуру даних у вигляді діаграми, що включає сутності, їх атрибути та зв’язки між ними. ER-модель є основою для створення реляційних баз даних, таких як SQLite, використана в даному проєкті, а також може бути адаптована до інших типів баз даних (ієрархічних, мережевих чи об’єктних) [8]. Основними елементами ER-моделі є поняття «сутність», «зв’язок» і «атрибут» [9].

Використання ER-моделі під час розробки системи управління районними локаціями дозволяє уникнути помилок у структурі бази даних, які можуть призвести до необхідності переробки програмного коду або ускладнень на етапі тестування чи експлуатації. Такі помилки можуть спричинити додаткові витрати часу, ресурсів і зусиль. ER-діаграма забезпечує наочне представлення предметної області, що охоплює райони міста, локації, користувачів, їх взаємодію та історію активності.

Сутність у контексті бази даних – це об’єкт предметної області, який має значення для системи, наприклад, район, локація, користувач чи коментар [10]. У системі управління районними локаціями розрізняють сильні та слабкі типи сутностей. Сильні сутності, такі як «districts» (райони) або «users» (користувачі), є незалежними та мають унікальні ідентифікатори. Слабкі сутності, наприклад,

«comments» (коментарі) або «ratings» (оцінки), залежать від сильних сутностей, таких як «locations» (локації) чи «users».

Кожна сутність має набір атрибутів, які описують її характеристики [11]. У системі можна виділити такі типи атрибутів:

- Прості атрибути складаються з одного компонента, наприклад, «name» (назва району чи локації) або «rating» (оцінка від 1 до 5).

- Складові атрибути об'єднують кілька простих, наприклад, «coords» (координати локації), що включає широту («lat») і довготу («lng»).

- Однозначні атрибути мають єдине значення, наприклад, «id» (унікальний ідентифікатор сутності).

- Багатозначні атрибути можуть містити кілька значень, наприклад, «boundaries» (межі району), що є списком координат.

У процесі розробки системи управління районними локаціями для реалізації бази даних обрано SQLite – легковагу реляційну СУБД, яка підтримує ефективну роботу з даними [12]. У реляційній моделі сутності представлені у вигляді таблиць, атрибути – як поля таблиць, а зв'язки – через зовнішні ключі. Наприклад:

- Прості атрибути, такі як «id» чи «name», реалізовані як цілочисельні (INTEGER) або текстові (TEXT) поля.

- Складові атрибути, наприклад, «coords» або «boundaries», зберігаються у вигляді JSON-рядків для зручності обробки.

- Однозначні атрибути, такі як «id», є первинними ключами (PRIMARY KEY), забезпечуючи унікальність записів.

- Зв'язки між сутностями, наприклад, між «locations» і «districts», реалізуються через зовнішні ключі (FOREIGN KEY).

Первинні ключі в таблицях, таких як «districts», «locations» чи «users», зазвичай мають цілочисельний тип (INTEGER), що забезпечує унікальність і

швидкий доступ до записів. Наприклад, атрибут «id» у таблиці «locations» є первинним ключем, який пов’язує локацію з коментарями та оцінками в таблицях «comments» і «ratings».

Таким чином, ER-модель системи управління районними локаціями забезпечує чітке концептуальне представлення структури даних, що дозволяє ефективно організувати інформацію про райони, локації, користувачів та їх взаємодію, мінімізуючи помилки на етапі проектування та реалізації.

2.2 Побудова ER- діаграми

Для побудови ER-діаграми системи управління районними локаціями міста було обрано програмне забезпечення PlantText. Програмний продукт PlantText є зручним онлайн-інструментом для створення діаграм, зокрема діаграм "сутність-зв’язок", за допомогою текстового опису на основі синтаксису PlantUML [13].

PlantText забезпечує просте та інтуїтивно зрозуміле створення діаграм шляхом написання текстових скриптів, які автоматично перетворюються на візуальні моделі. Цей інструмент дозволяє ефективно проектувати структури даних і є легким у використанні завдяки веб-інтерфейсу, що не потребує встановлення додаткового програмного забезпечення. PlantText підтримує створення як високорівневих (концептуальних), так і низькорівневих (фізичних) моделей даних. Високорівневі моделі використовуються на етапі концептуального проектування системи, тоді як низькорівневі моделі допомагають у реалізації бази даних та її тестуванні.

Основне призначення PlantText у контексті створення ER-діаграм полягає у швидкому генеруванні логічних і фізичних моделей даних на основі текстового опису. Інструмент дозволяє створювати діаграми "сутність-зв’язок" шляхом визначення сутностей, їхніх атрибутів і зв’язків у простому синтаксисі, який потім візуалізується у графічному вигляді.

Моделі даних, створені в PlantText, поділяються на два типи:

- Логічні моделі відображають структуру даних у термінах бізнес-об’єктів і використовуються для створення діаграм "сутність-зв’язок". У контексті системи управління районними локаціями міста логічна модель описує сутності, такі як райони, локації, користувачі, коментарі, оцінки, повідомлення та історія переглядів.

- Фізичні моделі слугують для реалізації бази даних у конкретній СУБД, враховуючи технічні особливості, такі як типи даних і обмеження цілісності, що відповідає структурі бази даних SQLite, використаної в проєкті.

Можливості PlantText у контексті проєкту:

- Візуалізація структур даних: PlantText дозволяє створювати чіткі та наочні діаграми шляхом написання текстового коду, що полегшує розуміння взаємозв’язків між сутностями, наприклад, зв’язок між локаціями та районами або між користувачами та їхніми коментарями.

- Використання стандартних елементів: Інструмент підтримує стандартний синтаксис PlantUML для опису сутностей і зв’язків, що сприяє підвищенню якості моделювання та уникненню дублювання даних, наприклад, при визначенні атрибутів для сутностей "districts" чи "locations".

- Порівняння моделей: PlantText дозволяє легко редагувати текстовий опис діаграми для внесення змін, що забезпечує можливість порівняння різних версій моделі та їхньої відповідності фізичній базі даних SQLite.

- Інтеграція з іншими інструментами: PlantText підтримує експорт діаграм у різні формати (PNG, SVG), що забезпечує можливість використання створених моделей для документування або інтеграції з іншими системами аналізу та проектування.

Таким чином, використання PlantText дозволило ефективно спроектувати ER-діаграму системи управління районними локаціями міста, забезпечивши простоту створення, редагування та візуалізації складних структур даних.

Розроблена ER-діаграма для системи управління районними локаціями міста включає такі основні сутності: "districts" (райони), "locations" (локації), "users" (користувачі), "comments" (коментарі), "ratings" (оцінки), "messages" (повідомлення) та "view_history" (історія переглядів). Кожна сутність має набір атрибутів і зв’язків, що відображають логіку роботи системи. Наприклад, сутність "locations" пов’язана з "districts" через атрибут district_id, а сутність "comments" пов’язана з "locations" і "users" через location_id та user_id відповідно.

На рис. 2.1 зображено модель бази даних, яка відображає структуру даних системи управління районними локаціями міста.

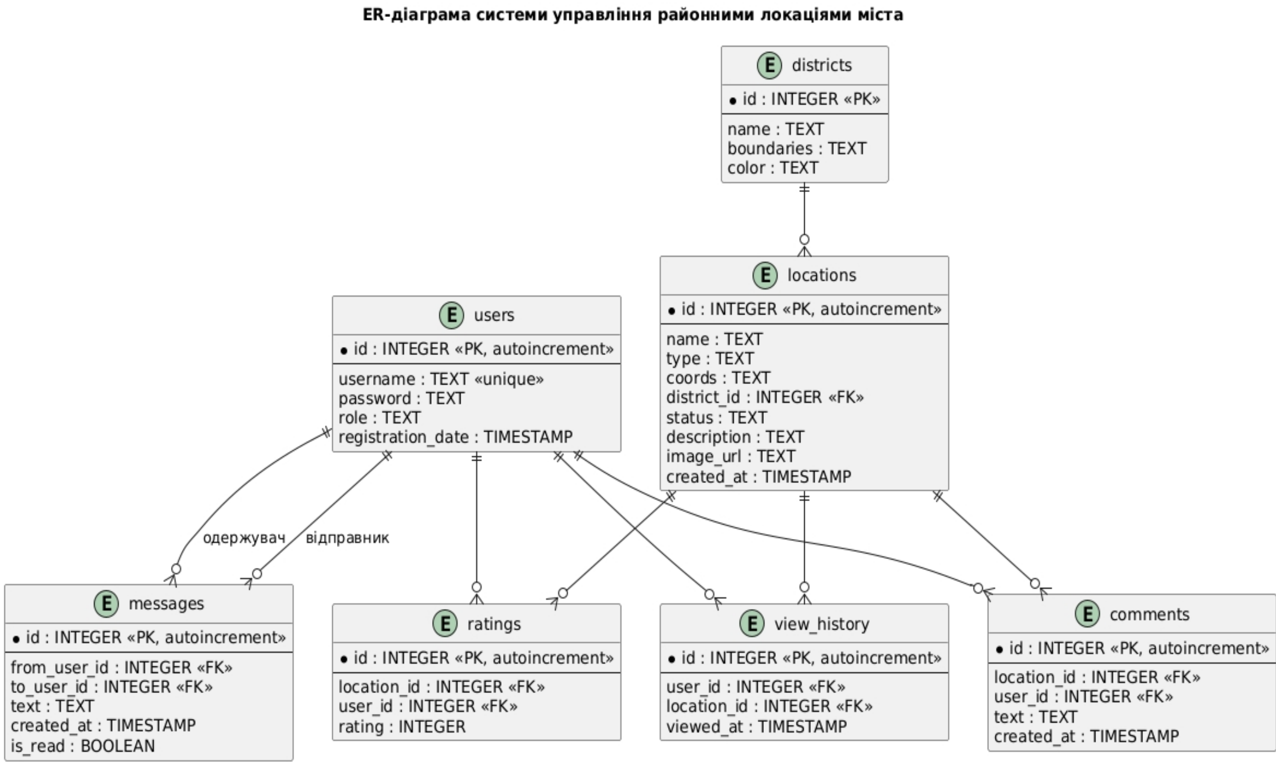

Рис. 2.1 ER-діаграма для системи управління районними локаціями міста

2.3 Вибір та обґрунтування СУБД

Бази даних є ключовим елементом сучасних інформаційних систем, забезпечуючи структуроване зберігання та обробку логічно пов’язаних даних для вирішення конкретних завдань. Система управління базами даних (СУБД) — це комплекс програмних і мовних засобів, що забезпечує централізоване

керування даними, їх створення, модифікацію, видалення, а також доступ до них [14]. Роль СУБД у забезпеченні ефективного зберігання, обробки та доступу до інформації постійно зростає, особливо в контексті систем управління районними локаціями міста, де потрібна швидка обробка даних, захист їх цілісності та підтримка багатокористувацької роботи.

Для вибору СУБД було сформульовано такі вимоги:

- забезпечення швидкого пошуку, модифікації та збереження даних;
- підтримка цілісності даних і захист від збоїв;
- можливість відновлення даних після апаратних або програмних помилок;
- забезпечення розмежування прав доступу та захист від несанкціонованого доступу;
- підтримка одночасної роботи кількох користувачів;
- мінімальні витрати на впровадження та підтримку;
- інтеграція з Python та веб-застосунками, що використовуються в проєкті.

Основною метою вибору СУБД було знайти систему, яка відповідає технічним вимогам проєкту при оптимальному рівні витрат, враховуючи апаратне забезпечення, програмну інтеграцію та легкість використання для розробників і кінцевих користувачів.

Для реалізації системи управління районними локаціями міста було обрано мову запитів SQL через її універсальність, простоту використання та широку підтримку в реляційних СУБД [15]. Реляційна модель даних є інтуїтивно зрозумілою, а SQL забезпечує зручний доступ до даних і їх обробку, що важливо для створення зрозумілої структури районів, локацій, коментарів, оцінок і повідомлень. SQL також добре інтегрується з Python через бібліотеки, такі як sqlite3, що використовується у проєкті для взаємодії з базою даних.

На ринку існує багато СУБД, які могли б потенційно підійти для реалізації проєкту. Слід розглянути найпопулярніші з них, оцінивши їх переваги та недоліки з точки зору потреб системи управління районними локаціями.

SQLite — це легка вбудована СУБД, яка не потребує окремого серверного процесу та зберігає всю базу даних у вигляді одного файлу [16]. Вона використовується у проєкті через її простоту, безкоштовність і легкість інтеграції з Python.

Переваги:

- Безкоштовна та з відкритим вихідним кодом, що знижує витрати на розробку.
- Легка в налаштуванні та використанні, не вимагає складного адміністрування.
- Підтримує основні можливості SQL, включаючи транзакції, індекси та обмеження цілісності.
- Компактна та ефективна для роботи з невеликими та середніми обсягами даних.
- Добре інтегрується з Python через стандартну бібліотеку sqlite3, що використовується у коді проєкту.
- Не потребує додаткового серверного обладнання, що ідеально для локальних або невеликих веб-застосунків.

Недоліки:

- Обмежена підтримка паралельної роботи кількох користувачів у порівнянні з серверними СУБД.
- Не підходить для високонавантажених систем із великою кількістю одночасних транзакцій.
- Відсутність вбудованих інструментів для масштабування на кілька серверів.

SQLite ідеально підходить для проєктів із помірними вимогами до масштабування, таких як система управління районними локаціями, де кількість одночасних користувачів обмежена, а обсяг даних не є надмірним.

MySQL — одна з найпопулярніших СУБД для веб-застосунків, що підтримує великі обсяги даних і багатокористувацький доступ [17].

Переваги:

- Безкоштовна у базовій версії з відкритим кодом.
- Висока продуктивність і підтримка паралельної роботи кількох користувачів.
- Широкий набір функцій, включаючи підтримку транзакцій і розподілених баз даних.
- Хороша документація та велика спільнота розробників.

Недоліки:

- Вимагає окремого серверного процесу, що ускладнює розгортання для невеликих проєктів.
- Потребує додаткового адміністрування порівняно з SQLite.
- Може бути надмірною для проєктів із невеликими базами даних.

MySQL більше підходить для веб-систем із високим навантаженням, але для локальної системи з обмеженою кількістю користувачів її можливості є надлишковими.

PostgreSQL — потужна безкоштовна СУБД із підтримкою складних запитів і великих обсягів даних [18].

Переваги:

- Безкоштовна та з відкритим кодом.
- Підтримує розширені можливості, такі як геопросторові дані та складні транзакції.
- Висока надійність і масштабованість.

- Хороша інтеграція з Python через бібліотеки, такі як psycopg2.

Недоліки:

- Складніша в налаштуванні та адмініструванні порівняно з SQLite.
- Вимагає більше ресурсів для розгортання.
- Надмірна для невеликих проєктів із простими вимогами до бази даних.

PostgreSQL краще підходить для складних систем із великими обсягами даних, але для даного проєкту її функціонал є надлишковим.

Microsoft SQL Server — комерційна СУБД із потужними можливостями для великих організацій [19].

Переваги:

- Висока продуктивність і підтримка складних сценаріїв обробки даних.
- Хороша інтеграція з продуктами Microsoft.
- Підтримка розподілених систем і паралельної обробки.

Недоліки:

- Висока вартість ліцензії, що робить її недоцільною для студентських проєктів.
- Складність розгортання для невеликих систем.
- Залежність від платформи Windows у більшості випадків.

Microsoft SQL Server більше підходить для корпоративних систем, а не для локальних веб-застосунків із помірними вимогами.

