МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ ПОШУКУ ТА ОРЕНДИ ЖИТЛА ЗАСОБАМИ NODE.JS ТА POSTGRESQL»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
Д.С. Мельник
(ініціали та прізвище)
Керівник доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Мельнику Дмитрові Сергійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка серверної частини системи пошуку та оренди житла засобами Node.js та PostgreSQL

керівник роботи Панасенко Євген Валерійович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 04.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичні основи та аналіз технологій.
2. Проєктування та розробка системи.
3. Програмування та реалізація системи.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	02.04.2025	
2.	Збір вихідних даних.	10.04.2025	
3.	Обробка методичних та теоретичних	09.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	20.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	18.12.2025	

Студент _____ Д.С. Мельник
(підпис) (ініціали та прізвище)

Керівник роботи _____ Є.В. Панасенко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка серверної частини системи пошуку та оренди житла засобами Node.js та PostgreSQL»: 88 сторінок, 19 рисунків, 12 джерел.

БАЗА ДАНИХ, ВЕБСЕРВІС, МІКРОСЕРВІСНА АРХІТЕКТУРА, ОРЕНДА ЖИТЛА, ПРОГРАМНА ІНЖЕНЕРІЯ, СИСТЕМА ПОШУКУ, API, GEO-ПОШУК, NODE.JS, POSTGRESQL, REST API.

Об’єкт дослідження – процес проєктування, розробки та впровадження серверної частини вебсистеми для пошуку та оренди житла, що включає архітектурні рішення, моделі даних, алгоритми геопросторового пошуку та механізми обробки транзакцій між орендарями та орендодавцями.

Мета роботи: розробка високопродуктивної, масштабованої та безпечної серверної частини системи пошуку та оренди житла з використанням сучасних вебтехнологій Node.js та PostgreSQL, що забезпечує ефективну взаємодію користувачів, обробку геоданих, пошук за складними критеріями та управління всіма етапами процесу оренди нерухомості.

Методи дослідження – методи аналізу вимог до програмного забезпечення, методи проєктування мікросервісної архітектури, методи реляційного та геопросторового моделювання даних, методи розробки RESTful API, методи інтеграції зовнішніх сервісів, методи оптимізації запитів до бази даних, а також методи тестування продуктивності та безпеки.

У результаті отримано функціональну серверну частину системи пошуку та оренди житла, яка забезпечує повний цикл обробки транзакцій оренди, включаючи пошук нерухомості за географічним розташуванням, управління оголошеннями, бронювання, оплату та комунікацію між користувачами. Система демонструє високу продуктивність при обробці складних геопросторових запитів та масштабується для роботи з великими обсягами даних. Результати роботи можуть бути використані як основа для комерційних платформ оренди житла.

SUMMARY

Master's qualifying paper "Development of the Backend for a Housing Search and Rental System Using Node.js and PostgreSQL": 88 pages, 19 figures, 12 references.

API, DATABASE, GEO-SEARCH, HOUSING RENTAL, MICROSERVICE ARCHITECTURE, NODE.JS, POSTGRESQL, REST API, SEARCH SYSTEM, SOFTWARE ENGINEERING, WEB SERVICE.

Object of the study is the process of designing, developing, and implementing the server part of a web system for housing search and rental, which includes architectural solutions, data models, geospatial search algorithms, and transaction processing mechanisms between tenants and landlords.

The aim of the study is development of a high-performance, scalable, and secure server part of a housing search and rental system using modern web technologies Node.js and PostgreSQL, ensuring effective user interaction, processing of geodata, search by complex criteria, and management of all stages of the real estate rental process.

Research methods include methods of analyzing software requirements, methods of designing microservice architecture, methods of relational and geospatial data modeling, methods of developing RESTful API, methods of integrating external services, methods of optimizing database queries, as well as methods of testing performance and security.

As a result of the qualification work, a functional server part of the housing search and rental system has been obtained, which provides a full cycle of rental transaction processing, including real estate search by geographic location, advertisement management, booking, payment, and communication between users. The system demonstrates high performance in processing complex geospatial queries and scales for working with large volumes of data. The results of the work can be used as a basis for commercial housing rental platforms, real estate aggregators, or corporate systems for managing rental property portfolios.

ЗМІСТ

Завдання на кваліфікаційну роботу ……..2
Реферат 4
Summary 5
Вступ..... 7
1 Теоретичні основи та аналіз технологій 9
1.1 Визначення, типи та особливості сучасного вебконтенту..... 9
1.2 Огляд великих мовних моделей (LLM) як інструменту генерації текстів..... 15
1.3 Аналіз існуючих сервісів та API для генерації контенту: можливості та обмеження 17
1.4 Архітектурні підходи до інтеграції LLM у вебзастосунки..... 20
1.5 Специфіка обробки геоданих у контексті пошуку нерухомості..... 22
1.6 Вимоги до безпеки та продуктивності систем оренди житла 25
1.7 Висновки до розділу 1 27
2 Проєктування та розробка системи 29
2.1 Формування вимог до функціоналу та архітектури застосунку 29
2.2 Вибір та обґрунтування технологічного стеку 32
2.3 Проєктування схеми бази даних PostgreSQL 40
2.4 Проєктування серверної частини 44
2.5 Висновки до розділу 2 51
3 Програмування та реалізація системи..... 53
3.1 Налаштування середовища розробки та інфраструктури..... 53
3.2 Розробка сервісу аутентифікації та авторизації..... 61
3.3 Реалізація модуля управління нерухомістю та геопошуку 67
3.4 Тестування та оптимізація продуктивності системи..... 72
3.5 Висновки до розділу 3 83
Висновки 85
Перелік посилань..... 87

ВСТУП

Сучасний розвиток технологій штучного інтелекту та обробки природної мови відкриває нові горизонти для автоматизації створення вебконтенту. Однією з ключових задач є перетворення коротких текстових запитів у якісний, релевантний контент для вебсайтів та соціальних мереж. Прикладом таких рішень є платформи на кшталт Jasper.ai, Copy.ai або ChatGPT, які дозволяють генерувати статті, соціальні пости та рекламні тексти шляхом використання великих мовних моделей.

Метою кваліфікаційної роботи є проєктування та реалізація вебзастосунку для генерації вебконтенту з використанням Python та FastAPI для реалізації серверної частини та ML-мікросервісів, React та TypeScript для розробки клієнтського інтерфейсу, WordPress REST API для інтеграції з CMS-системами, а також PostgreSQL та Redis як систем зберігання даних.

Для досягнення поставленої мети сформульовано такі задачі:
проаналізувати сучасні підходи та інструменти генерації контенту за допомогою великих мовних моделей;

- сформулювати функціональні та нефункціональні вимоги до вебзастосунку;
- спроєктувати архітектуру системи з чітким розмежуванням функціоналу;
- реалізувати основний API та бізнес-логіку на Python з використанням фреймворку FastAPI;
- розробити інтерфейс користувача з використанням React та TypeScript;
- створити модуль інтеграції з WordPress REST API для автоматичної публікації контенту;
- реалізувати систему кешування результатів генерації з використанням Redis;

– організувати зберігання та управління даними в PostgreSQL;
– виконати тестування окремих компонентів та провести інтеграційне тестування системи.

Структурно робота складається з трьох розділів. У першому розділі проаналізовано сучасні системи генерації контенту, їхні функціональні можливості та архітектурні рішення, а також обґрунтовано вибір технологічного стеку: Python та FastAPI для бекенду та ML-сервісів, React та TypeScript для інтуїтивного інтерфейсу, WordPress REST API для безперервної публікації контенту, а також PostgreSQL та Redis для надійного зберігання даних та підвищення продуктивності.