Для реалізації системи управління районними локаціями міста було обрано SQLite як основну СУБД. Цей вибір обґрунтовано її легкістю, безкоштовністю, простотою інтеграції з Python та достатньою функціональністю для виконання поставлених завдань. SQLite забезпечує ефективне зберігання даних про райони,

локації, користувачів, коментарі, оцінки та повідомлення, а також підтримує цілісність даних через зовнішні ключі, як видно у структурі бази даних у файлі seed.py. Вона не вимагає додаткового серверного обладнання, що спрощує розгортання системи, і є оптимальним рішенням для проєкту з обмеженим бюджетом і помірними вимогами до масштабування. Хоча SQLite має обмеження в паралельній обробці даних, для даної системи, де кількість одночасних користувачів невелика, це не є критичним. Таким чином, SQLite повністю відповідає потребам проєкту, забезпечуючи баланс між функціональністю, простотою та економічністю.

2.4 Створення БД

Для забезпечення функціонування системи управління районними локаціями міста було створено базу даних, яка підтримує наступні можливості:

- додавання даних до таблиць;
- створення та управління користувачами;
- збереження історії переглядів локацій;
- підтримка коментарів та оцінок локацій;
- забезпечення чату між користувачами та менеджерами.

База даних створена з використанням SQLite, що забезпечує легкість інтеграції з веб-додатком на основі Flask. Після ініціалізації бази даних створюються таблиці для зберігання інформації про райони, локації, користувачів, коментарі, оцінки, повідомлення та історію переглядів (рис. 2.2).

Структура | Данные | Ограничения | Индексы | Триггеры | DDL

locations Имя таблицы: locations ☐ WITHOUT ROWID ☐ STRICT

	Имя	Тип данных	Первичный ключ	Внешний ключ	Уникальность	Проверка	Не NULL	Сравнение	Сгенерированное	Значение по умолчанию
1	id	INTEGER								NULL
2	name	TEXT								NULL
3	type	TEXT								NULL
4	coords	TEXT								NULL
5	district_id	INTEGER								NULL
6	status	TEXT								NULL
7	description	TEXT								NULL
8	image_url	TEXT								NULL
9	created_at	TIMESTAMP								CURRENT_TIMESTAMP

Рис. 2.2 Структура таблиці

У процесі розробки створення таблиць здійснювалося двома способами:

- через виконання SQL-запитів у коді Python (файли seed.py та server.py);
- шляхом автоматичного створення при першому запуску системи, якщо база даних відсутня.

Після написання запиту він виконується через метод execute об'єкта курсора бази даних SQLite. Це дозволяє створити таблицю з унікальним ідентифікатором, назвою району, його межами (зберігаються у форматі JSON) та кольором для відображення на карті.

Для таблиці locations визначено додаткові атрибути, включаючи зовнішній ключ до таблиці districts (рис. 2.3)

```
CREATE TABLE IF NOT EXISTS locations (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    type TEXT NOT NULL,
    coords TEXT NOT NULL,
    district_id INTEGER NOT NULL,
    status TEXT NOT NULL,
    description TEXT,
    image_url TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (district_id) REFERENCES districts (id)
);
```

Рис. 2.3 Виконання SQL-запиту для створення таблиці

Після створення структури таблиці її необхідно зберегти в базі даних. У коді це реалізовано через виклик методу commit() після виконання запиту (рис. 2.4).

```
conn.commit()
conn.close()
print("Таблиці успішно створено")
```

Рис. 2.4 Збереження структури бази даних

Запити для створення всіх таблиць бази даних представлено у файлі seed.py (фрагменти наведено вище). Вони включають створення таблиць users, comments, ratings, messages та view_history з відповідними зв’язками через зовнішні ключі та обмеженнями, наприклад, унікальність комбінації location_id та user_id у таблиці ratings.

Таким чином, створення бази даних забезпечує надійне зберігання та управління даними системи, що є основою для реалізації всіх функціональних можливостей програмного забезпечення.

2.5 Додавання користувачів до БД

Додавання користувачів до бази даних реалізовано за допомогою Python-скрипту seed.py, який ініціалізує тестові дані для системи управління районними локаціями міста (рис. 2.5).

```
def add_users():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    # Додавання менеджерів
    managers = [
        {"username": "manager1", "password": "manager123"},
        {"username": "manager2", "password": "manager456"}
    ]
    for manager in managers:
        hashed_password = hashlib.sha256(manager["password"].encode()).hexdigest()
        registration_date = (datetime.now() - timedelta(days=random.randint(30,
90))).strftime("%Y-%m-%d %H:%M:%S")
        cursor.execute(
            "INSERT INTO users (username, password, role, registration_date) VALUES
(?, ?, ?, ?)",
            (manager["username"], hashed_password, "manager", registration_date)
        )

    # Додавання звичайних користувачів
    users = []
    for i in range(1, 11):
        users.append({
            "username": f"user{i}",
            "password": f"password{i}"
        })
    for user in users:
        hashed_password = hashlib.sha256(user["password"].encode()).hexdigest()
        registration_date = (datetime.now() - timedelta(days=random.randint(1,
60))).strftime("%Y-%m-%d %H:%M:%S")
        cursor.execute(
            "INSERT INTO users (username, password, role, registration_date) VALUES
(?, ?, ?, ?)",
            (user["username"], hashed_password, "user", registration_date)
        )

    conn.commit()
    conn.close()
    print(f"Додано {len(managers)} менеджерів та {len(users)} користувачів")
```

Рис.2.5 Фрагмент коду додавання користувачів до бази даних

Для створення таблиці користувачів у базі даних SQLite використовується SQL-запит у функції create_tables(). Таблиця users містить поля: id, username, password, role та registration_date. Додавання користувачів виконується у функції add_users(), яка створює двох менеджерів із фіксованими обліковими даними та

десять звичайних користувачів із динамічно згенерованими іменами та паролями. Паролі хешуються за допомогою алгоритму SHA-256 для забезпечення безпеки. Дата реєстрації генерується випадковим чином у межах 1–90 днів до поточного моменту.

Для налаштування зв’язків між таблицями використано SQL-запити з використанням зовнішніх ключів (KEY) (рис. 2.6).

Рис. 2.6 Ключові поля таблиці USER в табличній та кодовій формах

Схема зв’язків між таблицями, включаючи таблицю users, забезпечує цілісність даних, дозволяючи пов’язувати дії користувачів (коментарі, оцінки, повідомлення) з їхніми обліковими записами та локаціями (рис. 2.7).

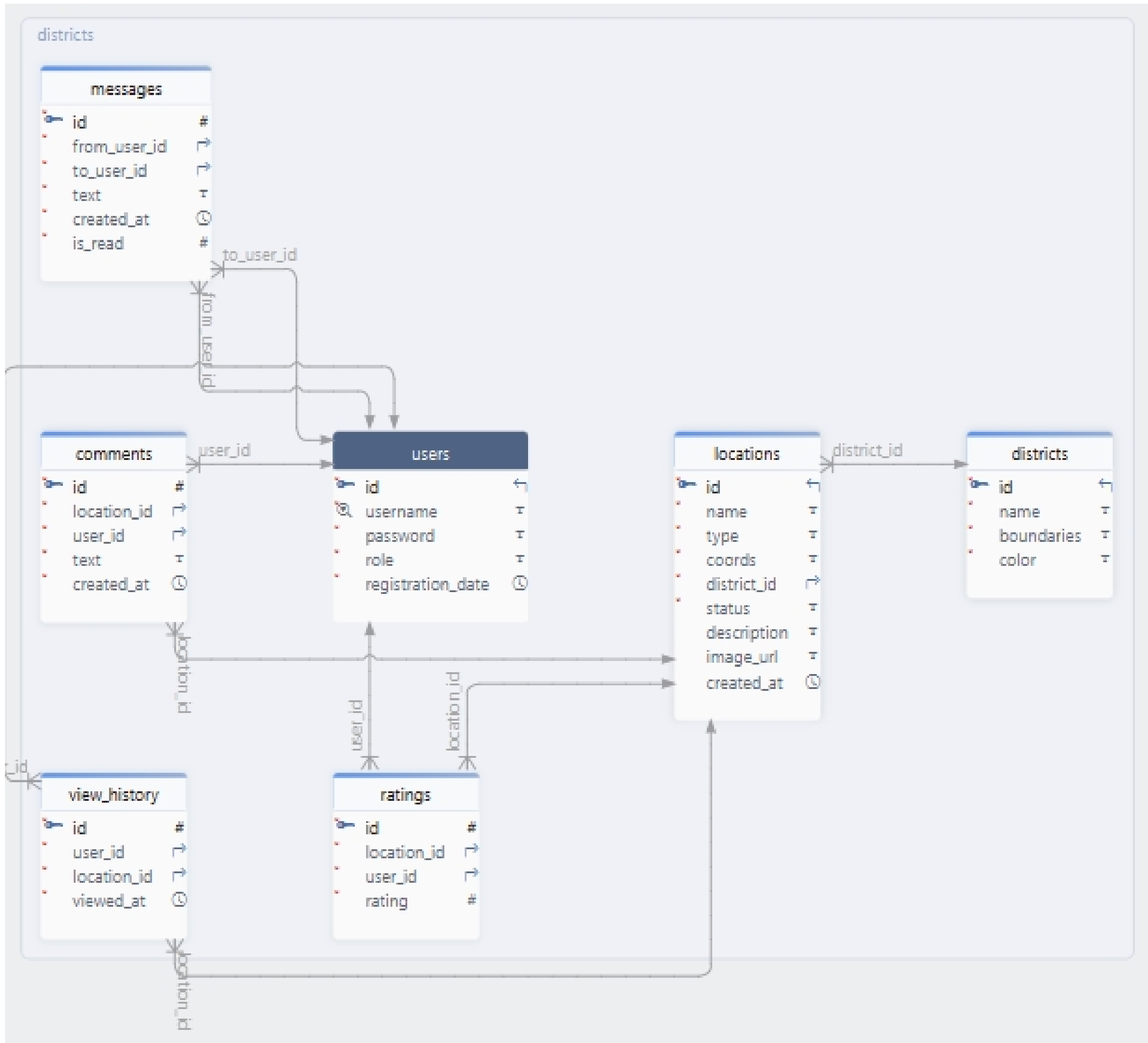

Рис. 2.7 Схема зв’язків між таблицями

2.6 Модель даних фізичного рівня

Модель даних фізичного рівня (рис. 2.8) описує структуру бази даних на рівні реалізації, включаючи назви таблиць, типи даних полів, обмеження (первинні та зовнішні ключі, унікальність) та індекси. Для системи управління районними локаціями міста фізична модель базується на SQLite і включає таблиці districts, locations, users, comments, ratings, messages та view_history з чітко визначеними типами даних і зв’язками. Вона забезпечує ефективне зберігання та обробку даних, відповідаючи вимогам системи.

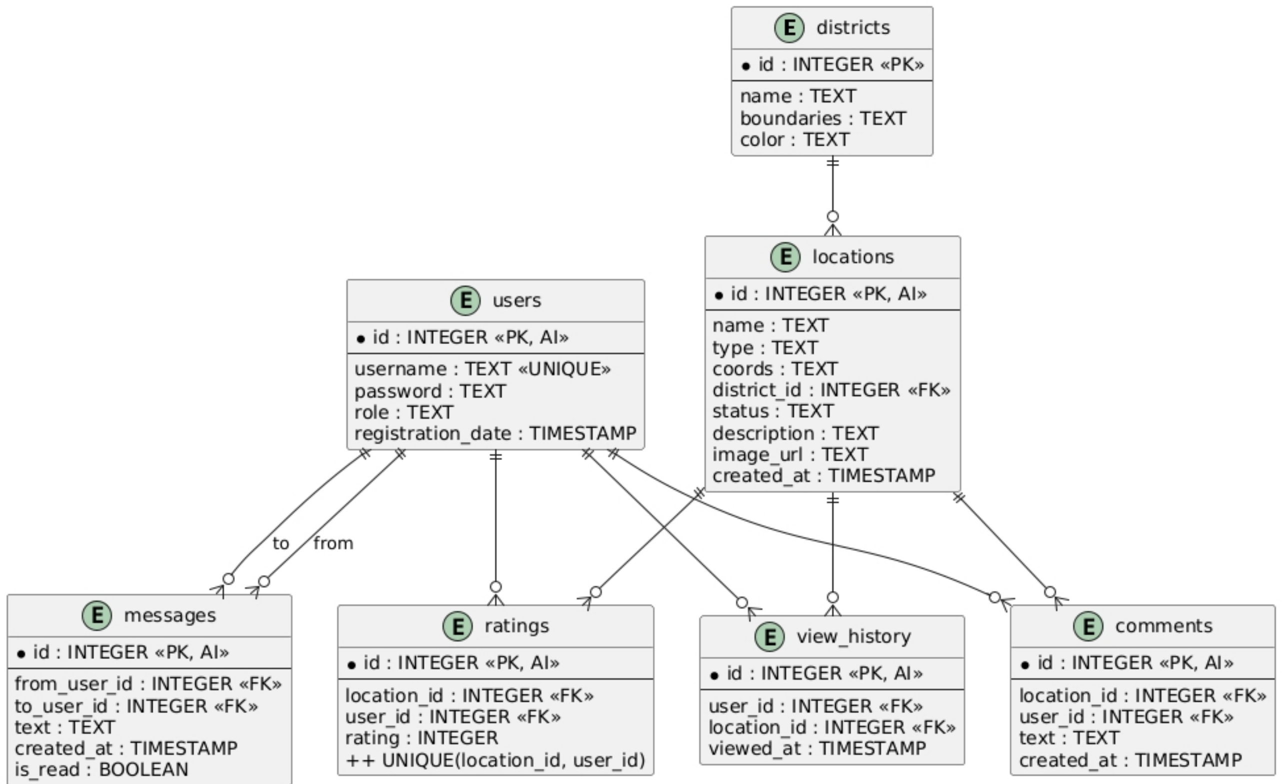

Рис. 2.8. Модель даних фізичного рівня

3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕПЕЧЕННЯ

3.1 Вибір інструментарію для розробки програмного забезпечення

Для розробки програмного забезпечення системи управління районними локаціями міста необхідно обрати оптимальну мову програмування та середовище розробки. Варто розглянути дві популярні мови програмування — Python і JavaScript, визначивши їх переваги та недоліки в контексті реалізації веб-додатку для управління локаціями.

Python — це високорівнева, інтерпретована мова програмування, яка широко використовується для розробки веб-додатків, завдяки своїй простоті та потужним бібліотекам [20].

Переваги мови Python:

- Простота синтаксису: Python має зрозумілий і лаконічний синтаксис, що полегшує розробку та підтримку коду, особливо для початківців.
- Багатий набір бібліотек: Наявність бібліотек, таких як Flask і Django, дозволяє швидко створювати веб-додатки з підтримкою баз даних, авторизації та API.
- Кросплатформність: Python підтримує різні операційні системи, що забезпечує гнучкість розгортання додатку.
- Інтеграція з базами даних: Python має зручні інструменти, такі як SQLite3 і SQLAlchemy, для роботи з базами даних, що є важливим для управління локаціями.
- Спільнота та документація: Велика спільнота розробників і детальна документація сприяють швидкому вирішенню проблем.

Недоліки мови Python:

- Продуктивність: Python є інтерпретованою мовою, що може призводити до нижчої швидкості виконання порівняно з компільованими мовами.

- Обмеження в клієнтській розробці: Python переважно використовується для серверної логіки, тому для створення інтерактивного інтерфейсу потрібні додаткові технології, такі як JavaScript.

- Розмір залежностей: Використання бібліотек може збільшувати розмір проєкту, що впливає на розгортання.

JavaScript — це мова програмування, яка використовується як для клієнтської, так і для серверної розробки веб-додатків, зокрема завдяки платформам, таким як Node.js [21].

Переваги мови JavaScript:

- Універсальність: JavaScript підтримує як клієнтську (React, Vue.js), так і серверну (Node.js) розробку, що дозволяє створювати повноцінні додатки в одній мові.

- Асинхронна обробка: Вбудована підтримка асинхронних операцій через async/await полегшує роботу з API та базами даних.

- Інтерактивність: JavaScript ідеально підходить для створення динамічних інтерфейсів, таких як карти з локаціями.

- Швидкість виконання: Node.js забезпечує високу продуктивність завдяки асинхронній моделі.

Недоліки мови JavaScript:

- Складність керування асинхронним кодом: Неправильне використання асинхронних конструкцій може призводити до складних помилок.

- Менш строга типізація: Відсутність суворої типізації може ускладнювати підтримку великих проєктів.

- Залежність від бібліотек: Для роботи з базами даних або серверною логікою потрібні додаткові модулі, такі як Express.js.

Висновок щодо вибору мови: для розробки системи управління районними локаціями міста обрано мову Python, оскільки вона забезпечує простоту розробки, зручну роботу з базами даних і швидке створення серверної логіки за допомогою фреймворку Flask. Python ідеально підходить для управління даними про локації, коментарі та рейтинги, а також для реалізації API.

Для розробки програмного забезпечення розглядалися два середовища: Microsoft Visual Studio та PyCharm.

Microsoft Visual Studio — це потужне інтегроване середовище розробки (IDE), яке підтримує кілька мов програмування, зокрема Python, і надає широкий набір інструментів для створення веб-додатків [22].

Переваги Microsoft Visual Studio:

- Інтеграція з Python: Visual Studio має спеціальний модуль для Python, який забезпечує автодоповнення коду, відлагодження та інтеграцію з віртуальними середовищами.
- Інструменти для веб-розробки: Вбудована підтримка Flask, API-тестування та інтеграція з Git спрощують розробку.
- Інтегрований відладник: Потужний відладчик дозволяє легко знаходити та виправляти помилки.
- Розширюваність: Наявність розширень, таких як Python Tools for Visual Studio, підвищує функціональність.
- Інтуїтивний інтерфейс: Зручний редактор коду з підтримкою IntelliSense полегшує написання та редагування коду.

Недоліки Microsoft Visual Studio:

- Ресурсомісткість: Visual Studio може бути важким для слабких комп’ютерів, що уповільнює розробку.

- Складність для новачків: Велика кількість функцій може бути надмірною для невеликих проєктів.

PyCharm — це спеціалізоване IDE для Python, розроблене компанією JetBrains, яке фокусується на зручності роботи з Python-проєктами [23].

Переваги PyCharm:

- Спеціалізація на Python: PyCharm оптимізовано для Python, що забезпечує кращу підтримку фреймворків, таких як Flask і Django.
- Інструменти для веб-розробки: Вбудована підтримка JavaScript, HTML і баз даних полегшує створення повноцінних додатків.
- Легкість налаштування: Швидке створення віртуальних середовищ і управління залежностями.

Недоліки PyCharm:

- Платна версія: Повна функціональність доступна лише в платній версії, хоча Community Edition підходить для базових завдань.
- Менша універсальність: PyCharm менш гнучкий для роботи з іншими мовами порівняно з Visual Studio.

Висновок щодо вибору середовища: Для розробки обрано Microsoft Visual Studio, оскільки воно забезпечує комплексний набір інструментів для роботи з Python, зручний відладчик і підтримку веб-розробки. Visual Studio дозволяє ефективно створювати, тестувати та розгортати систему управління районними локаціями, а також інтегрується з Git для контролю версій.

Додаткові інструменти:

- SQLite: Використано як легку базу даних для зберігання інформації про локації, райони, користувачів і коментарі [24].
- Flask: Фреймворк для створення серверної частини та API [25].
- Git: Для контролю версій і спільної роботи над проєктом [26].

Таким чином, обраний інструментарій — Python у середовищі Microsoft Visual Studio — є оптимальним для реалізації програмного забезпечення системи управління районними локаціями міста, забезпечуючи простоту, ефективність і гнучкість розробки.

3.2 Діаграма пакетів

Діаграма пакетів (рис. 3.1) є UML-діаграмою, яка відображає організацію системи у вигляді пакетів, що групують класи, модулі або інші елементи за функціональними чи логічними ознаками. Для системи управління районними локаціями міста діаграма пакетів показує розподіл функціональності між клієнтською частиною (веб-інтерфейс), серверною частиною (API та бізнес-логіка) та шаром даних (база даних). Вона допомагає зрозуміти модульну структуру системи та залежності між компонентами.

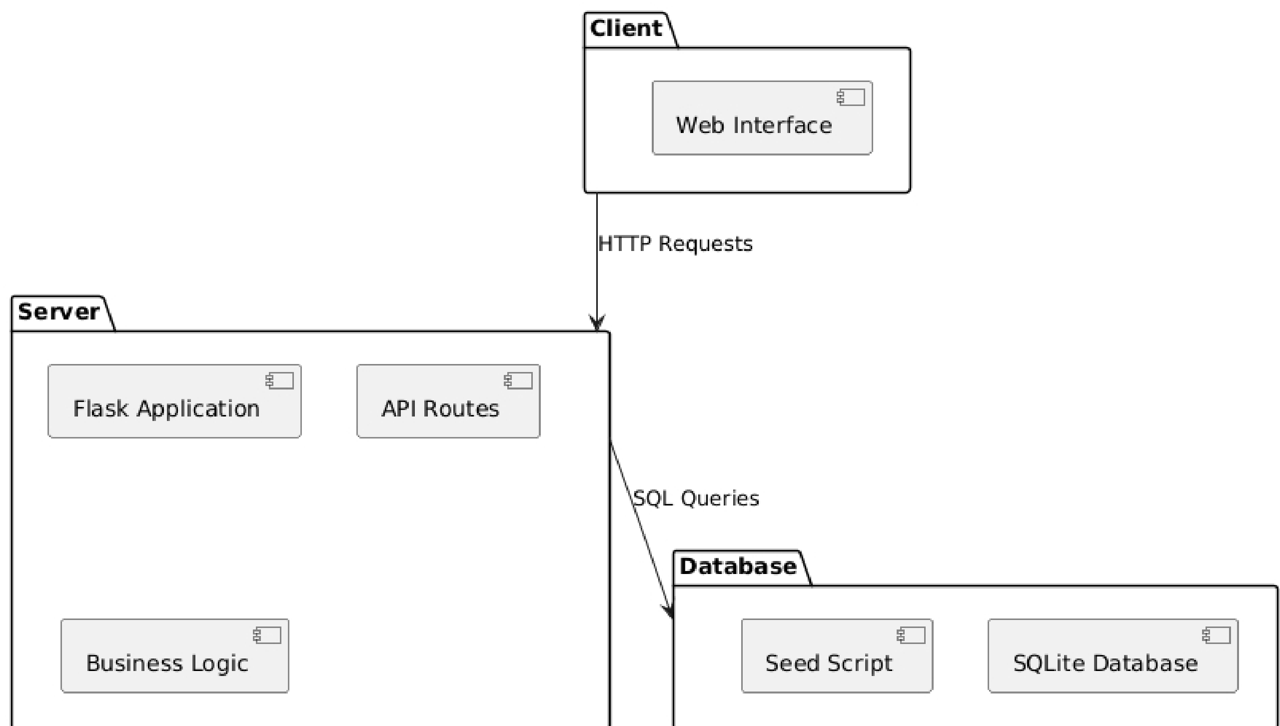

Рис.3.1. Діаграма пакетів

3.3 Принципи роботи у середовищі візуальної розробки програм

Для розробки програмного забезпечення системи управління районними локаціями міста було використано середовище Microsoft Visual Studio, яке надає широкі можливості для створення сучасних додатків. Це інтегроване середовище розробки (IDE) дозволяє проектувати графічний інтерфейс, редагувати програмний код, компілювати проекти, а також проводити покрокову відладку програмного забезпечення.

Microsoft Visual Studio забезпечує комплексний набір інструментів для розробки програмного забезпечення, включаючи підтримку різних мов програмування, таких як Python, який був використаний у даному проекті. Середовище дозволяє створювати як локальні, так і веб-додатки, що є важливим для реалізації системи управління локаціями з клієнт-серверною архітектурою. Інтерфейс Visual Studio відповідає стандартам Windows-програм, дозволяючи користувачу налаштовувати розмір і розташування вікон, згортати їх або переміщати для оптимізації робочого простору.

Основні елементи інтерфейсу Microsoft Visual Studio включають:

- Меню і панель інструментів, які надають швидкий доступ до команд для роботи з проектами, компіляції та налагодження.
- Оглядач рішень (Solution Explorer), де відображаються всі файли проекту, включаючи seed.py для ініціалізації бази даних та server.py для реалізації серверної частини.
- Редактор коду, який підтримує підсвітку синтаксису, автодоповнення та перевірку помилок у реальному часі, що полегшує написання коду на Python.
- Панель відладки, яка дозволяє встановлювати точки зупину, відстежувати значення змінних і аналізувати виконання програми покроково.

Microsoft Visual Studio базується на технології .NET Framework, яка забезпечує єдине середовище виконання та бібліотеки класів для створення надійних і масштабованих додатків. У контексті даного проекту Python використовувався разом із Flask для створення серверної частини, а SQLite — для управління базою даних. Visual Studio забезпечує інтеграцію з цими технологіями через розширення, такі як Python Tools for Visual Studio, що дозволяє зручно працювати з Python-проектами, запускати Flask-сервер і тестувати API-ендпоінти.

Середовище підтримує розробку клієнт-серверних додатків, що було ключовим для реалізації системи управління локаціями. Наприклад, у файлі server.py створено REST API з ендпоінтами для отримання даних про райони, локації, коментарі та оцінки. Visual Studio забезпечує зручне налаштування конфігурацій запуску, що дозволяє швидко тестувати сервер у режимі відладки. Крім того, вбудовані інструменти для роботи з базами даних спростили створення та тестування структури бази даних, визначеної у seed.py.

Переваги використання Microsoft Visual Studio

- Інтеграція з Python: Розширення для Python забезпечують повноцінну підтримку мови, включаючи автодоповнення, рефакторинг і профілювальники продуктивності.
- Інструменти відладки: Можливість покрокового виконання коду допомогла виявити та виправити помилки в логіці API-ендпоінтів і обробці запитів до бази даних.
- Управління проектами: Оглядач рішень дозволяє зручно організувати файли проекту, такі як скрипти для ініціалізації даних і серверної логіки.
- Підтримка веб-розробки: Інтеграція з Flask і інструменти для тестування HTTP-запитів спростили розробку і перевірку API.

Для реалізації прикладного програмного забезпечення системи управління районними локаціями міста було обрано Microsoft Visual Studio через його

потужні інструменти для розробки, зручний інтерфейс і підтримку сучасних технологій, що забезпечило ефективну реалізацію проекту.

4. ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Тестування системи

Відповідно до стандарту IEEE Std 829-1983, тестування є процесом аналізу програмного забезпечення, спрямованим на виявлення розбіжностей між фактичними та необхідними характеристиками, а також на оцінку властивостей програми. Такі розбіжності визначаються як дефекти.

Метою тестування системи управління районними локаціями міста є:

- виявлення та документування дефектів якості програмного продукту;
- надання рекомендацій щодо загальної якості системи;
- перевірка виконання ключових вимог до системи на конкретних прикладах;
- забезпечення відповідності функціональності системи запроектованим вимогам;
- підтвердження правильності реалізації вимог до системи.

Згідно з ГОСТ Р ІСО МЕК 12207-99, у життєвому циклі програмного забезпечення визначено процеси забезпечення якості, верифікації, валідації, спільної оцінки та аудиту. Діяльність із забезпечення якості включає інспекцію, верифікацію та валідацію програмного забезпечення.

Основною метою розробки системи є підвищення якості, зокрема її надійності, що є критично важливим для управління інформацією про районні локації міста. Надійність забезпечує стабільну роботу системи для користувачів і менеджерів, що взаємодіють із даними про локації, коментарі, оцінки та повідомлення.

Верифікація системи передбачає перевірку відповідності функціональності вимогам, сформульованим на етапах проєктування. Цей

процес включає аналіз коду, перевірку структури бази даних і тестування всіх компонентів системи. Наприклад, верифікувалася коректність збереження координат локацій у форматі JSON у базі даних SQLite.

Валідація системи полягала у перевірці відповідності її функціональності потребам кінцевих користувачів — звичайних користувачів, які переглядають локації, залишають коментарі та оцінки, а також менеджерів, які керують даними про локації.

Спільна оцінка була спрямована на аналіз стану проєкту, зокрема на контроль планування, використання ресурсів і відповідності розробки технічному завданню.

Аудит передбачав перевірку відповідності системи вимогам, зазначеним у технічному завданні, а також стандартам безпеки та якості даних.

Тестування системи управління районними локаціями міста є ключовим етапом розробки, під час якого перевірялася працездатність усіх компонентів і виявлялися можливі помилки. Мета тестування — отримання коректних результатів за заданими даними, забезпечення якості програми та безпомилкової роботи системи.

Сучасні методи тестування не завжди здатні виявити всі дефекти, особливо в системах із складною логікою, як-от управління локаціями з інтерактивними функціями (коментарі, оцінки, чат). Тому тестування проводилося в межах формального процесу верифікації, який підтверджує відсутність дефектів лише в рамках застосованих методів.

Тестування системи проводилося в середовищі розробки Visual Studio із використанням Python та фреймворку Flask. Воно охоплювало перевірку всіх ключових компонентів програми та мінімальний набір вимог, зокрема:

- коректність розподілу прав доступу між користувачами (звичайні користувачі можуть залишати коментарі та оцінки, менеджери — додавати, редагувати та видаляти локації);
- додавання нових облікових записів і призначення відповідних ролей (user або manager);

- режими додавання та редагування даних про локації з допустимими (правильні координати, назви) та недопустимими значеннями (порожні поля, некоректні формати);
- коректність роботи API-ендпоінтів для отримання, створення, оновлення та видалення даних;
- збереження та відновлення даних у базі SQLite, зокрема при некоректному завершенні роботи програми;
- коректність завершення сесії користувача при виході з системи (видалення даних із сесії).

Для тестування застосовувалися методи "білої скриньки" та "чорної скриньки".

Тестування "білої скриньки" передбачало доступ до вихідного коду системи. Розробники перевіряли логіку обробки запитів у файлі server.py, коректність SQL-запитів до бази даних і обробку JSON-даних (наприклад, координат локацій). Це дозволило переконатися в працездатності окремих компонентів, таких як функції авторизації, додавання коментарів і оцінок.

Тестування "чорної скриньки" проводилося через інтерфейс API та веб-інтерфейс, доступні користувачам. Тестувальники імітували дії користувачів, наприклад, надсилання POST-запитів для створення локацій, авторизацію з неправильними даними, залишення коментарів і оцінки. Це дозволило перевірити відповідність поведінки системи очікуванням користувачів, описаним у вимогах.

Додатково проводилися спеціалізовані види тестування:

- Тестування продуктивності: перевірялася здатність системи обробляти велику кількість запитів, наприклад, одночасне отримання даних про локації кількома користувачами. Для цього використовувалися тестові сценарії з файлу seed.py, який заповнював базу даних великою кількістю локацій, коментарів і оцінок.

- Тестування юзабіліті: оцінювалася зручність веб-інтерфейсу (index.html) для перегляду локацій і взаємодії з картою. Перевірялася інтуїтивність навігації та відображення районів із відповідними кольорами.

- Тестування безпеки: перевірялася захищеність системи від несанкціонованого доступу, зокрема спроби додавання локацій неавторизованими користувачами або доступу до даних чату між іншими користувачами. Використовувалася хеш-функція SHA-256 для паролів, а також перевірка ролей у сесіях.

Усі виявлені дефекти документувалися, аналізувалися та виправлялися на етапі розробки. Тестування підтвердило, що система управління районними локаціями міста відповідає вимогам щодо функціональності, безпеки та зручності використання, забезпечуючи надійну роботу для користувачів і менеджерів.