У другому розділі виконано аналіз функціональних і нефункціональних вимог до системи, спроєктовано архітектуру з використанням діаграм UML, розроблено схеми взаємодії між компонентами системи, а також спроєктовано структуру бази даних PostgreSQL та моделі кешування в Redis.

У третьому розділі реалізовано основні компоненти системи: серверну частину на Python з використанням FastAPI, клієнтський інтерфейс на React з TypeScript, модуль інтеграції з WordPress, а також інтегровано систему зберігання даних. Проведено комплексне тестування окремих модулів та системи в цілому, включаючи тестування продуктивності та якості згенерованого контенту.

1 ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ТЕХНОЛОГІЙ

1.1 Визначення, типи та особливості сучасного вебконтенту

Розробка систем пошуку та оренди житла вимагає застосування технологічних рішень, що забезпечують високу продуктивність, масштабованість, надійність та зручність користування. Сучасні платформи повинні обробляти великі обсяги даних про нерухомість, підтримувати тисячі одночасних запитів, забезпечувати швидкий геопросторовий пошук та надавати інтуїтивний інтерфейс для користувачів [1].

Основними технологічними тенденціями у цій галузі є використання мікросервісної архітектури, API-First підходу, впровадження геолокаційних сервісів реального часу, оптимізація для мобільних пристроїв та інтеграція штучного інтелекту для персоналізації пропозицій.

Для серверної розробки систем оренди житла широко використовуються JavaScript/Node.js фреймворки, що дозволяють створювати високопродуктивні та масштабовані API. Node.js з Express.js часто обирається через асинхронну модель введення-виведення, що дозволяє ефективно обробляти одночасні запити, а також через єдину мову програмування на всіх рівнях додатку. Ця технологія використовується у таких платформах, як Uber та LinkedIn, що демонструє її придатність для високонавантажених систем [1].

Альтернативні серверні рішення включають Python з FastAPI або Django для швидкої розробки прототипів, Go з Gin або Echo для максимальної продуктивності, та Java з Spring Boot для корпоративних рішень. Вибір технології залежить від розміру команди, очікуваного навантаження та специфічних вимог до бізнес-логіки [1].

PostgreSQL є основним рішенням для зберігання даних систем оренди житла через свою надійність, розширені можливості та підтримку геопросторових запитів за допомогою розширення PostGIS.

Ця база даних підтримує:

– складні геопросторові індекси для прискорення пошуку за місцезнаходженням;
– типи даних GeoJSON для роботи з координатами;
– транзакційну цілісність для операцій бронювання;
– повнотекстовий пошук для описів нерухомості;
– JSONB для зберігання динамічних атрибутів.

Альтернативні рішення включають MySQL для менших проєктів, MongoDB для гнучкого зберігання документів та Elasticsearch для розширеного пошуку. Часто використовується комбінація PostgreSQL з Redis для кешування частотних запитів та підвищення продуктивності [1].

Для забезпечення швидкого геопошуку використовуються спеціалізовані технології:

– PostGIS для геопросторових запитів у PostgreSQL;
– Google Maps API або Mapbox для візуалізації;
– Redis GEO для кешування геоданих;
– Elasticsearch з геопросторовими можливостями для складних пошукових сценаріїв.

Сучасні системи також інтегрують механізми реального часу через WebSocket для:

– миттєвих сповіщень про нові пропозиції;
– Online-чату між орендодавцями та орендарями;
– оновлення статусу бронювань у реальному часі;
– синхронізації календаря доступності.

Тенденцією останніх років є поєднання традиційних технологій з AI/ML компонентами для:

– прогнозування цін на оренду;
– персоналізованих рекомендацій;
– автоматичної модерації оголошень;
– розпізнавання обману та шахрайства.

Розвиток мобільних технологій призвів до поширення PWA (Progressive Web Apps), що поєднують переваги веб- та нативних додатків, забезпечуючи офлайн-роботу та push-сповіщення навіть без встановлення додатку [1].

По-перше, наведемо переваги Node.js:

– асинхронна, подієво-орієнтована архітектура (Node.js заснований на неблокуючій моделі введення-виведення, що дозволяє ефективно обробляти тисячі одночасних підключень на одному потоці, забезпечуючи високу продуктивність для реального часу та високонавантажених систем оренди житла);
– JavaScript як універсальна мова розробки (Node.js дозволяє використовувати JavaScript як на сервері, так і на клієнті, спрощуючи розробку, підтримку та співпрацю між frontend та backend командами, зменшуючи контекстне перемикання між мовами);
– потужна екосистема NPM (Node Package Manager) (має найбільший репозиторій відкритих пакетів серед усіх мов програмування, що надає готові рішення для практично будь-якої задачі – від автентифікації до геопошуку, значно прискорюючи розробку);
– швидкість виконання завдяки рушію V8 (використовує той самий високошвидкісний JavaScript-рушій, що й Chrome, що забезпечує відмінну продуктивність та ефективне використання ресурсів сервера);
– легкість масштабування (архітектура Node.js ідеально підходить для горизонтального масштабування, створення мікросервісів та контейнеризації за допомогою Docker, що критично важливо для систем оренди житла зі змінним навантаженням);
– ідеальна сумісність з сучасними базами даних (особливо з NoSQL базами даних та JSON-орієнтованими рішеннями, а також чудова підтримка для роботи з PostgreSQL через такі бібліотеки як node-postgres, що спрощує реалізацію геопошуку через PostGIS);
– вбудована підтримка вебсокетів для real-time функціоналу (Node.js ідеально підходить для реалізації чатів між орендодавцями та

орендарями, миттєвих сповіщень про нові оголошення та оновлення статусу бронювань у реальному часі);

- активна спільнота та довгострокова підтримка (підтримується компанією OpenJS Foundation та великою корпоративною спільнотою, що гарантує стабільність, безпеку та регулярні оновлення);
- оптимальна архітектура для RESTful API та GraphQL (Node.js з фреймворками на кшталт Express.js дозволяє швидко створювати легкі, продуктивні API для взаємодії з вебклієнтами та мобільними додатками);
- зручні інструменти розробки та налагодження (широкий вибір IDE, дебагерів, профайлерів та інструментів моніторингу, таких як PM2 для керування процесами, що спрощує розробку та підтримку продакшн-середовищ).

Незважаючи на переваги Node.js, ця платформа містить обмеження та недоліки, які потрібно враховувати при розробці серверної частини системи пошуку та оренди житла:

- проблеми з обробкою CPU-інтенсивних задач (архітектура Node.js оптимізована для I/O-операцій, але може демонструвати низьку продуктивність при виконанні важких обчислень, таких як складні геопросторові аналізи або обробка великих масивів даних без використання воркерів);
- необхідність ручного управління пам’яттю (Node.js не має вбудованого ефективного збору сміття для довгоживучих об’єктів, що може призводити до витоку пам’яті у великих додатках, особливо при роботі з кешем та довгоживучими підключеннями до бази даних);
- складність обробки помилок у асинхронному коді (потік виконання в асинхронному середовищі може ускладнювати відстеження та обробку помилок, особливо при використанні вкладених зворотних

викликів (callback hell), хоча сучасні async/await покращують ситуацію);

- залежність від якості сторонніх пакетів (велика екосистема NPM містить як високоякісні, так і нестабільні бібліотеки, що потребує ретельного вибору та моніторингу залежностей для забезпечення стабільності продакшн-середовища);
- обмежена багатопоточність (хоча Node.js підтримує Worker Threads, модель багатопоточності є більш складною у використанні порівняно з іншими мовами, що може обмежувати використання всіх ядер процесора для паралельної обробки запитів).