4.2 Апаратні та технічі засоби

Повний склад обчислювальної системи називається її конфігурацією. Конфігурацію поділяють на програмну і апаратну.

Всі сучасні комп’ютери мають блокову конструкцію. Узгоджене функціонування і взаємодія окремих блоків виконується за допомогою спеціальних пристроїв, які називають апаратними інтерфейсами. Стандарти на апаратні інтерфейси називають протоколами. Передача даних здійснюється через порти.

Діаграму розгортання системи зображено на рис. 4.1.

Рис. 4.1 Діаграма розгортання

Для роботи з системою управління районними локаціями міста необхідно мати сервер баз даних та персональні комп'ютери. На сервері баз даних розгорнута база даних SQLite, до якої підключено клієнтські комп'ютери через веб-інтерфейс. На комп'ютерах користувачів буде доступне програмне забезпечення через веб-браузер. Апаратні та програмні вимоги для сервера наведено відповідно у таблицях 4.1 та 4.2.

Таблиця 4.1

Апаратні вимоги для сервера

Ресурс	Мінімальний	Рекомендований
1	2	3
Процесор	1 ГГц	2.5 ГГц і вище
Оперативна пам'ять	1 Гб	4 Гб і вище
Жорсткий диск	50 Гб	200 Гб і вище
Привод CD (DVD)-ROM	12x	52x і вище

Продовження таблиці 4.1

1	2	3
Операційна система	Ubuntu 18.04, Windows Server 2016	Ubuntu 20.04, Windows Server 2019 або вище
Сервер баз даних	SQLite	SQLite або PostgreSQL
Канал Інтернета, корпоративної або локальної мережі	5 Мбіт/с	20 Мбіт/с і вище

Таблиця 4.2

Програмні вимоги для клієнтських комп’ютерів

Ресурс	Мінімальний	Рекомендований
Процесор	800 МГц	2 ГГц і вище
Оперативна пам’ять	512 Мб	2 Гб і вище
Жорсткий диск	20 Гб	100 Гб і вище
Канал Інтернета, корпоративної або локальної мережі	5 Мбіт/с	20 Мбіт/с і вище
Роздільність монітору	1024x768	1920x1080
Привод CD-ROM	4x	40x і вище
Відеокарта	256 Мб	512 Мб та вище

Для коректної роботи та підтримки злагодженості роботи розробленої системи управління районними локаціями міста було обране наступне програмне забезпечення серверної платформи, яке забезпечує оптимальну роботу системи.

На сервері встановлюється операційна система Ubuntu 20.04 (або Windows Server 2019 чи вище) та сервер баз даних SQLite. Для розробки серверної частини використано Python з фреймворком Flask, а для обробки запитів — веб-сервер, наприклад, Gunicorn.

На клієнтських комп’ютерах встановлюється операційна система Microsoft Windows 10 (або вище) чи будь-яка сучасна ОС з підтримкою веб-браузерів (Google Chrome, Mozilla Firefox тощо). Для розробки використано середовище Visual Studio, яке забезпечує зручну роботу з Python та JavaScript.

Для нормальної роботи програми (клієнтська частина) потрібно:

- IBM PC/AT сумісний ПЕОМ не нижче від Pentium IV, 1 ГГц з об'ємом ОЗУ не менше 512 Мб;

- вільного простору на жорсткому диску – не менше 20 Мб;
- стандартний SVGA монітор;
- клавіатура, миша.

Система не потребує додаткових програм для друку звітів, оскільки вся інформація відображається через веб-інтерфейс.

4.3 Опис роботи програми

На початку роботи з інформаційною системою управління районними локаціями міста перед користувачем відкривається головна сторінка веб-додатку, яка представлена на рис. 4.2. На цій сторінці відображається карта міста з позначеними районами та локаціями, а також панель навігації для авторизації або реєстрації. Користувач може увійти до системи як звичайний користувач або як менеджер, використовуючи форму авторизації.

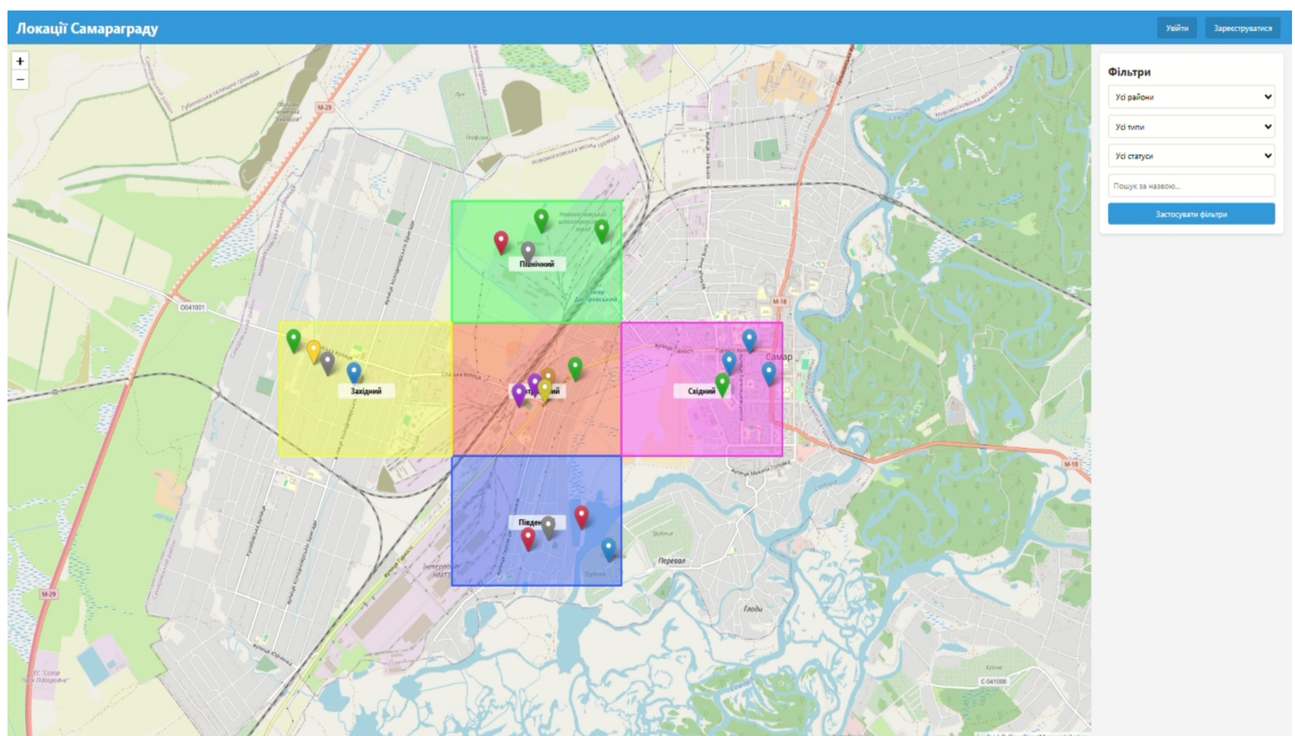

Рис. 4.2 Головна сторінка системи

Форма авторизації (рис. 4.3) пропонує користувачу ввести логін і пароль. Для ролей «Користувач» і «Менеджер» логіни та паролі різні. Після успішної авторизації користувач перенаправляється до відповідного інтерфейсу залежно від своєї ролі.

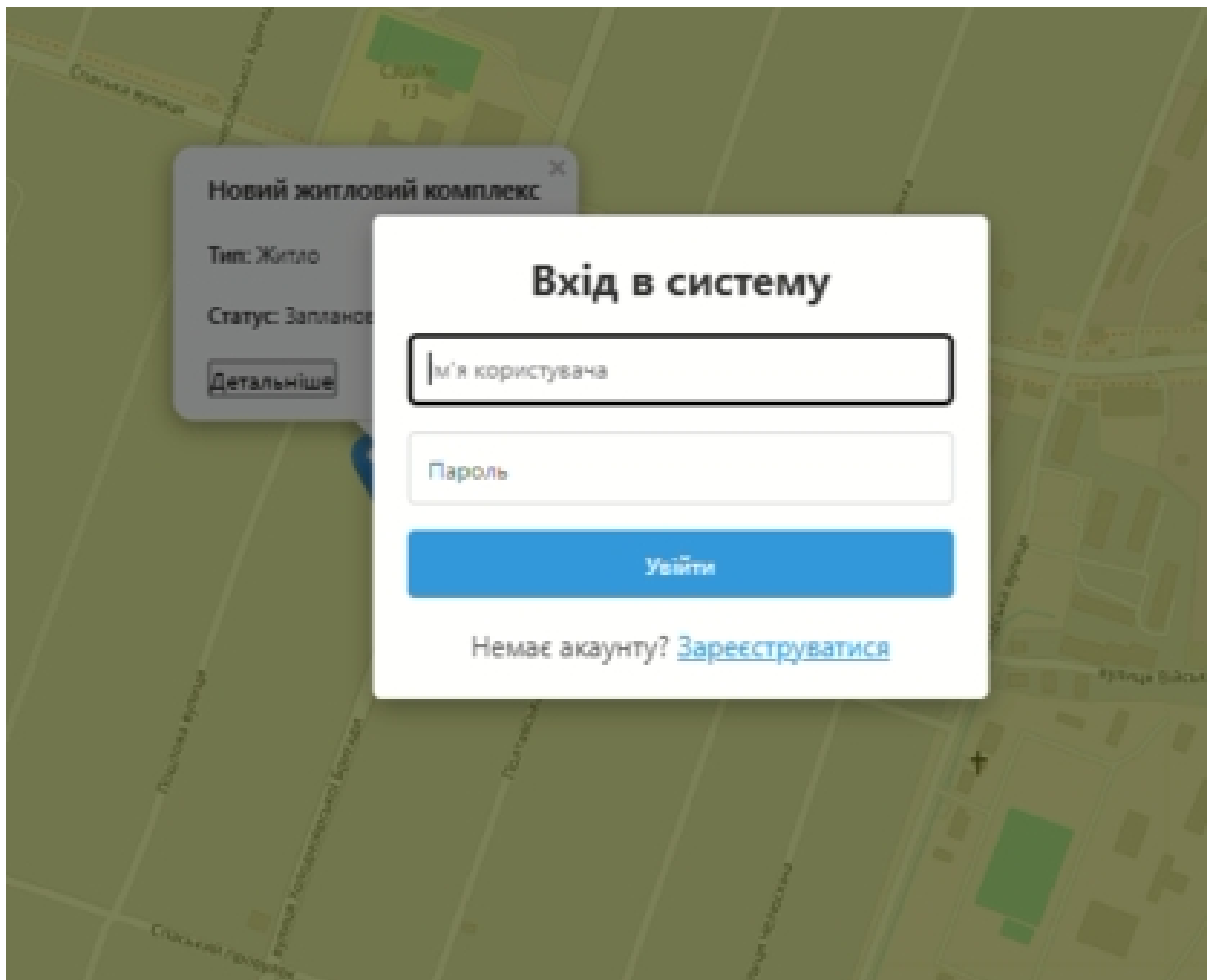

Рис. 4.3 Форма авторизації

Для звичайного користувача відкривається інтерфейс перегляду локацій (рис. 4.4), де можна вибрати район, тип локації (наприклад, парк, транспорт, магазин) або скористатися пошуком за назвою чи описом. На карті відображаються доступні локації з відповідними позначками.

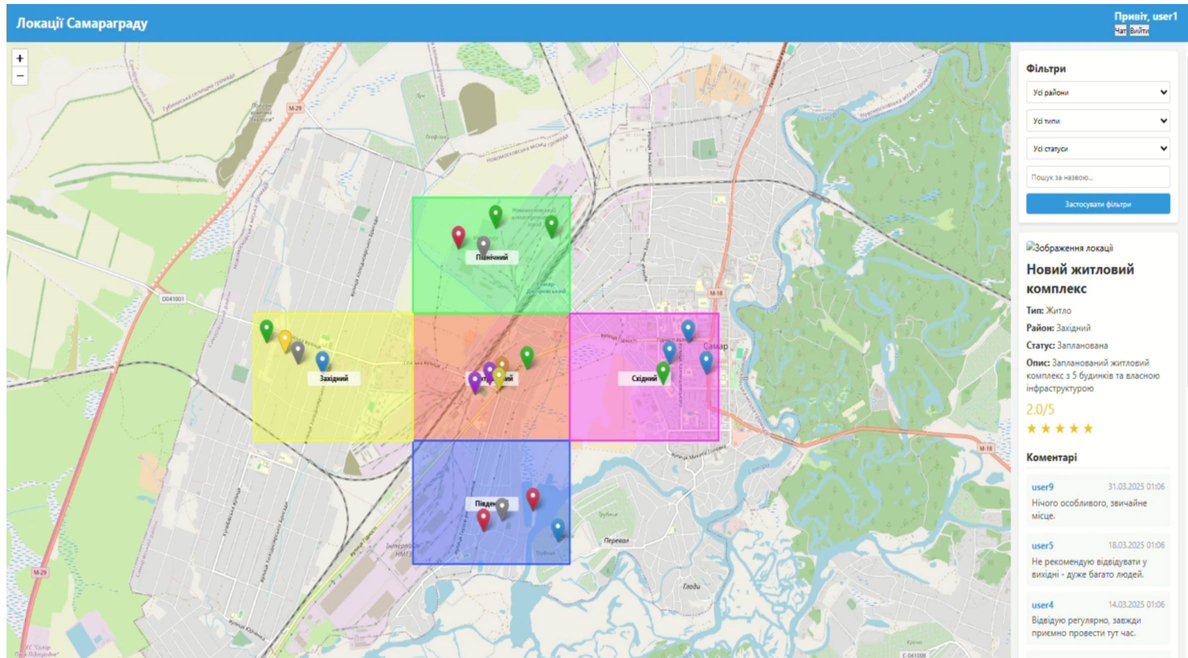

Рис. 4.4 Вікно перегляду локацій

При виборі конкретної локації відкривається її детальна сторінка (рис. 4.5), яка містить інформацію про назву, тип, координати, опис, статус, середню оцінку та коментарі інших користувачів. Користувач може залишити власний коментар або поставити оцінку від 1 до 5.

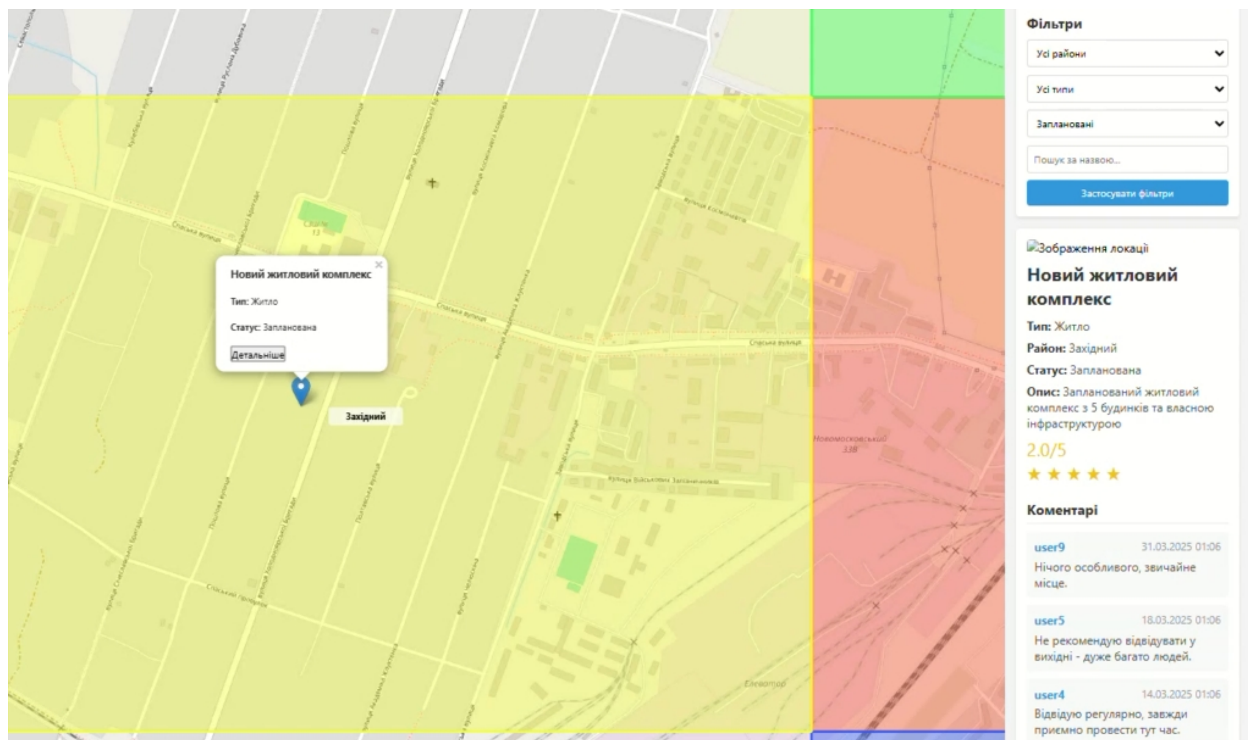

Рис. 4.5 Детальна сторінка локації

Користувач також має доступ до чату з менеджерами для отримання додаткової інформації про локації (рис. 4.6). У чаті відображаються надіслані та отримані повідомлення, а нові повідомлення позначаються як непрочитані до їх перегляду.

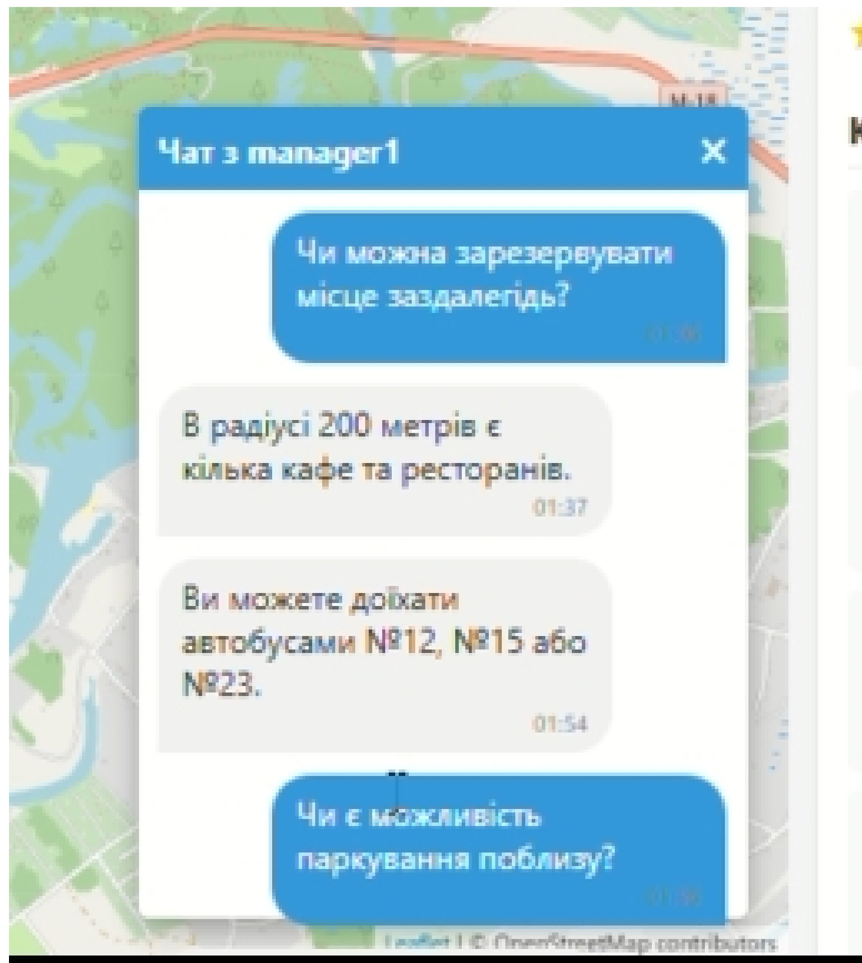

Рис. 4.6 Вікно чату з менеджером

Для менеджерів після авторизації відкривається панель управління локаціями (рис. 4.7), де вони можуть додавати нові локації, редагувати або видаляти існуючі. Форма додавання локації включає поля для введення назви, типу, координат, району, статусу та опису.

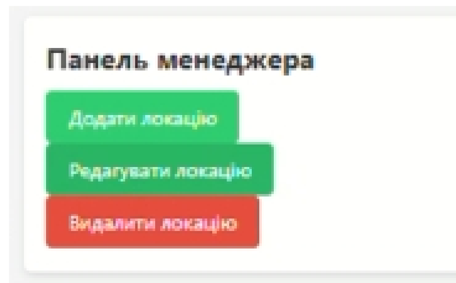

Рис. 4.7 Панель управління локаціями для менеджера

Менеджери також можуть переглядати коментарі користувачів до локацій (рис. 4.8) і видаляти некоректні записи за потреби. Це дозволяє підтримувати актуальність і якість інформації в системі.

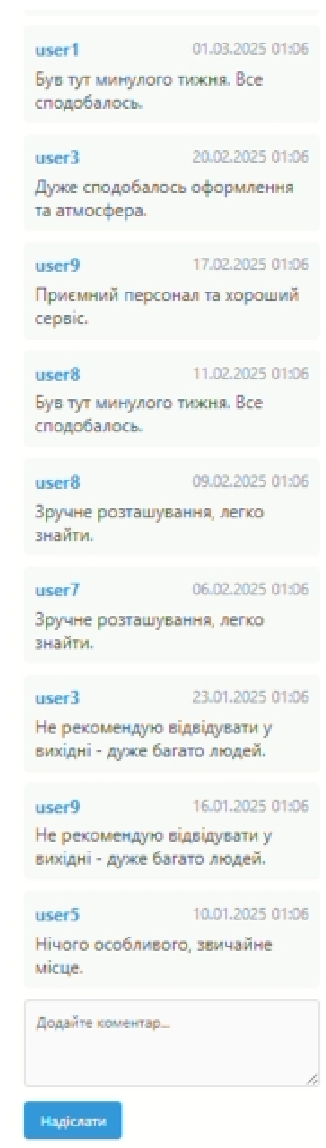

Рис. 4.8 Вікно перегляду коментарів

Для створення нової локації менеджер заповнює форму, представлену на рис. 4.9, після чого нова локація додається до бази даних і стає доступною для перегляду всіма користувачами.

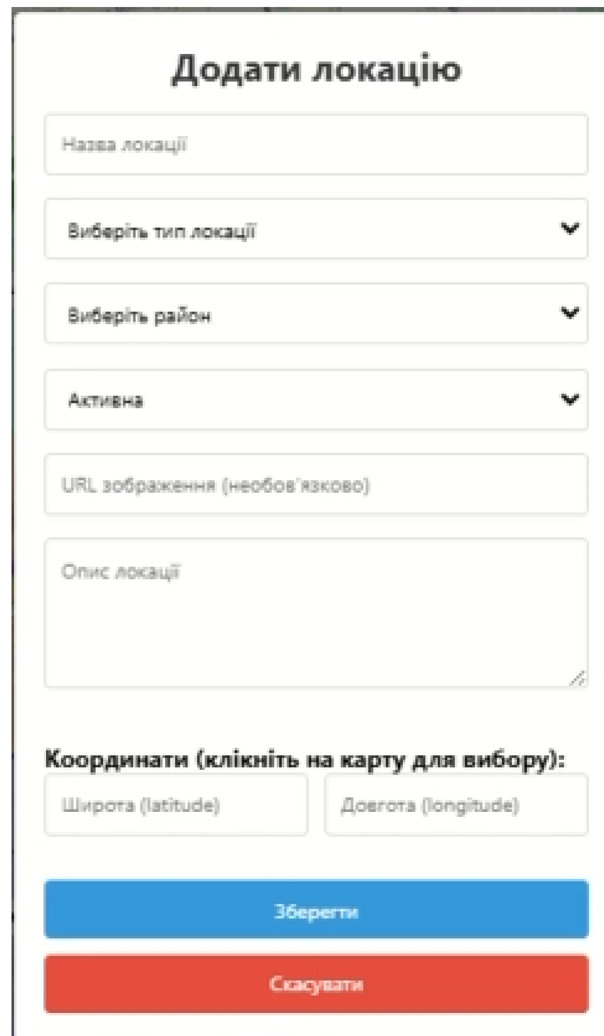

Рис. 4.9 Форма додавання нової локації

Нарешті, система дозволяє зберігати звіти про локації, наприклад, перелік локацій у певному районі або статистику оцінок (рис. 4.10). Ці звіти доступні менеджерам для аналізу популярності локацій.

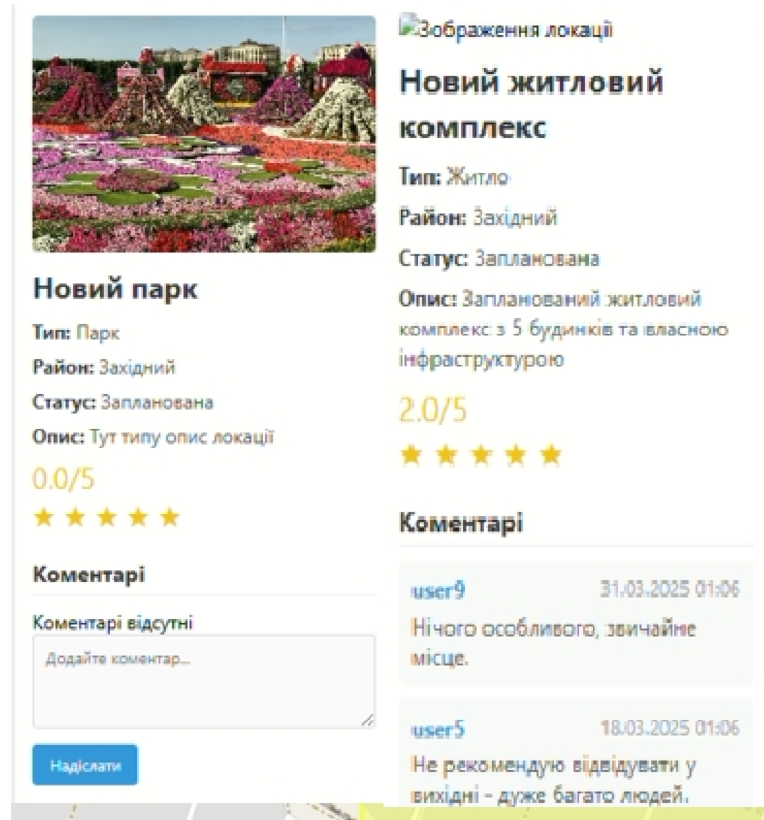

Рис. 4.10 Зразок звіту про локації в районі

Подальша робота з системою залежить від дій користувача чи менеджера, але всі операції, такі як перегляд, додавання, редагування чи видалення даних, виконуються через інтуїтивно зрозумілий веб-інтерфейс, розроблений у середовищі Visual Studio.

ВИСНОВКИ

У ході виконання бакалаврської кваліфікаційної роботи було розроблено програмне забезпечення системи управління районними локаціями міста. Програмне забезпечення створювалося відповідно до поставленого завдання. Була спроектована зручна, зрозуміла та функціональна база даних, яка забезпечує ефективне зберігання та обробку інформації про локації, райони, користувачів та їхню взаємодію.

Дана інформаційна система дозволяє користувачам переглядати та оцінювати локації, залишати коментарі, спілкуватися з менеджерами через чат, а також вести облік переглядів. Менеджерам система надає можливості для створення, редагування та видалення локацій, модерації коментарів і управління чатом з користувачами. Це забезпечує автоматизацію процесів управління міськими локаціями та покращує взаємодію між адміністрацією та мешканцями.

Для виконання роботи було обрано наступне програмне забезпечення:

- Для розробки бази даних використано SQLite на основі реляційних технологій. SQLite є легкою та ефективною системою управління базами даних, що ідеально підходить для проєктів такого масштабу.

- Для розробки серверної частини використано мову програмування Python у середовищі Visual Studio з використанням фреймворку Flask. Visual Studio є потужним і зручним середовищем для розробки, яке підтримує інтеграцію з базами даних, створення API та зручне налаштування проєктів. Основні переваги цього середовища включають:

  зручний інтерфейс для написання та налагодження коду;

  підтримка роботи з базами даних;

  можливість швидкої розробки веб-додатків.

У першому розділі бакалаврської кваліфікаційної роботи було сформульовано завдання на розробку, визначено вхідні та вихідні дані,

проведено аналіз об’єкта автоматизації — системи управління районними локаціями міста. Вхідними даними є інформація про локації (назва, тип, координати, статус, опис), райони (межі, колір), а також дії користувачів (коментарі, оцінки, повідомлення). Вихідними даними є списки локацій, деталі про них, коментарі, середні оцінки та повідомлення в чаті.

Було проведено моделювання системи та її системний аналіз. На цій стадії розробки визначено два основні режими роботи: для звичайних користувачів, які можуть переглядати локації, оцінювати їх і спілкуватися з менеджерами, та для менеджерів, які мають розширені права для управління даними.

Другий розділ присвячено вибору інформаційного забезпечення. Для роботи з базою даних обрано SQLite, яка забезпечує простоту інтеграції та ефективну роботу з даними. Для створення API використано мову Python і фреймворк Flask.

У третьому розділі описано розробку прикладного програмного забезпечення. Для серверної частини використано Python у середовищі Visual Studio, що дозволило реалізувати RESTful API для взаємодії з базою даних і клієнтською частиною.

У четвертому розділі розглянуто тестування програмного продукту. Було проведено перевірку коректності роботи API, обробки запитів, безпеки авторизації та цілісності даних. Описано сценарії використання системи, включаючи авторизацію, пошук локацій, додавання коментарів і роботу чату.