Проаналізувавши загальні характеристики Node.js та врахувавши особливості розроблюваної системи пошуку та оренди житла, можна виділити ряд ключових переваг даної платформи, які роблять її оптимальним вибором саме для розробки серверної частини [1].

Специфіка системи пошуку житла, враховуючи її високі вимоги до продуктивності, необхідність обробки великої кількості одночасних запитів та інтеграцію з геолокаційними сервісами, знаходить відображення в архітектурних та технологічних особливостях Node.js:

- асинхронна обробка запитів (система оренди житла потребує ефективної обробки тисячі одночасних запитів на пошук, фільтрацію та бронювання; неблокуюча модель Node.js ідеально підходить для такого типу навантаження);
- ефективна робота з базами даних (Node.js відмінно інтегрується з PostgreSQL через бібліотеку node-postgres, забезпечуючи швидкий доступ до даних, асинхронні запити та підтримку транзакцій, що критично важливо для операцій бронювання);
- геопросторові можливості (можливість безпосередньої роботи з GeoJSON та інтеграція з PostGIS дозволяє ефективно реалізувати геопошук нерухомості з підтримкою радіусного пошуку, фільтрації за районами та оптимізації маршрутів);

– Real-time функціональність (вбудована підтримка WebSocket дозволяє реалізувати чат між орендодавцями та орендарями, миттєві сповіщення про нові пропозиції та оновлення статусу бронювань у реальному часі);
– масштабованість архітектури (модульна структура Node.js дозволяє легко розділяти систему на мікросервіси для окремої масштабованості компонентів пошуку, бронювання, автентифікації та нотифікацій).

При виборі технології для розробки серверної частини системи пошуку та оренди житла важливо провести порівняльний аналіз з іншими популярними серверними технологіями, які потенційно підходили б для даної задачі [1].

На 2025 рік виділяють п’ять наступних популярних серверних технологій для розробки вебсистем:

– Node.js з Express/Nest.js;
– Python з FastAPI/Django;
– Java з Spring Boot;
– Go з Gin/Echo;
– C# з ASP.NET Core.

Виконаємо коротке порівняння з двома найпопулярнішими альтернативами Node.js з наведеного списку. Python з FastAPI пропонує швидку розробку прототипів, вбудовану документацію OpenAPI та відмінну підтримку машинного навчання, але поступається Node.js у продуктивності при обробці великої кількості одночасних I/O-запитів, що для системи пошуку житла означає потенційні проблеми з масштабуванням при зростанні навантаження.

Хоча Python має чудові бібліотеки для аналізу даних, його синхронна природа може обмежувати продуктивність у сценаріях з великою кількістю паралельних підключень до бази даних та зовнішніх API.

Java з Spring Boot пропонує потужну корпоративну архітектуру, відмінну багатопоточність та довгострокову підтримку, але поступається

Node.js у легкості розробки, швидкості запуску та споживанні пам’яті, що для системи оренди житла може означати більш високі витрати на інфраструктуру та складніше масштабування. Хоча Spring Boot надає вичерпний набір інструментів для побудови складних систем, його крива навчання є значно крутішою, а процес розгортання більш складним порівняно з Node.js [2].

Go як альтернатива демонструє видатну продуктивність та ефективне використання ресурсів, але має менш розвинену екосистему для веброзробки та більш обмежені можливості для швидкої розробки складних бізнес-логік порівняно з Node.js, що може уповільнити розробку системи пошуку житла з її складною доменною логікою та інтеграціями [2].

Таким чином, вибір Node.js для розробки серверної частини системи пошуку та оренди житла є обґрунтованим рішенням, що базується на його асинхронній архітектурі, високій продуктивності при I/O-навантаженні, єдиній мові програмування з frontend частиною, великій екосистемі NPM для швидкої інтеграції геопошуку, автентифікації та кешування, а також простоті масштабування та контейнеризації**.** Ці характеристики забезпечують необхідну продуктивність, масштабованість та гнучкість для реалізації функціональних вимог системи пошуку та оренди житла, де критично важливими є обробка тисячі одночасних запитів, швидкий геопошук та стабільна робота в режимі реального часу [2].

1.2 Огляд великих мовних моделей (LLM) як інструменту генерації текстів

Для розробки серверної частини системи пошуку та оренди житла необхідно обґрунтовано обрати технологічний стек, що забезпечить продуктивність, надійність, масштабованість та безпеку системи. Розглянемо ключові технології, які будуть використані у даному проєкті [2].

Node.js – це середовище виконання JavaScript побудоване на рушії Chrome V8, яке дозволяє виконувати JavaScript-код на сервері. Ключовою

перевагою Node.js є асинхронна, подієво-орієнтована архітектура, яка дозволяє обробляти велику кількість одночасних підключень з мінімальними витратами ресурсів. Це робить Node.js ідеальним вибором для систем реального часу, API з високим навантаженням та додатків, що потребують інтенсивної роботи з I/O-операціями [2].

Основні характеристики Node.js, важливі для системи оренди житла:

– асинхронне програмування через async/await та проміси дозволяє ефективно обробляти запити до бази даних та зовнішніх сервісів без блокування потоку виконання;
– єдина мова програмування на сервері та клієнті спрощує розробку та підтримку;
– модульна система через npm (Node Package Manager) з найбільшою екосистемою відкритих бібліотек;
– вбудована підтримка WebSocket для реалізації функціоналу реального часу;
– легка контейнеризація за допомогою Docker та оркестрація через Kubernetes.

Express.js – мінімалістичний та гнучкий вебфреймворк для Node.js, який надає набір потужних інструментів для створення вебдодатків та API. Express дозволяє швидко створювати серверні додатки з підтримкою маршрутизації, middleware-функцій, шаблонізації та обробки HTTP-запитів [2].

Ключові можливості Express.js для системи оренди житла:

– маршрутизація з підтримкою параметрів, запитів та проміжної обробки;
– Middleware-архітектура для реалізації автентифікації, логування, обробки помилок та CORS;
– інтеграція з шаблонізаторами для генерації динамічного HTML (якщо потрібно);
– підтримка RESTful API з чіткою структурою ендпоінтів;
– легка інтеграція з базами даних, кеш-системами та зовнішніми API.

PostgreSQL – потужна, відкрита об’єктно-реляційна система управління базами даних (СУБД) з розширеними можливостями. PostgreSQL особливо добре підходить для складних, транзакційних додатків, що вимагають надійності, цілісності даних та підтримки складних запитів [2].

Критично важливі можливості PostgreSQL для системи пошуку житла включають, по-перше, постійність та надійність через повну ACID-сумісність для гарантованої цілісності даних, транзакційність для операцій бронювання та реплікацію для високої доступності системи. Ці характеристики є фундаментальними для забезпечення безпеки фінансових транзакцій, уникнення конфліктів при одночасних бронюваннях та мінімізації часу простою платформи [2].

По-друге, геопросторові можливості через розширення PostGIS забезпечують підтримку географічних та геометричних типів даних, просторові індекси для прискорення геопошуку, функції для розрахунку відстаней, площ та перетинів, а також підтримку стандарту GeoJSON. Ця функціональність дозволяє реалізувати ефективний пошук нерухомості в заданому радіусі, фільтрацію за районами міста та точне відображення локацій на інтерактивних картах [2].