Результатом виконаної роботи став програмний продукт, який повністю реалізує поставлені функції управління районними локаціями міста, забезпечуючи зручність для користувачів і менеджерів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Процес моделювання предметної області. Гайд для бізнес-аналітиків [Електронний ресурс] – Режим доступу до ресурсу: https://dou.ua/forums/topic/42366/
2. Уніфікована мова моделювання (Unified Modeling Language - UML) [Електронний ресурс] – Режим доступу до ресурсу: https://www.maxzosim.com/unifikovana-mova-modeluvannia/
3. Для чого потрібні UML діаграми? [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/uml-diagramy/
4. Що таке діаграма варіантів використання UML: символи, шаблони, інструмент і посібник [Електронний ресурс] – Режим доступу до ресурсу: https://www.mindonmap.com/uk/blog/what-is-a-uml-use-case-diagram/
5. Діаграма діяльності в UML: символ, компоненти та приклад [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/uk/uml-activity-diagram.html
6. Діаграма послідовності (Sequence Diagrams) [Електронний ресурс] – Режим доступу до ресурсу: https://www.maxzosim.com/sequence-diagrams/
7. Що таке моделювання даних? Типи (концептуальний, логічний, фізичний) [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/uk/data-modelling-conceptual-logical.html
8. Модель діаграми зв’язків сутностей (ER) із прикладом СУБД [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/uk/er-diagram-tutorial-dbms.html
9. Модель сутність-зв’язок [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/10080855/page:6/
10. Моделювання даних (Data Modelling) [Електронний ресурс] – Режим доступу до ресурсу: https://www.maxzosim.com/data-modelling/
11. 4.3. Елементи моделі. Основні поняття er-діаграм [Електронний ресурс] – Режим доступу до ресурсу: https://studfile.net/preview/7075747/page:3/

12. What Is SQLite? [Електронний ресурс] – Режим доступу до ресурсу: https://sqlite.org/

13. About What Is PlantText? [Електронний ресурс] – Режим доступу до ресурсу: https://blog.planttext.com/about/

14. Що таке СУБД і для чого вони потрібні [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/systema-upravlinnia-bazamy-danykh/

15. Основи побудови SQL запитів [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/sql-zapyty/

16. Що таке SQlite? [Електронний ресурс] – Режим доступу до ресурсу: https://freehost.com.ua/ukr/faq/wiki/chto-takoe-sqlite/

17. Що таке MySQL [Електронний ресурс] – Режим доступу до ресурсу: https://freehost.com.ua/ukr/faq/wiki/chto-takoe-mysql/

18. Що таке PostgreSQL і для чого використовується? [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/postgresql-shcho-tse/

19. Microsoft SQL Server [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Microsoft_SQL_Server

20. Мова програмування Python та її застосування в різних галузях Server [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/de-vykorystovuietsia-python/

21. En Es De Fr JavaScript Підручник. Основи веб програмування [Електронний ресурс] – Режим доступу до ресурсу: https://w3schoolsua.github.io/js/index.html#gsc.tab=0

22. Microsoft Visual Studio: що це таке і для чого це потрібно [Електронний ресурс] – Режим доступу до ресурсу: https://macrosoft.store/uk/blog/post/29-для-чого-потрібна-microsoft-visual-studio

23. PyCharm: як встановити та використовувати для Python [Електронний ресурс] – Режим доступу до ресурсу: https://goit.global/ua/articles/pycharm-iak-vstanovyty-ta-vykorystovuvaty-dlia-python/

24. Введення в базу даних Sqlite [Електронний ресурс] – Режим доступу до ресурсу: http://yoip.com.ua/vvedennya-v-bazu-danih-sqlite/

25. Flask: Легкий у використанні веб-фреймворк та його реальні застосування [Електронний ресурс] – Режим доступу до ресурсу: https://javascript.org.ua/flask-legkij-u-vikoristanni-veb-frejmvork-ta-jogo-realni-zastosuvannya/

26. Git [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Git

ДОДАТОК А

База даних

Сторінок 3

Київ-2025

```
CREATE TABLE comments (
id INTEGER PRIMARY KEY AUTOINCREMENT,
location_id INTEGER NOT NULL,
user_id INTEGER NOT NULL,
text TEXT NOT NULL,
created_at TIMESTAMP NOT NULL,
FOREIGN KEY (
location_id
)
REFERENCES locations (id),
FOREIGN KEY (
user_id
)
REFERENCES users (id)
);



CREATE TABLE districts (
id INTEGER PRIMARY KEY,
name TEXT NOT NULL,
boundaries TEXT NOT NULL,
color TEXT NOT NULL
);



CREATE TABLE locations (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
type TEXT NOT NULL,
coords TEXT NOT NULL,
district_id INTEGER NOT NULL,
status TEXT NOT NULL,
description TEXT,
image_url TEXT,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY (
district_id
)
REFERENCES districts (id)
);
```

```
CREATE TABLE messages (
id INTEGER PRIMARY KEY AUTOINCREMENT,
from_user_id INTEGER NOT NULL,
to_user_id INTEGER NOT NULL,
text TEXT NOT NULL,
created_at TIMESTAMP NOT NULL,
is_read BOOLEAN NOT NULL
DEFAULT 0,
FOREIGN KEY (
from_user_id
)
REFERENCES users (id),
FOREIGN KEY (
to_user_id
)
REFERENCES users (id)
);
```

```
CREATE TABLE ratings (
id INTEGER PRIMARY KEY AUTOINCREMENT,
location_id INTEGER NOT NULL,
user_id INTEGER NOT NULL,
rating INTEGER NOT NULL
CHECK (rating BETWEEN 1 AND 5),
FOREIGN KEY (
location_id
)
REFERENCES locations (id),
FOREIGN KEY (
user_id
)
REFERENCES users (id),
UNIQUE (
location_id,
user_id
)
);
```

```
CREATE TABLE users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT UNIQUE
NOT NULL,
password TEXT NOT NULL,
role TEXT NOT NULL,
registration_date TIMESTAMP NOT NULL
);
```

```
CREATE TABLE view_history (
id INTEGER PRIMARY KEY AUTOINCREMENT,
user_id INTEGER NOT NULL,
location_id INTEGER NOT NULL,
viewed_at TIMESTAMP NOT NULL,
FOREIGN KEY (
user_id
)
REFERENCES users (id),
FOREIGN KEY (
location_id
)
REFERENCES locations (id)
);
```

ДОДАТОК Б

Код програми

Сторінок 20

Київ-2025

seed.py

```
import sqlite3
import json
import hashlib
import random
from datetime import datetime, timedelta
import os
# Шлях до бази даних
DB_PATH = "locations.db"
# Видалення існуючої бази даних, якщо потрібно створити нову
if os.path.exists(DB_PATH):
    os.remove(DB_PATH)
    print(f"Видалено існуючу базу даних {DB_PATH}")
# Функція для створення таблиць
def create_tables():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Таблиця районів
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS districts (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        boundaries TEXT NOT NULL,
        color TEXT NOT NULL
    )''')
    # Таблиця локацій
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS locations (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        type TEXT NOT NULL,
        coords TEXT NOT NULL,
        district_id INTEGER NOT NULL,
        status TEXT NOT NULL,
        description TEXT,
        image_url TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (district_id) REFERENCES districts (id)
    )''')
    # Таблиця користувачів
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        password TEXT NOT NULL,
        role TEXT NOT NULL,
        registration_date TIMESTAMP NOT NULL
    )''')
    # Таблиця коментарів
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS comments (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        location_id INTEGER NOT NULL,
        user_id INTEGER NOT NULL,
        text TEXT NOT NULL,
        created_at TIMESTAMP NOT NULL,
        FOREIGN KEY (location_id) REFERENCES locations (id),
        FOREIGN KEY (user_id) REFERENCES users (id)
    )''')
    # Таблиця оцінок
    cursor.execute('''
```

```
    CREATE TABLE IF NOT EXISTS ratings (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        location_id INTEGER NOT NULL,
        user_id INTEGER NOT NULL,
        rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
        FOREIGN KEY (location_id) REFERENCES locations (id),
        FOREIGN KEY (user_id) REFERENCES users (id),
        UNIQUE (location_id, user_id)
    )''')
    # Таблиця повідомлень для чату
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS messages (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        from_user_id INTEGER NOT NULL,
        to_user_id INTEGER NOT NULL,
        text TEXT NOT NULL,
        created_at TIMESTAMP NOT NULL,
        is_read BOOLEAN NOT NULL DEFAULT 0,
        FOREIGN KEY (from_user_id) REFERENCES users (id),
        FOREIGN KEY (to_user_id) REFERENCES users (id)
    )''')
    # Таблиця історії переглядів
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS view_history (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        user_id INTEGER NOT NULL,
        location_id INTEGER NOT NULL,
        viewed_at TIMESTAMP NOT NULL,
        FOREIGN KEY (user_id) REFERENCES users (id),
        FOREIGN KEY (location_id) REFERENCES locations (id)
    )''')
    conn.commit()
    conn.close()
    print("Таблиці успішно створено")
# Функція для додавання районів
def add_districts():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    districts = [
        {
            "id": 1,
            "name": "Центральний",
            "boundaries": [
                [48.6389, 35.2107],
                [48.6389, 35.2360],
                [48.6268, 35.2360],
                [48.6268, 35.2107]
            ],
            "color": "#FF5733"
        },
        {
            "id": 2,
            "name": "Північний",
            "boundaries": [
                [48.6500, 35.2107],
                [48.6500, 35.2360],
                [48.6389, 35.2360],
                [48.6389, 35.2107]
            ],
            "color": "#33FF57"
        },
        {
            "id": 3,
```

```
            "name": "Південний",
            "boundaries": [
                [48.6268, 35.2107],
                [48.6268, 35.2360],
                [48.6150, 35.2360],
                [48.6150, 35.2107]
            ],
            "color": "#3357FF"
        },
        {
            "id": 4,
            "name": "Західний",
            "boundaries": [
                [48.6389, 35.1850],
                [48.6389, 35.2107],
                [48.6268, 35.2107],
                [48.6268, 35.1850]
            ],
            "color": "#F3FF33"
        },
        {
            "id": 5,
            "name": "Східний",
            "boundaries": [
                [48.6389, 35.2360],
                [48.6389, 35.2600],
                [48.6268, 35.2600],
                [48.6268, 35.2360]
            ],
            "color": "#FF33F3"
        }
    ]
    for district in districts:
        cursor.execute(
            "INSERT INTO districts (id, name, boundaries, color) VALUES (?, ?, ?, ?)",
            (district['id'], district['name'], json.dumps(district['boundaries']),
district['color'])
        )
    conn.commit()
    conn.close()
    print(f"Додано {len(districts)} районів")
# Функція для додавання користувачів
def add_users():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Додавання менеджерів
    managers = [
        {"username": "manager1", "password": "manager123"},
        {"username": "manager2", "password": "manager456"}
    ]
    for manager in managers:
        hashed_password = hashlib.sha256(manager["password"].encode()).hexdigest()
        registration_date = (datetime.now() - timedelta(days=random.randint(30,
90))).strftime("%Y-%m-%d %H:%M:%S")
        cursor.execute(
            "INSERT INTO users (username, password, role, registration_date) VALUES (?, ?,
?, ?)",
            (manager["username"], hashed_password, "manager", registration_date)
        )
    # Додавання звичайних користувачів
    users = []
    for i in range(1, 11):
        users.append({
            "username": f"user{i}",
```

```
            "password": f"password{i}"
        })
    for user in users:
        hashed_password = hashlib.sha256(user["password"].encode()).hexdigest()
        registration_date = (datetime.now() - timedelta(days=random.randint(1, 
60))).strftime("%Y-%m-%d %H:%M:%S")
        cursor.execute(
            "INSERT INTO users (username, password, role, registration_date) VALUES (?, ?, 
?, ?)",
            (user["username"], hashed_password, "user", registration_date)
        )
    conn.commit()
    conn.close()
    print(f"Додано {len(managers)} менеджерів та {len(users)} користувачів")
# Функція для додавання локацій
def add_locations():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    locations = [
        # Центральний район (id=1)
        {
            "name": "Троїцький собор",
            "type": "landmark",
            "coords": {"lat": 48.6311, "lng": 35.2206},
            "district_id": 1,
            "status": "active",
            "description": "Історична дерев'яна церква, пам'ятка архітектури національного 
значення"
        },
        {
            "name": "Парк імені Сучкова",
            "type": "park",
            "coords": {"lat": 48.6335, "lng": 35.2290},
            "district_id": 1,
            "status": "active",
            "description": "Центральний міський парк відпочинку"
        },
        {
            "name": "Центральна площа",
            "type": "square",
            "coords": {"lat": 48.6325, "lng": 35.2250},
            "district_id": 1,
            "status": "active",
            "description": "Головна площа міста з фонтаном та міською адміністрацією"
        },
        {
            "name": "Міська бібліотека",
            "type": "landmark",
            "coords": {"lat": 48.6320, "lng": 35.2230},
            "district_id": 1,
            "status": "active",
            "description": "Центральна міська бібліотека з великим архівом книг"
        },
        {
            "name": "Кінотеатр 'Зірка'",
            "type": "entertainment",
            "coords": {"lat": 48.6315, "lng": 35.2245},
            "district_id": 1,
            "status": "active",
            "description": "Міський кінотеатр з трьома залами"
        },
        # Північний район (id=2)