По-третє, розширені типи даних та механізми продуктивності, включаючи JSON/JSONB для гнучкого зберігання динамічних атрибутів, діапазони (range types) для календаря доступності, матеріалізовані подання для швидких агрегацій та розділення таблиць (partitioning) для масштабування, забезпечують високу ефективність системи при роботі з великими обсягами даних та складними бізнес-процесами оренди житла [2].

1.3 Аналіз існуючих сервісів та API для генерації контенту: можливості та обмеження

Сучасний ринок платформ з пошуку та оренди житла представлений різноманітними сервісами, кожен з яких пропонує унікальний набір функцій,

архітектурних рішень та технологічних підходів. Аналіз цих рішень дозволяє виявити кращі практики, потенційні проблеми та оптимальні архітектурні шаблони для розробки власної системи [3].

Глобальні платформи (Airbnb, Booking.com, Vrbo) демонструють високий рівень технологічної складності та масштабованості. Airbnb, зокрема, використовує мікросервісну архітектуру з сотнями незалежних сервісів, що комунікують через REST API та подійні шини. Ключовою особливістю є розвинена система геопошуку на базі Elasticsearch з підтримкою складних фільтрів за типом житла, зручностями, відстані до визначних місць. Booking.com відрізняється потужною системою динамічного ціноутворення та алгоритмами рекомендацій на основі машинного навчання. Обмеженнями цих платформ є висока складність архітектури, що ускладнює розгортання та підтримку для невеликих команд, та жорстка прив’язка до власної екосистеми сервісів [3].

Локальні та регіональні платформи (OLX Нерухомість, Dom.Ria, LUN) пропонують більш прості, але ефективні рішення, орієнтовані на специфіку локальних ринків. Ці сервіси часто використовують монолітну або гібридну архітектуру з елементами мікросервісів для критичних компонентів. Особливістю є інтеграція з місцевими платіжними системами, підтримка специфічних типів нерухомості та локалізований підхід до верифікації оголошень. Технологічний стек часто включає PHP (Laravel/Symfony), Python (Django) або Node.js для основних сервісів, PostgreSQL або MySQL для даних, Redis для кешування. Обмеженнями є менша масштабованість порівняно з глобальними платформами та частіше обмежений набір API для інтеграції зі сторонніми сервісами [3].

API та сервіси третьої сторони становлять важливу частину екосистеми пошуку житла. Google Maps Platform та Mapbox надають потужні інструменти для геокодування, відображення мап, розрахунку маршрутів та геопошуку. Stripe, PayPal та локальні платіжні системи (LiqPay, Fondy) забезпечують інфраструктуру для безпечних транзакцій. SendGrid, Mailgun та Twilio

використовуються для комунікації з користувачами. Обмеженнями цих сервісів є залежність від зовнішніх провайдерів, вартість використання при високих навантаженнях та потенційні проблеми з доступністю [3].

Відкриті API та стандарти також відіграють важливу роль. OpenStreetMap пропонує безкоштовні геодані, але з обмеженою точністю порівняно з комерційними рішеннями. Schema.org надає стандартизовану розмітку для нерухомості, що покращує SEO та інтеграцію з пошуковими системами. Обмеженням відкритих рішень часто є нестача документації, обмежена підтримка та непередбачувані зміни в API [3].

Архітектурні підходи та технологічні рішення в існуючих системах варіюються від традиційних монолітів до сучасних серверліс-архітектур. Монолітна архітектура (часто зустрічається у стартапів та локальних платформ) пропонує простоту розробки та розгортання, але обмежує масштабованість та гнучкість. Мікросервісна архітектура (Airbnb, Booking.com) забезпечує високу масштабованість та відмовостійкість, але вимагає значних зусиль для оркестрації, моніторингу та підтримки. Серверліс-підхід (AWS Lambda, Google Cloud Functions) набирає популярності для специфічних задач, таких як обробка зображень, відправка сповіщень або обчислення рекомендацій [3].

Технологічні стекли існуючих рішень показують різноманітність підходів. Для серверної частини популярними є: Java (Spring Boot) у корпоративних рішеннях, Node.js (Express, NestJS) у стартапів та середніх проєктів, Python (Django, FastAPI) для швидкої розробки прототипів. Бази даних включають: PostgreSQL (часто з PostGIS) для основних даних, MongoDB для документо-орієнтованих даних, Redis для кешування та сесій, Elasticsearch для пошуку. Інфраструктура зазвичай базується на хмарних платформах (AWS, Google Cloud, Azure) з використанням контейнеризації (Docker) та оркестрації (Kubernetes) [3].

Цей аналіз дозволяє сформулювати оптимальні архітектурні рішення для розроблюваної системи, уникнути поширених помилок та обрати

технології, які забезпечать конкурентні переваги на ринку послуг оренди житла [3].

1.4 Архітектурні підходи до інтеграції LLM у вебзастосунки

Розробка серверної частини системи пошуку та оренди житла вимагає ретельного вибору архітектурного підходу, який забезпечить продуктивність, масштабованість, надійність та легкість підтримки системи. Розглянемо основні архітектурні підходи та їх застосування в контексті розробки платформи оренди житла [4].

Монолітна архітектура є традиційним підходом, де всі компоненти системи (обробка запитів, бізнес-логіка, доступ до даних) об’єднані в єдину кодобуду та розгортаються як один процес. Для системи пошуку житла цей підхід може бути ефективним на початкових етапах розробки, коли команда невелика, а функціональність обмежена. Переваги включають простоту розгортання, легшу відладку та меншу складність розробки. Однак моноліт обмежує масштабованість – при зростанні навантаження доводиться масштабувати весь додаток, а не окремі компоненти. Також ускладнюється інтеграція нових технологій та оновлення окремих частин системи [4].

Мікросервісна архітектура передбачає розподіл системи на набір незалежних сервісів, кожен з яких відповідає за конкретну бізнес-функцію. Для системи пошуку житла це може включати окремі сервіси для:

- пошуку та фільтрації – обробка запитів на пошук, геолокація, рангінг результатів;
- управління оголошеннями – CRUD-операції для нерухомості, валідація даних;
- бронювання та транзакцій – резервування, оплата, управління календарем доступності;
- автентифікації та користувачів – реєстрація, логін, управління профілями;

– сповіщень та комунікації – email/SMS повідомлення, чат між користувачами.

Переваги мікросервісної архітектури включають незалежне масштабування окремих компонентів, можливість використання різних технологій для різних сервісів, покращену відмовостійкість та полегшену підтримку великими командами. Недоліки – збільшена складність оркестрації, моніторингу, налагодження та забезпечення консистентності даних між сервісами [4].

Подієво-орієнтована архітектура (Event-Driven Architecture) базується на обміні подіями між компонентами системи через message broker (RabbitMQ, Apache Kafka, Redis Pub/Sub). У контексті системи оренди житла цей підхід дозволяє ефективно обробляти асинхронні операції, такі як:

– обробка нових оголошень – після створення оголошення генерується подія для індексації в пошуковій системі, перевірки модератором та розсилки сповіщень підписникам;
– бронювання – подія про успішне бронювання запускає процеси сповіщення власника, оновлення календаря, створення інвойсу;
– оновлення даних – зміни в профілі користувача або характеристиках житла автоматично поширюються на всі залежні сервіси.

Ця архітектура покращує масштабованість, зменшує зв’язність компонентів та підвищує відмовостійкість системи [4].

Архітектура на основі API Gateway використовує єдину точку входу для всіх клієнтських запитів, яка перенаправляє їх до відповідних мікросервісів. API Gateway для системи оренди житла може виконувати такі функції:

– маршрутизація запитів до відповідних сервісів;
– агрегація даних з кількох сервісів для складних відповідей;
– автентифікація та авторизація всіх вхідних запитів;
– кешування частотних запитів;
– лімітування запитів (rate limiting) для захисту від DDoS-атак;
– трансформація протоколів (REST/gRPC/WebSocket).

Архітектурний вибір для конкретної системи пошуку житла залежить від багатьох факторів:

– очікуване навантаження – кількість користувачів, запитів на пошук, операцій бронювання;
– команда розробників – досвід, розмір, методологія роботи;
– терміни розробки – необхідність швидкого виходу на ринок (time-to-market);
– бюджет – витрати на інфраструктуру, ліцензії, підтримку;
– плани масштабування – географічне розширення, додавання нових функцій.

Гібридний підхід часто виявляється оптимальним, де критичні компоненти (пошук, бронювання) реалізуються як мікросервіси, а допоміжні функції (статичні сторінки, адмін-панель) – у моноліті. Стратегія розгортання може включати контейнеризацію (Docker) та оркестрацію (Kubernetes) для забезпечення гнучкості та масштабованості [4].

Вибір архітектурного підходу безпосередньо впливає на технологічний стек, процеси розробки, тестування, розгортання та подальшої підтримки системи пошуку та оренди житла [4].

1.5 Специфіка обробки геоданих у контексті пошуку нерухомості

Обробка географічних даних є критично важливим аспектом розробки системи пошуку та оренди житла, що вимагає спеціалізованих технологічних рішень та архітектурних підходів. Геопросторові дані в контексті нерухомості включають координати об’єктів, межі районів, інфраструктурні об’єкти та транспортні мережі, що створює унікальні виклики для їх ефективного зберігання, індексування та пошуку [4].

Основні типи геопросторових даних у системі пошуку житла включають точкові об’єкти (точні координати будинків та квартир), полігональні об’єкти

(межу районів міст, парків, зелених зон) та лінійні об’єкти (транспортні магістралі, лінії метро). Кожен тип даних вимагає специфічних методів зберігання та індексування в PostgreSQL з використанням розширення PostGIS, яке підтримує географічні типи даних GEOGRAPHY для роботи з координатами на сфероїді Землі та GEOMETRY для роботи з координатами в декартовій системі [4].

Оптимізація геопросторових запитів вимагає ретельного вибору просторових індексів. Для точкових даних найчастіше використовуються GIST (Generalized Search Tree) індекси, які ефективно працюють з операторами відстані та включення. Для складних полігональних об’єктів може застосовуватися SP-GIST індекс для просторового розділення. Критично важливим є вибір між географічним та геометричним типом даних: географічний тип точніше працює з відстанями на реальній земній поверхні, але має обмежену підтримку деяких операцій та вищу обчислювальну складність [5].

Архітектурні підходи до геопошуку включають різні стратегії обробки запитів. При реалізації радіусного пошуку необхідно враховувати кривизну земної поверхні через функцію ST_DistanceSphere, яка обчислює відстань по великому колу, на відміну від ST_Distance, що працює з евклідовою відстанню. Для пошуку в межах полігону (районів, адміністративних одиниць) використовується оператор ST_Within, що вимагає попередньої оптимізації полігонів та використання просторових індексів.

Продуктивність геопросторових запитів залежить від декількох ключових факторів. При роботі з великими наборами даних ефективним рішенням є просторовий кластеризація (spatial clustering) за допомогою функції ST_ClusterDBSCAN, що групує близько розташовані об’єкти для зменшення кількості перевірок. Матеріалізовані подання (materialized views) можуть кешувати результати складних геопросторових агрегацій, таких як статистика по районах або середні ціни в географічних кластерах.

Кешування геоданих реалізується через Redis з використанням геопросторових команд. GEOADD дозволяє додавати об’єкти з координатами

до геопросторового індексу Redis, а GEORADIUS здійснює пошук об’єктів у заданому радіусі. Це особливо ефективно для кешування результатів пошуку по популярним локаціям, оскільки Redis зберігає дані в оперативній пам’яті та забезпечує субмілісекундну затримку при пошуку.

Інтеграція з зовнішніми геосервісами є критично важливою для розширення функціоналу системи. Геокодування адрес через Google Maps Geocoding API або OpenStreetMap Nominatim перетворює текстові адреси у географічні координати. Зворотне геокодування дозволяє отримати адресу за координатами. Розрахунок маршрутів та відстаней здійснюється через Google Directions API або OSRM (Open Source Routing Machine) для обчислення часу подорожі до ключових об’єктів інфраструктури.

Обробка реального часу геолокації вимагає спеціалізованих підходів. При відстеженні позиції користувачів для персоналізованих пропозицій використовуються геофенси (geofences) – віртуальні периметри, які запускають події при вході або виході користувача. Для цього застосовуються функції PostGIS ST_Intersects та ST_Contains з оптимізованими просторовими індексами [5].

Масштабування геопросторової інфраструктури передбачає розподілене зберігання геоданих. Шардінг за географічними регіонами дозволяє розподілити дані по кількох серверах PostgreSQL на основі географічного розташування об’єктів. Географічно розподілені кеші реалізуються через Redis Cluster з підтримкою геопросторових операцій, що дозволяє зберігати дані ближче до кінцевих користувачів для зменшення затримок [5].

Валідація та очищення геоданих є важливим етапом обробки. Автоматична перевірка координат на відповідність регіону, виявлення викидів (outliers) за допомогою статистичних методів та нормалізація адрес забезпечують якість даних у системі. Для цього використовуються алгоритми кластеризації та просторової статистики, реалізовані в PostGIS через функції просторової агрегації [5].

Візуалізація геопросторових даних на клієнтській стороні реалізується через інтеграцію з бібліотеками Leaflet.js або Mapbox GL JS. Серверна частина

генерує векторні плитки (vector tiles) у форматі MVT (Mapbox Vector Tiles) через PostGIS функцію ST_AsMVT, що дозволяє ефективно передавати великі обсяги геоданих для відображення на інтерактивних картах [5].

1.6 Вимоги до безпеки та продуктивності систем оренди житла

Розробка сучасної системи пошуку та оренди житла передбачає забезпечення високої продуктивності, надійності та масштабованості на рівні архітектурних рішень та інфраструктури. Технічні вимоги до системи формуються з урахуванням специфіки високонавантажених вебдодатків, що працюють з геопросторовими даними та реалізують складну бізнес-логіку в режимі реального часу [6].

Архітектурні вимоги визначають основоположні принципи побудови системи. Система повинна базуватися на мікросервісній архітектурі з чітким розділенням відповідальності між компонентами. Кожен мікросервіс повинен мати автономну базу даних та комунікувати через легкі транспортні протоколи (REST, gRPC або події через message broker). Подієво-орієнтована архітектура має застосовуватися для асинхронної обробки операцій, таких як нотифікації, індексація пошуку та оновлення кешу. API Gateway повинен служити єдиною точкою входу для всіх клієнтських запитів, виконуючи функції маршрутизації, агрегації даних, автентифікації та кешування [6].

Вимоги до продуктивності включають строгі обмеження на час відповіді системи. Для API запитів 95-й процентиль часу відповіді не повинен перевищувати 200 мс при середньому навантаженні. Геопросторовий пошук з використанням PostGIS має виконуватися за ≤100 мс навіть при складних фільтрах та великих радіусах пошуку. Операції читання з кешу (Redis) повинні мати латенсі ≤5 мс, а операції запису – ≤10 мс. Система повинна підтримувати ≥10 000 одночасних підключень без деградації продуктивності, з можливістю обробки ≥1000 запитів за секунду (RPS) на кожному з критичних ендпоінтів [6].