        {
            "name": "Автовокзал",
```

```
            "type": "transport",
            "coords": {"lat": 48.6450, "lng": 35.2180},
            "district_id": 2,
            "status": "active",
            "description": "Головний автовокзал міста"
        },
        {
            "name": "Супермаркет 'Велика Кишеня'",
            "type": "shopping",
            "coords": {"lat": 48.6440, "lng": 35.2220},
            "district_id": 2,
            "status": "active",
            "description": "Великий супермаркет з широким асортиментом товарів"
        },
        {
            "name": "Парк Перемоги",
            "type": "park",
            "coords": {"lat": 48.6470, "lng": 35.2240},
            "district_id": 2,
            "status": "active",
            "description": "Парк з меморіалом воїнам, які загинули у Другій світовій війні"
        },
        {
            "name": "Спортивний комплекс 'Метеор'",
            "type": "sport",
            "coords": {"lat": 48.6460, "lng": 35.2330},
            "district_id": 2,
            "status": "active",
            "description": "Спортивний комплекс з басейном, тренажерним залом та ігровими
майданчиками"
        },
        # Південний район (id=3)
        {
            "name": "Залізничний вокзал",
            "type": "transport",
            "coords": {"lat": 48.6200, "lng": 35.2300},
            "district_id": 3,
            "status": "active",
            "description": "Залізнична станція Новомосковськ"
        },
        {
            "name": "Ринок 'Південний'",
            "type": "shopping",
            "coords": {"lat": 48.6190, "lng": 35.2250},
            "district_id": 3,
            "status": "active",
            "description": "Великий ринок з продуктами харчування та промисловими товарами"
        },
        {
            "name": "Лікарня №2",
            "type": "healthcare",
            "coords": {"lat": 48.6180, "lng": 35.2220},
            "district_id": 3,
            "status": "active",
            "description": "Міська лікарня з поліклінікою та стаціонаром"
        },
        {
            "name": "Школа №8",
            "type": "education",
            "coords": {"lat": 48.6170, "lng": 35.2340},
            "district_id": 3,
            "status": "active",
            "description": "Загальноосвітня школа I-III ступенів"
        },
```

```
        # Західний район (id=4)
        {
            "name": "Парк 'Західний'",
            "type": "park",
            "coords": {"lat": 48.6360, "lng": 35.1870},
            "district_id": 4,
            "status": "active",
            "description": "Парк відпочинку з озером та зоною для пікніка"
        },
        {
            "name": "Торговий центр 'Захід'",
            "type": "shopping",
            "coords": {"lat": 48.6340, "lng": 35.1920},
            "district_id": 4,
            "status": "active",
            "description": "Багатофункціональний торговий центр з магазинами та
розважальними закладами"
        },
        {
            "name": "Дитячий майданчик 'Казка'",
            "type": "playground",
            "coords": {"lat": 48.6350, "lng": 35.1900},
            "district_id": 4,
            "status": "active",
            "description": "Сучасний дитячий майданчик з різними атракціонами"
        },
        {
            "name": "Новий житловий комплекс",
            "type": "residential",
            "coords": {"lat": 48.6330, "lng": 35.1960},
            "district_id": 4,
            "status": "planned",
            "description": "Запланований житловий комплекс з 5 будинків та власною
інфраструктурою"
        },
        # Східний район (id=5)
        {
            "name": "Промисловий парк",
            "type": "industrial",
            "coords": {"lat": 48.6360, "lng": 35.2550},
            "district_id": 5,
            "status": "active",
            "description": "Промислова зона з різними підприємствами"
        },
        {
            "name": "Склади логістичного центру",
            "type": "industrial",
            "coords": {"lat": 48.6340, "lng": 35.2520},
            "district_id": 5,
            "status": "active",
            "description": "Логістичний центр з великими складськими приміщеннями"
        },
        {
            "name": "Новий міст",
            "type": "infrastructure",
            "coords": {"lat": 48.6330, "lng": 35.2580},
            "district_id": 5,
            "status": "maintenance",
            "description": "Міст через річку, на даний момент в ремонті"
        },
        {
            "name": "Спортивний майданчик",
            "type": "sport",
            "coords": {"lat": 48.6320, "lng": 35.2510},
```

```
            "district_id": 5,
            "status": "active",
            "description": "Відкритий спортивний майданчик з тренажерами та футбольним
полем"
        }
    ]
    for loc in locations:
        cursor.execute(
            """INSERT INTO locations
            (name, type, coords, district_id, status, description, image_url)
            VALUES (?, ?, ?, ?, ?, ?, ?)""",
            (
                loc['name'],
                loc['type'],
                json.dumps(loc['coords']),
                loc['district_id'],
                loc['status'],
                loc['description'],
                loc.get('image_url', '')
            )
        )
    conn.commit()
    conn.close()
    print(f"Додано {len(locations)} локацій")
# Функція для додавання оцінок
def add_ratings():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Отримання всіх користувачів
    cursor.execute("SELECT id FROM users WHERE role = 'user'")
    users = [row[0] for row in cursor.fetchall()]
    # Отримання всіх локацій
    cursor.execute("SELECT id FROM locations")
    locations = [row[0] for row in cursor.fetchall()]
    ratings_count = 0
    for location_id in locations:
        # Кожну локацію оцінює від 3 до 8 користувачів
        user_sample = random.sample(users, min(random.randint(3, 8), len(users)))
        for user_id in user_sample:
            rating = random.randint(1, 5)
            try:
                cursor.execute(
                    "INSERT INTO ratings (location_id, user_id, rating) VALUES (?, ?, ?)",
                    (location_id, user_id, rating)
                )
                ratings_count += 1
            except sqlite3.IntegrityError:
                # Ігноруємо дублікати (може статися при повторному запуску)
                pass
    conn.commit()
    conn.close()
    print(f"Додано {ratings_count} оцінок")
# Функція для додавання коментарів
def add_comments():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Отримання всіх користувачів
    cursor.execute("SELECT id FROM users WHERE role = 'user'")
    users = [row[0] for row in cursor.fetchall()]
    # Отримання всіх локацій
    cursor.execute("SELECT id FROM locations")
    locations = [row[0] for row in cursor.fetchall()]
    # Приклади коментарів
    comment_templates = [
```

```
        "Дуже гарне місце! Рекомендую відвідати.",
        "Був тут минулого тижня. Все сподобалось.",
        "Не дуже сподобалося. Очікував більшого.",
        "Цікаве місце з багатою історією.",
        "Зручне розташування, легко знайти.",
        "Потребує ремонту та оновлення.",
        "Відвідую регулярно, завжди приємно провести тут час.",
        "Красива архітектура та атмосфера.",
        "Хороше місце для відпочинку з сім'єю.",
        "Не рекомендую відвідувати у вихідні - дуже багато людей.",
        "Чисто та охайно, все добре організовано.",
        "Вартує відвідування! Залишив найкращі враження.",
        "Нічого особливого, звичайне місце.",
        "Дуже сподобалось оформлення та атмосфера.",
        "Приємний персонал та хороший сервіс."
    ]
    comments_count = 0
    for location_id in locations:
        # Для кожної локації створюємо від 5 до 15 коментарів
        num_comments = random.randint(5, 15)
        for _ in range(num_comments):
            user_id = random.choice(users)
            comment_text = random.choice(comment_templates)
            # Дата створення коментаря (останні 3 місяці)
            days_ago = random.randint(0, 90)
            comment_date = (datetime.now() - timedelta(days=days_ago)).strftime("%Y-%m-%d
%H:%M:%S")
            cursor.execute(
                "INSERT INTO comments (location_id, user_id, text, created_at) VALUES (?, ?,
?, ?)",
                (location_id, user_id, comment_text, comment_date)
            )
            comments_count += 1
    conn.commit()
    conn.close()
    print(f"Додано {comments_count} коментарів")
# Функція для додавання повідомлень у чаті
def add_messages():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Отримання всіх користувачів
    cursor.execute("SELECT id FROM users WHERE role = 'user'")
    users = [row[0] for row in cursor.fetchall()]
    # Отримання всіх менеджерів
    cursor.execute("SELECT id FROM users WHERE role = 'manager'")
    managers = [row[0] for row in cursor.fetchall()]
    # Приклади повідомлень від користувачів
    user_message_templates = [
        "Доброго дня! Підкажіть будь ласка, як знайти цю локацію?",
        "Чи можна відвідати це місце з дітьми?",
        "Коли планується завершення реконструкції?",
        "Яка вартість квитків?",
        "Чи є можливість паркування поблизу?",
        "Який графік роботи цієї локації?",
        "Дякую за інформацію!",
        "Чи можна зарезервувати місце заздалегідь?",
        "Як доїхати громадським транспортом?",
        "Чи є поруч заклади харчування?"
    ]
    # Приклади повідомлень від менеджерів
    manager_message_templates = [
        "Доброго дня! Чим можу допомогти?",
        "Звичайно, це місце чудово підходить для відвідування з дітьми.",
        "Реконструкція планується до завершення через 2 місяці.",
```

```
        "Вхід вільний для всіх відвідувачів.",
        "Так, поруч є велика парковка на 50 місць.",
        "Локація працює щоденно з 9:00 до 18:00.",
        "Завжди раді допомогти!",
        "Для резервування зателефонуйте за номером +380XXXXXXXXX.",
        "Ви можете доїхати автобусами №12, №15 або №23.",
        "В радіусі 200 метрів є кілька кафе та ресторанів."
    ]
    messages_count = 0
    # Для кожного користувача створюємо бесіду з кожним менеджером
    for user_id in users:
        for manager_id in managers:
            # Створюємо від 5 до 20 повідомлень у бесіді
            num_messages = random.randint(5, 20)
            # Початкова дата (до 3 місяців тому)
            conversation_start = datetime.now() - timedelta(days=random.randint(30, 90))
            # Формуємо ланцюжок повідомлень
            sender_id = user_id  # Перше повідомлення від користувача
            for i in range(num_messages):
                if sender_id == user_id:
                    message_text = random.choice(user_message_templates)
                    recipient_id = manager_id
                else:
                    message_text = random.choice(manager_message_templates)
                    recipient_id = user_id
                # Дата повідомлення (кожне наступне через випадковий проміжок часу)
                minutes_later = random.randint(5, 60)
                message_date = (conversation_start + timedelta(minutes=minutes_later *
i)).strftime("%Y-%m-%d %H:%M:%S")
                # Статус прочитання (частина старих повідомлень прочитана, нові можуть бути
непрочитані)
                is_read = 1 if random.random() < 0.8 or conversation_start.date() <
(datetime.now() - timedelta(days=7)).date() else 0
                cursor.execute(
                    "INSERT INTO messages (from_user_id, to_user_id, text, created_at,
is_read) VALUES (?, ?, ?, ?, ?)",
                    (sender_id, recipient_id, message_text, message_date, is_read)
                )
                messages_count += 1
                # Міняємо відправника для наступного повідомлення
                sender_id = recipient_id
    conn.commit()
    conn.close()
    print(f"Додано {messages_count} повідомлень у чаті")
# Функція для додавання записів до історії переглядів
def add_view_history():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    # Отримання всіх користувачів
    cursor.execute("SELECT id FROM users WHERE role = 'user'")
    users = [row[0] for row in cursor.fetchall()]
    # Отримання всіх локацій
    cursor.execute("SELECT id FROM locations")
    locations = [row[0] for row in cursor.fetchall()]
    views_count = 0
    for user_id in users:
        # Кожен користувач переглядає від 8 до 15 випадкових локацій
        location_sample = random.sample(locations, min(random.randint(8, 15),
len(locations)))
        for location_id in location_sample:
            # Кожну локацію користувач міг переглядати кілька разів
            num_views = random.randint(1, 5)
            for _ in range(num_views):
                days_ago = random.randint(0, 30)
```

```
                view_date = (datetime.now() - timedelta(days=days_ago,
                                                        hours=random.randint(0, 23),
                                                        minutes=random.randint(0, 59))
                             ).strftime("%Y-%m-%d %H:%M:%S")
                cursor.execute(
                    "INSERT INTO view_history (user_id, location_id, viewed_at) VALUES (?,
?, ?)",
                    (user_id, location_id, view_date)
                )
                views_count += 1
    conn.commit()
    conn.close()
    print(f"Додано {views_count} записів до історії переглядів")
# Виконуємо всі функції з заповнення бази даних
if __name__ == "__main__":
    create_tables()
    add_districts()
    add_users()
    add_locations()
    add_ratings()
    add_comments()
    add_messages()
    add_view_history()
    print("База даних успішно заповнена тестовими даними!")
```