Вимоги до масштабованості визначають здатність системи до горизонтального масштабування. Архітектура повинна дозволяти незалежне масштабування окремих мікросервісів залежно від навантаження. Автоматичне масштабування (auto-scaling) має бути реалізоване на основі метрик CPU, пам’яті та кількості запитів. Система повинна підтримувати географічне розподілення (geo-distribution) для зменшення затримок для користувачів з різних регіонів. Балансування навантаження має здійснюватися на рівні L4 (TCP/UDP) та L7 (HTTP/HTTPS) з підтримкою health checks та circuit breakers [6].

Інфраструктурні вимоги стосуються технічного середовища розгортання:

- контейнеризація: всі компоненти повинні бути пакувані в Docker-контейнери;
- оркестрація: використання Kubernetes для автоматичного розгортання, масштабування та керування контейнерами;
- хмарна інфраструктура: розгортання на AWS, Google Cloud Platform або Azure з використанням managed-сервісів (RDS для PostgreSQL, ElastiCache для Redis);
- CI/CD пайплайни: автоматизоване тестування, збірка та розгортання з використанням GitLab CI/CD, Jenkins або GitHub Actions;
- моніторинг та логування: інтеграція з Prometheus для збору метрик, Grafana для візуалізації, ELK Stack (Elasticsearch, Logstash, Kibana) для централізованого логування.

Вимоги до обробки геопросторових даних визначають спеціалізовані можливості системи. PostgreSQL з розширенням PostGIS має підтримувати географічні типи даних (GEOGRAPHY, GEOMETRY) та протокол GeoJSON для обміну даними з клієнтськими додатками. Просторові індекси (GIST, SP-GIST) повинні застосовуватися для прискорення геопошуку, з підтримкою запитів на відстань (ST_Distance), пошуку в межах полігону (ST_Within) та розрахунку площ перетину. Кешування геоданих має реалізовуватися через Redis з використанням геопросторових команд (GEOADD, GEORADIUS) [6].

Вимоги до надійності та відмовостійкості включають гарантії доступності системи. Сервіс-левел угоди (SLA) повинні забезпечувати доступність на рівні ≥99.95% для критичних компонентів. Реплікація баз даних має реалізовуватися в конфігурації master-slave або master-master з автоматичним failover. Резервне копіювання даних повинно виконуватися щоденно зі зберіганням backup протягом 30 днів. Disaster recovery план повинен забезпечувати відновлення системи протягом ≤4 годин після катастрофічного збою.

Вимоги до безпеки визначають захищеність системи на різних рівнях. Шифрування даних має застосовуватися як при передачі (TLS 1.3), так і при зберіганні (AES-256). Контроль доступу повинен реалізовуватися через JWT токени з коротким терміном дії та refresh токенами. Валідація вхідних даних має здійснюватися на всіх рівнях системи з захистом від OWASP Top 10 вразливостей. Аудит дій користувачів та системи повинен вестися зі зберіганням логів протягом не менше 6 місяців [6].

Вимоги до інтеграції визначають здатність системи взаємодіяти зі сторонніми сервісами. API-first підхід повинен забезпечувати чітку документацію через OpenAPI (Swagger) специфікацію. Webhook-підтримка необхідна для сповіщення зовнішніх систем про ключові події (нові бронювання, зміни статусів). Стандартизація формату даних через JSON Schema дозволить забезпечити сумісність при інтеграції з іншими платформами. Ці технічні вимоги формують фундамент для проєктування архітектури системи пошуку та оренди житла, забезпечуючи її конкурентоздатність, надійність та здатність до еволюції в умовах динамічного ринку цифрових послуг [6].

1.7 Висновки до розділу 1

У першому розділі проведено комплексний аналіз теоретичних основ та сучасних технологічних рішень для розробки серверної частини системи

пошуку та оренди житла. Виявлено, що ключовими вимогами до таких систем є висока продуктивність геопросторових запитів, масштабованість архітектури, надійність транзакційних операцій бронювання та інтеграція з сучасними геосервісами.

Аналіз ринку показав перевагу мікросервісних архітектур та подієво-орієнтованих підходів для обробки асинхронних операцій. Технологічний стек на базі Node.js, Express.js та PostgreSQL з розширенням PostGIS визначено як оптимальний для реалізації системи, забезпечуючи ефективну обробку геоданих, високу продуктивність при великій кількості одночасних запитів та гнучкість масштабування.

Специфіка обробки геопросторових даних вимагає спеціалізованих методів індексування, кешування та оптимізації запитів для забезпечення швидкого пошуку нерухомості в заданому радіусі та за складними географічними критеріями.

Отримані результати формують теоретичну основу для проєктування архітектури та вибору технологій у наступних розділах роботи.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ

2.1 Формування вимог до функціоналу та архітектури застосунку

Перед початком розробки системи пошуку та оренди житла, виконавши аналіз переваг та недоліків конкурентних рішень на ринку (таких як Airbnb, Booking.com, OLX-нерухомість та інших аналогів розроблюваного продукту), слід провести детальний аналіз вимог до системи. Це дозволить сформувати чітке розуміння майбутнього продукту, визначити його функціональні межі та можливості, а також встановити критерії успішності проєкту. Такий систематичний підхід забезпечує відповідність кінцевого програмного рішення реальним потребам користувачів (орендарів, орендодавців та адміністраторів), мінімізує ризики невдачі проєкту та створює надійний фундамент для подальших етапів проєктування, розробки та впровадження [7].

Функціональні вимоги визначають конкретні функції та можливості, які повинна надавати фінальна система, деталізуючи її поведінку в різних сценаріях використання. Короткий перелік основних функціональних вимог для системи пошуку та оренди житла виглядає наступним чином:

- реєстрація, аутентифікація та авторизація користувачів з різними ролями (орендар, орендодавець, адміністратор) та можливістю верифікації облікових записів;
- створення, редагування, публікація та архівація детальних оголошень про нерухомість із завантаженням фотографій, вказівкою цін, описів, зручностей та правил;
- розширений геопросторовий пошук нерухомості за локацією (на карті), типом, ціновим діапазоном, кількістю кімнат, доступними зручностями та датами доступності;
- інтеграція з платіжними системами для безпечної онлайн-оплати оренди, внесення завдатків та управління поверненнями коштів;

- система бронювання та календар доступності з можливістю вибору дат заїзду/виїзду, автоматичним розрахунком вартості та попереднім підтвердженням/блокуванням дат;
- вбудована система комунікації у формі чату/обміну повідомленнями між орендарем та орендодавцем щодо конкретного бронювання;
- система рейтингів та відгуків, де обидві сторони можуть залишати оцінки та коментарі після завершення оренди;
- панель управління для орендодавця з оглядом всіх оголошень, бронювань, фінансовою статистикою та календарем зайнятості;
- особистий кабінет користувача з історією пошуків, улюбленими оголошеннями, активними бронюваннями та повідомленнями;
- панель адміністратора для модерації оголошень, управління користувачами, моніторингу транзакцій та перегляду системної аналітики.

Система пошуку та оренди житла повинна забезпечувати повний життєвий цикл взаємодії з нерухомістю: від пошуку та вибору житла до бронювання, оплати, проживання та подальшого оцінювання. Особливу увагу слід приділити інтуїтивному та потужному інтерфейсу пошуку, який дозволяє швидко та точно відфільтрувати тисячі оголошень за десятками критеріїв, включаючи візуальний пошук на карті. Важливим аспектом є наявність прозорої та безпечної системи оплат, що захищає фінансові інтереси обох сторін. Механізми перевірки та репутації (відгуки, верифікація) мають формувати довіру між незнайомими користувачами платформи [7].

Нефункціональні вимоги визначають якісні характеристики системи, які не пов’язані безпосередньо з її конкретними функціями, але є критичними для забезпечення належної продуктивності, надійності, безпеки та користувацького досвіду. Категоріально нефункціональні вимоги до системи наступні:

- продуктивність – система повинна забезпечувати швидкий час відгуку при одночасній роботі тисяч користувачів, особливо для операцій пошуку та фільтрації;

- надійність – доступність системи (uptime) повинна становити не менше 99.5% часу, забезпечуючи безперебійний доступ до бронювань та календарів;
- масштабованість – архітектура повинна дозволяти горизонтальне масштабування для обробки зростаючої кількості оголошень, користувачів та транзакцій;
- безпека – захист конфіденційних даних (особиста інформація, платіжні реквізити) через шифрування, захист від SQL-ін’єкцій, XSS-атак та регулярні пентести;
- зручність використання (UX) – інтуїтивно зрозумілий, послідовний інтерфейс, який мінімізує кількість кроків для ключових операцій (пошук, бронювання);
- адаптивність – коректне та зручне відображення інтерфейсу на різних пристроях (десктопи, планшети, мобільні телефони);
- обробка геоданих – висока швидкість виконання геопросторових запитів (пошук у радіусі, поблизу точок інтересу) за допомогою спеціалізованих індексів (PostGIS);
- відновлюваність – надійна система резервного копіювання даних з можливістю швидкого відновлення після апаратних чи програмних збоїв.

Забезпечення високої продуктивності та надійності системи є ключовими факторами її конкурентоспроможності та комерційного успіху, особливо в умовах високого навантаження та необхідності обробки складних геопросторових запитів. Система повинна забезпечувати час відгуку не більше 200 мілісекунд для 95% операцій пошуку навіть при піковому навантаженні, а також гарантувати цілісність і узгодженість даних при транзакціях бронювання та оплати. Архітектура системи має будуватися на принципах мікросервісів, що дозволяє незалежно масштабувати найбільш навантажені компоненти (пошук, бронювання) [7].

Особлива увага має бути приділена питанням безпеки та відповідності законодавству (наприклад, GDPR), включаючи безпечне зберігання паролів,

захист від шахрайських операцій, ретельну модерацію контенту та прозору політику конфіденційності. Інтерфейс користувача повинен будуватися на принципах user-centered design, будучи однаково ефективним як для технічно підготовлених агентств нерухомості, так і для звичайних користувачів, які вперше шукають житло [7].

Отже, виконавши комплексний аналіз функціональних та нефункціональних вимог, було отримано детальну та структуровану основу для подальшого проєктування архітектури, вибору технологічного стеку, моделювання бази даних та безпосередньої розробки серверної частини системи пошуку та оренди житла [7].

2.2 Вибір та обґрунтування технологічного стеку

Вибір технологій для розробки серверної частини системи пошуку та оренди житла є стратегічним рішенням, що визначає майбутню продуктивність, масштабованість та ефективність підтримки проєкту. На основі аналізу функціональних вимог, зокрема необхідності обробки геопросторових даних, забезпечення високої доступності та підтримки складних транзакцій, було сформовано оптимальний технологічний стек [8].

Node.js з TypeScript обрано як основну платформу розробки через її асинхронну, подієво-орієнтовану архітектуру, що ідеально підходить для високонавантажених систем з великою кількістю одночасних підключень. Важливими перевагами є висока швидкість обробки I/O-операцій, єдина мова програмування для всіх шарів системи, а також найбільша в світі екосистема пакетів npm. Використання TypeScript забезпечує статичну типізацію, що значно підвищує надійність коду, спрощує рефакторинг та покращує підтримку в сучасних IDE [8].

Фреймворк Express.js обраний для побудови REST API через свою мінімалістичність, гнучкість та доведену стабільність у великомасштабних проєктах. Його архітектура middleware дозволяє легко інтегрувати додаткову

функціональність, таку як аутентифікація, валідація даних, логування та обробка помилок. Express.js має велику спільноту розробників, численні готові рішення та добре документований API, що прискорює процес розробки та спрощує підтримку системи [8].

На рисунку 2.1 зображено діаграму послідовності для процесу бронювання житла, яка візуалізує повний цикл взаємодії між компонентами системи при створенні нового бронювання. Діаграма демонструє поетапну послідовність дій, починаючи від ініціювання запиту користувачем до фінального підтвердження бронювання та відправки сповіщень.

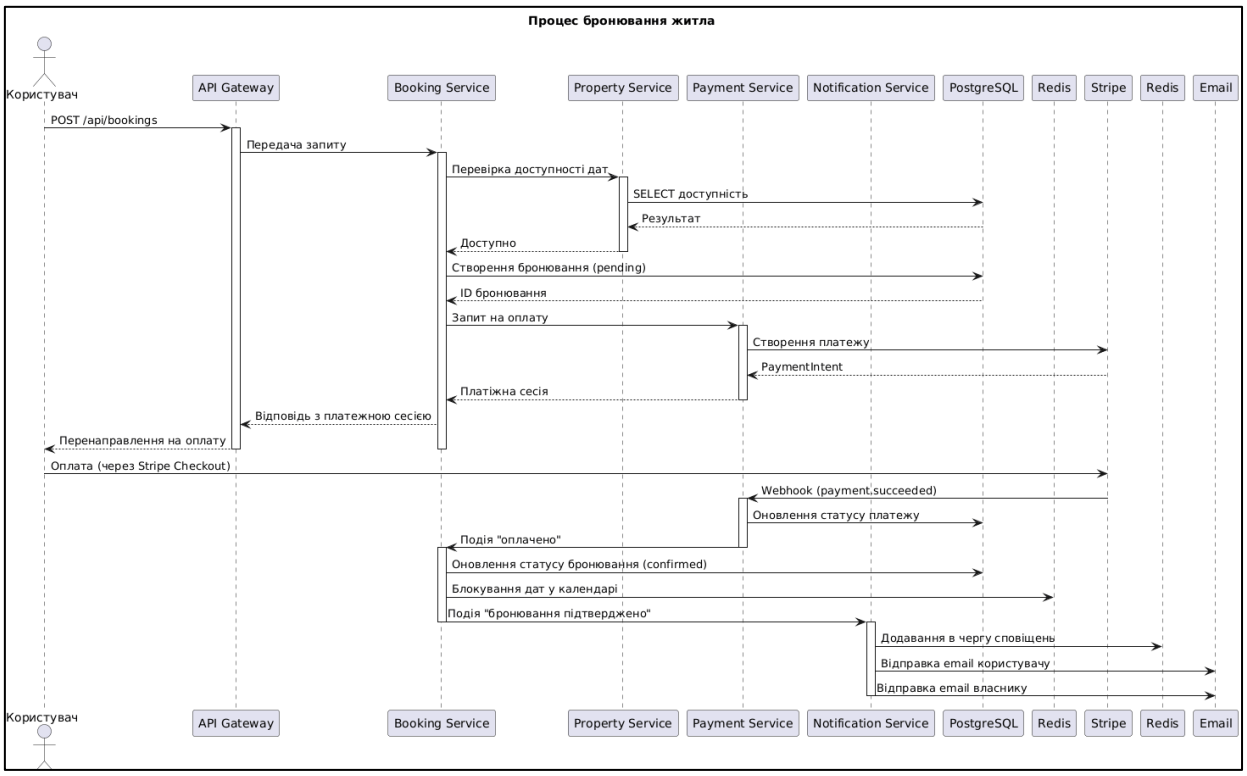


Рисунок 2.1 – Діаграму послідовності для процесу бронювання житла

Перший етап процесу починається з ініціювання запиту користувачем. Користувач через вебінтерфейс або мобільний додаток відправляє POST-запит на створення бронювання, вказуючи бажані дати заїзду та виїзду, ідентифікатор нерухомості та інші необхідні параметри. Цей запит спочатку потрапляє до API Gateway, який виконує функції маршрутизації, балансування навантаження та первинної валідації. Після перевірки автентифікаційного

токена та базової валідації даних, запит перенаправляється до Booking Service – спеціалізованого мікросервісу, відповідального за всю логіку бронювань [8].