server.py

```
from flask import Flask, jsonify, request, render_template, g, session, redirect, url_for
import sqlite3
import json
import os
import hashlib
import uuid
from datetime import datetime
app = Flask(__name__)
app.secret_key = os.urandom(24)  # для сесії
app.config["DATABASE"] = "locations.db"
# Функції для роботи з базою даних
def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = sqlite3.connect(app.config["DATABASE"])
        db.row_factory = sqlite3.Row
    return db
@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
def create_tables_manually():
    conn = sqlite3.connect(app.config["DATABASE"])
    # Таблиця районів
    conn.execute('''
    CREATE TABLE IF NOT EXISTS districts (
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        boundaries TEXT NOT NULL,
        color TEXT NOT NULL
```

```
)''')
# Таблиця локацій
conn.execute('''
CREATE TABLE IF NOT EXISTS locations (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    type TEXT NOT NULL,
    coords TEXT NOT NULL,
    district_id INTEGER NOT NULL,
    status TEXT NOT NULL,
    description TEXT,
    image_url TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (district_id) REFERENCES districts (id)
)''')
# Таблиця користувачів
conn.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT UNIQUE NOT NULL,
    password TEXT NOT NULL,
    role TEXT NOT NULL,
    registration_date TIMESTAMP NOT NULL
)''')
# Таблиця коментарів
conn.execute('''
CREATE TABLE IF NOT EXISTS comments (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    location_id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    text TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    FOREIGN KEY (location_id) REFERENCES locations (id),
    FOREIGN KEY (user_id) REFERENCES users (id)
)''')
# Таблиця оцінок
conn.execute('''
CREATE TABLE IF NOT EXISTS ratings (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    location_id INTEGER NOT NULL,
    user_id INTEGER NOT NULL,
    rating INTEGER NOT NULL CHECK (rating BETWEEN 1 AND 5),
    FOREIGN KEY (location_id) REFERENCES locations (id),
    FOREIGN KEY (user_id) REFERENCES users (id),
    UNIQUE (location_id, user_id)
)''')
# Таблиця повідомлень для чату
conn.execute('''
CREATE TABLE IF NOT EXISTS messages (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    from_user_id INTEGER NOT NULL,
    to_user_id INTEGER NOT NULL,
    text TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    is_read BOOLEAN NOT NULL DEFAULT 0,
    FOREIGN KEY (from_user_id) REFERENCES users (id),
    FOREIGN KEY (to_user_id) REFERENCES users (id)
)''')
# Таблиця історії переглядів
conn.execute('''
CREATE TABLE IF NOT EXISTS view_history (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    location_id INTEGER NOT NULL,
```

```
        viewed_at TIMESTAMP NOT NULL,
        FOREIGN KEY (user_id) REFERENCES users (id),
        FOREIGN KEY (location_id) REFERENCES locations (id)
    )''')
    conn.commit()
    conn.close()
def init_db():
    if not os.path.exists(app.config["DATABASE"]):
        # Створення таблиць напряму з коду
        create_tables_manually()
        with app.app_context():
            db = get_db()
            # Визначення районів безпосередньо в коді
            districts = [
                {
                    "id": 1,
                    "name": "Центральний",
                    "boundaries": [
                        [48.6389, 35.2107],
                        [48.6389, 35.2360],
                        [48.6268, 35.2360],
                        [48.6268, 35.2107]
                    ],
                    "color": "#FF5733"
                },
                {
                    "id": 2,
                    "name": "Північний",
                    "boundaries": [
                        [48.6500, 35.2107],
                        [48.6500, 35.2360],
                        [48.6389, 35.2360],
                        [48.6389, 35.2107]
                    ],
                    "color": "#33FF57"
                },
                {
                    "id": 3,
                    "name": "Південний",
                    "boundaries": [
                        [48.6268, 35.2107],
                        [48.6268, 35.2360],
                        [48.6150, 35.2360],
                        [48.6150, 35.2107]
                    ],
                    "color": "#3357FF"
                },
                {
                    "id": 4,
                    "name": "Західний",
                    "boundaries": [
                        [48.6389, 35.1850],
                        [48.6389, 35.2107],
                        [48.6268, 35.2107],
                        [48.6268, 35.1850]
                    ],
                    "color": "#F3FF33"
                },
                {
                    "id": 5,
                    "name": "Східний",
                    "boundaries": [
                        [48.6389, 35.2360],
                        [48.6389, 35.2600],
```

```
                        [48.6268, 35.2600],
                        [48.6268, 35.2360]
                    ],
                    "color": "#FF33F3"
                }
            ]
            # Додавання районів до бази даних
            try:
                for district in districts:
                    db.execute(
                        "INSERT INTO districts (id, name, boundaries, color) VALUES (?, ?,
?, ?)",
                        (district['id'], district['name'],
json.dumps(district['boundaries']), district['color'])
                    )
                db.commit()
                print("Райони успішно додано до бази даних")
            except Exception as e:
                print(f"Помилка при додаванні районів: {e}")
            # Додавання базових користувачів
            default_manager_password = hashlib.sha256("manager123".encode()).hexdigest()
            db.execute(
                "INSERT INTO users (username, password, role, registration_date) VALUES (?,
?, ?, ?)",
                ("manager", default_manager_password, "manager",
datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
            )
            # Додавання початкових локацій
            locations = [
                {
                    "name": "Троїцький собор",
                    "type": "landmark",
                    "coords": {"lat": 48.6311, "lng": 35.2206},
                    "district_id": 1,
                    "status": "active",
                    "description": "Історична дерев'яна церква, пам'ятка архітектури
національного значення",
                    "image_url": ""
                },
                {
                    "name": "Парк імені Сучкова",
                    "type": "park",
                    "coords": {"lat": 48.6335, "lng": 35.2290},
                    "district_id": 1,
                    "status": "active",
                    "description": "Центральний міський парк відпочинку",
                    "image_url": ""
                },
                {
                    "name": "Центральна площа",
                    "type": "square",
                    "coords": {"lat": 48.6325, "lng": 35.2250},
                    "district_id": 1,
                    "status": "active",
                    "description": "Головна площа міста з фонтаном та міською
адміністрацією",
                    "image_url": ""
                },
                {
                    "name": "Автовокзал",
                    "type": "transport",
                    "coords": {"lat": 48.6350, "lng": 35.2180},
                    "district_id": 2,
                    "status": "active",
```

```
                "description": "Головний автовокзал міста",
                "image_url": ""
            },
            {
                "name": "Залізничний вокзал",
                "type": "transport",
                "coords": {"lat": 48.6200, "lng": 35.2300},
                "district_id": 3,
                "status": "active",
                "description": "Залізнична станція Новомосковськ",
                "image_url": ""
            }
        ]
        for loc in locations:
            db.execute(
                """INSERT INTO locations
                (name, type, coords, district_id, status, description, image_url)
                VALUES (?, ?, ?, ?, ?, ?, ?)""",
                (loc['name'], loc['type'], json.dumps(loc['coords']),
                loc['district_id'], loc['status'], loc['description'], loc['image_url'])
            )
        db.commit()
        print("Початкові дані успішно завантажено")
# Маршрути для веб-інтерфейсу
@app.route('/')
def index():
    return render_template('index.html')
# API для отримання районів
@app.route('/api/districts', methods=['GET'])
def get_districts():
    districts = query_db('SELECT * FROM districts')
    print(f"Запитано районів. Знайдено: {len(districts)}")
    result = []
    for d in districts:
        district_dict = dict(d)
        district_dict['boundaries'] = json.loads(district_dict['boundaries'])
        result.append(district_dict)
    print(f"Відправлено районів: {len(result)}")
    return jsonify(result)
# API для отримання локацій
@app.route('/api/locations', methods=['GET'])
def get_locations():
    district_id = request.args.get('district_id')
    location_type = request.args.get('type')
    status = request.args.get('status')
    search = request.args.get('search')
    query = 'SELECT * FROM locations WHERE 1=1'
    params = []
    if district_id:
        query += ' AND district_id = ?'
        params.append(district_id)
    if location_type:
        query += ' AND type = ?'
        params.append(location_type)
    if status:
        query += ' AND status = ?'
        params.append(status)
    if search:
        query += ' AND (name LIKE ? OR description LIKE ?)'
        search_term = f'%{search}%'
        params.extend([search_term, search_term])
    locations = query_db(query, params)
    result = []
    for loc in locations:
```

```
        loc_dict = dict(loc)
        loc_dict['coords'] = json.loads(loc_dict['coords'])
        result.append(loc_dict)
    return jsonify(result)
# API для отримання деталей конкретної локації
@app.route('/api/locations/<int:location_id>', methods=['GET'])
def get_location(location_id):
    location = query_db('SELECT * FROM locations WHERE id = ?', [location_id], one=True)
    if not location:
        return jsonify({'error': 'Локацію не знайдено'}), 404
    loc_dict = dict(location)
    loc_dict['coords'] = json.loads(loc_dict['coords'])
    # Отримання коментарів для локації
    comments = query_db('''
        SELECT c.*, u.username
        FROM comments c
        JOIN users u ON c.user_id = u.id
        WHERE c.location_id = ?
        ORDER BY c.created_at DESC
    ''', [location_id])
    comments_list = [dict(c) for c in comments]
    loc_dict['comments'] = comments_list
    # Отримання середньої оцінки
    rating = query_db('SELECT AVG(rating) as avg_rating FROM ratings WHERE location_id = ?',
[location_id], one=True)
    loc_dict['average_rating'] = rating['avg_rating'] if rating['avg_rating'] else 0
    return jsonify(loc_dict)
# Авторизація та реєстрація
@app.route('/api/auth/login', methods=['POST'])
def login():
    if not request.json or not 'username' in request.json or not 'password' in request.json:
        return jsonify({'error': 'Потрібно вказати логін та пароль'}), 400
    username = request.json['username']
    password = hashlib.sha256(request.json['password'].encode()).hexdigest()
    user = query_db('SELECT * FROM users WHERE username = ? AND password = ?', [username,
password], one=True)
    if user:
        user_dict = dict(user)
        session['user_id'] = user_dict['id']
        session['username'] = user_dict['username']
        session['role'] = user_dict['role']
        # Видаляємо пароль перед відправкою
        del user_dict['password']
        return jsonify({'success': True, 'user': user_dict})
    else:
        return jsonify({'error': 'Неправильний логін або пароль'}), 401
@app.route('/api/auth/register', methods=['POST'])
def register():
    if not request.json or not 'username' in request.json or not 'password' in request.json:
        return jsonify({'error': 'Потрібно вказати логін та пароль'}), 400
    username = request.json['username']
    password = hashlib.sha256(request.json['password'].encode()).hexdigest()
    # Перевірка, чи існує користувач
    user_exists = query_db('SELECT * FROM users WHERE username = ?', [username], one=True)
    if user_exists:
        return jsonify({'error': 'Користувач з таким ім\'ям вже існує'}), 400
    # Додавання нового користувача
    db = get_db()
    try:
        db.execute(
            "INSERT INTO users (username, password, role, registration_date) VALUES (?, ?,
?, ?)",
            (username, password, "user", datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
        )
```

```
        db.commit()
        # Отримання створеного користувача
        new_user = query_db('SELECT * FROM users WHERE username = ?', [username], one=True)
        user_dict = dict(new_user)
        # Встановлення сесії
        session['user_id'] = user_dict['id']
        session['username'] = user_dict['username']
        session['role'] = user_dict['role']
        # Видаляємо пароль перед відправкою
        del user_dict['password']
        return jsonify({'success': True, 'user': user_dict})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка реєстрації: {str(e)}'}), 500
@app.route('/api/auth/logout', methods=['POST'])
def logout():
    session.pop('user_id', None)
    session.pop('username', None)
    session.pop('role', None)
    return jsonify({'success': True})
@app.route('/api/auth/user', methods=['GET'])
def get_current_user():
    if 'user_id' in session:
        user = query_db('SELECT id, username, role, registration_date FROM users WHERE id = 
?', [session['user_id']], one=True)
        if user:
            return jsonify({'user': dict(user)})
    return jsonify({'user': None})
# API для взаємодії з локаціями (CRUD для менеджерів)
@app.route('/api/locations', methods=['POST'])
def add_location():
    if 'role' not in session or session['role'] != 'manager':
        return jsonify({'error': 'Немає доступу'}), 403
    if not request.json:
        return jsonify({'error': 'Неправильний запит'}), 400
    required_fields = ['name', 'type', 'coords', 'district_id', 'status', 'description']
    for field in required_fields:
        if field not in request.json:
            return jsonify({'error': f'Поле {field} обов\'язкове'}), 400
    location = request.json
    db = get_db()
    try:
        db.execute(
            """INSERT INTO locations
            (name, type, coords, district_id, status, description, image_url)
            VALUES (?, ?, ?, ?, ?, ?, ?)""",
            (
                location['name'],
                location['type'],
                json.dumps(location['coords']),
                location['district_id'],
                location['status'],
                location['description'],
                location.get('image_url', '')
            )
        )
        db.commit()
        new_location_id = db.cursor().lastrowid
        return jsonify({'success': True, 'id': new_location_id})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка додавання локації: {str(e)}'}), 500
@app.route('/api/locations/<int:location_id>', methods=['PUT'])
def update_location(location_id):
```

```
    if 'role' not in session or session['role'] != 'manager':
        return jsonify({'error': 'Немає доступу'}), 403
    if not request.json:
        return jsonify({'error': 'Неправильний запит'}), 400
    location = query_db('SELECT * FROM locations WHERE id = ?', [location_id], one=True)
    if not location:
        return jsonify({'error': 'Локацію не знайдено'}), 404
    update_data = request.json
    update_fields = []
    update_values = []
    for field in ['name', 'type', 'district_id', 'status', 'description', 'image_url']:
        if field in update_data:
            update_fields.append(f"{field} = ?")
            update_values.append(update_data[field])
    if 'coords' in update_data:
        update_fields.append("coords = ?")
        update_values.append(json.dumps(update_data['coords']))
    if not update_fields:
        return jsonify({'error': 'Немає даних для оновлення'}), 400
    update_values.append(location_id)
    db = get_db()
    try:
        db.execute(
            f"UPDATE locations SET {', '.join(update_fields)} WHERE id = ?",
            update_values
        )
        db.commit()
        return jsonify({'success': True})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка оновлення локації: {str(e)}'}), 500
@app.route('/api/locations/<int:location_id>', methods=['DELETE'])
def delete_location(location_id):
    if 'role' not in session or session['role'] != 'manager':
        return jsonify({'error': 'Немає доступу'}), 403
    location = query_db('SELECT * FROM locations WHERE id = ?', [location_id], one=True)
    if not location:
        return jsonify({'error': 'Локацію не знайдено'}), 404
    db = get_db()
    try:
        # Видалення пов'язаних коментарів та оцінок
        db.execute('DELETE FROM comments WHERE location_id = ?', [location_id])
        db.execute('DELETE FROM ratings WHERE location_id = ?', [location_id])
        # Видалення локації
        db.execute('DELETE FROM locations WHERE id = ?', [location_id])
        db.commit()
        return jsonify({'success': True})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка видалення локації: {str(e)}'}), 500
# API для коментарів
@app.route('/api/locations/<int:location_id>/comments', methods=['GET'])
def get_comments(location_id):
    comments = query_db('''
        SELECT c.*, u.username
        FROM comments c
        JOIN users u ON c.user_id = u.id
        WHERE c.location_id = ?
        ORDER BY c.created_at DESC
    ''', [location_id])
    return jsonify([dict(c) for c in comments])
@app.route('/api/locations/<int:location_id>/comments', methods=['POST'])
def add_comment(location_id):
    if 'user_id' not in session:
```

```
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    if not request.json or 'text' not in request.json:
        return jsonify({'error': 'Текст коментаря обов\'язковий'}), 400
    location = query_db('SELECT * FROM locations WHERE id = ?', [location_id], one=True)
    if not location:
        return jsonify({'error': 'Локацію не знайдено'}), 404
    db = get_db()
    try:
        now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        db.execute(
            "INSERT INTO comments (location_id, user_id, text, created_at) VALUES (?, ?, ?,
?)",
            (location_id, session['user_id'], request.json['text'], now)
        )
        db.commit()
        # Отримання доданого коментаря
        new_comment = query_db('''
            SELECT c.*, u.username
            FROM comments c
            JOIN users u ON c.user_id = u.id
            WHERE c.location_id = ? AND c.user_id = ?
            ORDER BY c.created_at DESC LIMIT 1
        ''', [location_id, session['user_id']], one=True)
        return jsonify({'success': True, 'comment': dict(new_comment)})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка додавання коментаря: {str(e)}'}), 500
@app.route('/api/comments/<int:comment_id>', methods=['DELETE'])
def delete_comment(comment_id):
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    comment = query_db('SELECT * FROM comments WHERE id = ?', [comment_id], one=True)
    if not comment:
        return jsonify({'error': 'Коментар не знайдено'}), 404
    # Перевірка прав: користувач може видаляти свої коментарі, менеджер - всі
    if dict(comment)['user_id'] != session['user_id'] and session.get('role') != 'manager':
        return jsonify({'error': 'Немає доступу для видалення цього коментаря'}), 403
    db = get_db()
    try:
        db.execute('DELETE FROM comments WHERE id = ?', [comment_id])
        db.commit()
        return jsonify({'success': True})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка видалення коментаря: {str(e)}'}), 500
# API для оцінок
@app.route('/api/locations/<int:location_id>/ratings', methods=['POST'])
def rate_location(location_id):
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    if not request.json or 'rating' not in request.json:
        return jsonify({'error': 'Оцінка обов\'язкова'}), 400
    rating = request.json['rating']
    if not isinstance(rating, int) or rating < 1 or rating > 5:
        return jsonify({'error': 'Оцінка повинна бути числом від 1 до 5'}), 400
    location = query_db('SELECT * FROM locations WHERE id = ?', [location_id], one=True)
    if not location:
        return jsonify({'error': 'Локацію не знайдено'}), 404
    # Перевірка, чи користувач вже оцінював цю локацію
    existing_rating = query_db(
        'SELECT * FROM ratings WHERE location_id = ? AND user_id = ?',
        [location_id, session['user_id']],
        one=True
    )
```

```
    db = get_db()
    try:
        if existing_rating:
            # Оновлення існуючої оцінки
            db.execute(
                'UPDATE ratings SET rating = ? WHERE location_id = ? AND user_id = ?',
                [rating, location_id, session['user_id']]
            )
        else:
            # Додавання нової оцінки
            db.execute(
                'INSERT INTO ratings (location_id, user_id, rating) VALUES (?, ?, ?)',
                [location_id, session['user_id'], rating]
            )
        db.commit()
        # Отримання оновленої середньої оцінки
        avg_rating = query_db(
            'SELECT AVG(rating) as avg_rating FROM ratings WHERE location_id = ?',
            [location_id],
            one=True
        )
        return jsonify({
            'success': True,
            'average_rating': avg_rating['avg_rating'] if avg_rating['avg_rating'] else 0
        })
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка оцінювання локації: {str(e)}'}), 500
# API для чату
@app.route('/api/messages/<int:recipient_id>', methods=['GET'])
def get_messages(recipient_id):
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    # Отримання повідомлень між користувачами
    messages = query_db('''
        SELECT m.*, u_from.username as from_username, u_to.username as to_username
        FROM messages m
        JOIN users u_from ON m.from_user_id = u_from.id
        JOIN users u_to ON m.to_user_id = u_to.id
        WHERE (m.from_user_id = ? AND m.to_user_id = ?) OR (m.from_user_id = ? AND
m.to_user_id = ?)
        ORDER BY m.created_at ASC
    ''', [session['user_id'], recipient_id, recipient_id, session['user_id']])
    # Позначення повідомлень як прочитаних
    db = get_db()
    db.execute(
        'UPDATE messages SET is_read = 1 WHERE to_user_id = ? AND from_user_id = ? AND
is_read = 0',
        [session['user_id'], recipient_id]
    )
    db.commit()
    return jsonify([dict(m) for m in messages])
@app.route('/api/messages/<int:recipient_id>', methods=['POST'])
def send_message(recipient_id):
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    if not request.json or 'text' not in request.json:
        return jsonify({'error': 'Текст повідомлення обов\'язковий'}), 400
    # Перевірка на існування одержувача
    recipient = query_db('SELECT * FROM users WHERE id = ?', [recipient_id], one=True)
    if not recipient:
        return jsonify({'error': 'Одержувача не знайдено'}), 404
    # Перевірка прав для чату між користувачем та менеджером
    sender_role = session.get('role')
```

```
    recipient_role = dict(recipient)['role']
    if (sender_role == 'user' and recipient_role != 'manager') or (sender_role == 'manager'
and recipient_role != 'user'):
        return jsonify({'error': 'Чат можливий лише між користувачем та менеджером'}), 403
    db = get_db()
    try:
        now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        db.execute(
            "INSERT INTO messages (from_user_id, to_user_id, text, created_at, is_read)
VALUES (?, ?, ?, ?, ?)",
            (session['user_id'], recipient_id, request.json['text'], now, 0)
        )
        db.commit()
        # Отримання надісланого повідомлення
        new_message = query_db('''
            SELECT m.*, u_from.username as from_username, u_to.username as to_username
            FROM messages m
            JOIN users u_from ON m.from_user_id = u_from.id
            JOIN users u_to ON m.to_user_id = u_to.id
            WHERE m.from_user_id = ? AND m.to_user_id = ?
            ORDER BY m.created_at DESC LIMIT 1
        ''', [session['user_id'], recipient_id], one=True)
        return jsonify({'success': True, 'message': dict(new_message)})
    except Exception as e:
        db.rollback()
        return jsonify({'error': f'Помилка надсилання повідомлення: {str(e)}'}), 500
@app.route('/api/messages/unread', methods=['GET'])
def get_unread_messages():
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    # Підрахунок непрочитаних повідомлень від кожного користувача
    unread_counts = query_db('''
        SELECT from_user_id, COUNT(*) as count, u.username
        FROM messages m
        JOIN users u ON m.from_user_id = u.id
        WHERE to_user_id = ? AND is_read = 0
        GROUP BY from_user_id
    ''', [session['user_id']])
    return jsonify([dict(u) for u in unread_counts])
@app.route('/api/users/managers', methods=['GET'])
def get_managers():
    if 'user_id' not in session:
        return jsonify({'error': 'Необхідно авторизуватися'}), 401
    managers = query_db('SELECT id, username FROM users WHERE role = "manager"')
    return jsonify([dict(m) for m in managers])
@app.route('/api/users', methods=['GET'])
def get_users():
    if 'role' not in session or session['role'] != 'manager':
        return jsonify({'error': 'Немає доступу'}), 403
    users = query_db('SELECT id, username, role FROM users WHERE role = "user"')
    return jsonify([dict(u) for u in users])
# Запуск сервера
if __name__ == '__main__':
    # Перевірка існування БД і створення, якщо потрібно
    init_db()
    app.run(debug=True)
```