Другий етап включає перевірку доступності обраних дат. Отримавши запит, Booking Service звертається до Property Service для підтвердження, що обрана нерухомість вільна у вказаний період. Property Service виконує запит до бази даних PostgreSQL, де зберігається календар доступності кожної нерухомості. Запит використовує оптимізовані індекси та можливості PostGIS для швидкої перевірки конфліктуючих бронювань. Якщо дати доступні, Property Service повертає позитивну відповідь, після чого Booking Service створює новий запис про бронювання в базі даних зі статусом “pending” (очікує оплати) [8].

Третій етап – організація платіжної транзакції. Після успішного створення запису про бронювання, Booking Service ініціює процес оплати через Payment Service. Цей сервіс спеціалізується на взаємодії із зовнішніми платіжними системами. Payment Service створює платіжну сесію в системі Stripe, передаючи необхідні параметри: суму, валюту, опис послуги та callback URL. Stripe генерує унікальний ідентифікатор сесії та URL для сторінки оплати, який повертається через ланцюжок сервісів назад до користувача. Користувач автоматично перенаправляється на захищену сторінку Stripe Checkout для введення платіжних даних [8].

Четвертий етап – асинхронна обробка підтвердження оплати. Після успішного завершення платежу на стороні Stripe, платіжна система надсилає асинхронний вебхук (подія payment.succeeded) на заздалегідь налаштований ендпоінт Payment Service. Така асинхронна модель взаємодії дозволяє системі коректно обробляти оплати навіть у разі тимчасових мережевих проблем або високого навантаження. Отримавши вебхук, Payment Service перевіряє його автентичність через цифровий підпис, оновлює статус транзакції в базі даних PostgreSQL та генерує внутрішню подію «оплата підтверджена» [8].

П’ятий етап – фіналізація бронювання та блокування дат. Booking Service, отримавши подію про успішну оплату, змінює статус бронювання з

“pending” на “confirmed”. Одночасно він виконує критично важливу операцію – блокування обраних дат в календарі доступності. Ця операція відбувається через Redis, що забезпечує атомарність операції та миттєву синхронізацію між усіма екземплярами сервісів. Використання Redis для цієї мети дозволяє уникнути race condition у випадку одночасних бронювань однієї нерухомості [8].

Шостий та завершальний етап – система сповіщень. Після підтвердження бронювання Booking Service активує Notification Service, передаючи йому всі необхідні деталі: ідентифікатори користувача та власника, дати бронювання, адресу нерухомості та контактну інформацію. Notification Service використовує чергу повідомлень в Redis для асинхронної обробки сповіщень, що дозволяє не блокувати основний потік виконання. Для кожного отримувача готуються персоналізовані email-повідомлення з деталями бронювання: користувач отримує підтвердження бронювання з інструкціями, а власник нерухомості – сповіщення про нове бронювання з контактними даними орендаря [8].

На рисунку 2.2 представлено детальну діаграму компонентів пошукового сервісу, яка демонструє внутрішню архітектуру та взаємодію всіх складових частин спеціалізованого мікросервісу, відповідального за обробку пошукових запитів у системі оренди житла. Ця діаграма візуалізує ключовий принцип гібридного пошуку, що поєднує геопросторовий пошук через PostGIS та повнотекстовий пошук через Elasticsearch.

Центральним компонентом архітектури є Search Engine, який виконує роль координатора всіх пошукових операцій. Цей компонент отримує пошукові запити від API Controller та організовує їх паралельну або послідовну обробку через спеціалізовані модулі. Search Engine реалізує складну логіку об’єднання результатів від різних джерел пошуку, застосовуючи алгоритми ранжування та релевантності, що враховують географічну близькість, цінові параметри, кількість зручностей та інші критерії. Важливою функцією Search Engine є управління життєвим циклом

кешування – визначення того, які запити варто кешувати, на який термін та коли інвалідувати закешовані дані [9].

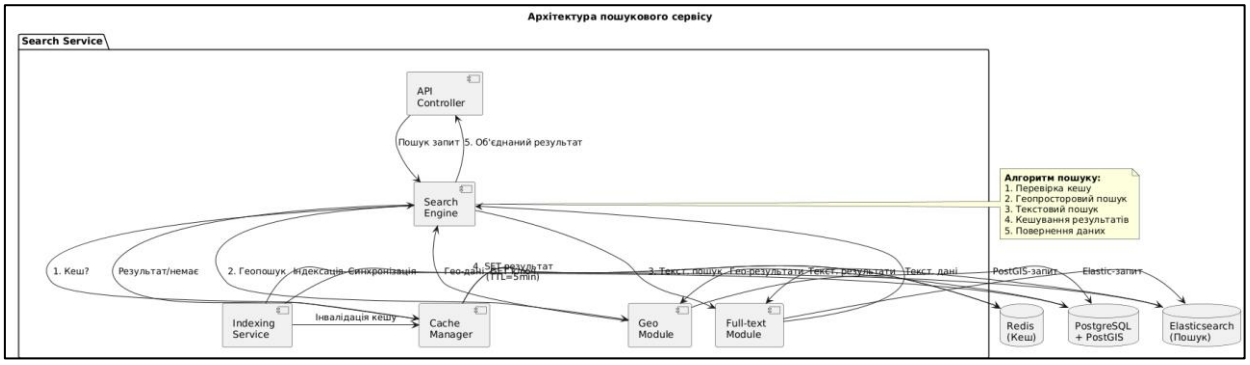


Рисунок 2.2 – Діаграма компонентів пошукового сервісу

Геопросторовий модуль (Geospatial Module) відповідає за обробку запитів, пов’язаних із фізичним розташуванням нерухомості. Цей модуль використовує потужні можливості розширення PostGIS для PostgreSQL, що дозволяє виконувати складні просторові запити. Серед типових операцій, які реалізує модуль: пошук нерухомості в межах заданого радіусу від певної точки (функція ST_DWithin), обчислення точної відстані між об’єктами (ST_Distance), пошук у межах полігону (ST_Within) та обробка шарів геоданих. Модуль оптимізований для роботи з просторовими індексами типу GiST (Generalized Search Tree), що забезпечують високу швидкість виконання запитів навіть при роботі з мільйонами географічних точок [9].

Повнотекстовий пошуковий модуль (Full-text Module) спеціалізується на обробці текстових пошукових запитів – пошуку за ключовими словами в назвах, описів нерухомості, переліку зручностей тощо. Цей модуль інтегрується з Elasticsearch, розподіленою пошуковою системою, що забезпечує миттєву індексацію та пошук. Elasticsearch використовує інвертовані індекси для ефективного пошуку слів, підтримує морфологічний аналіз для різних мов, синоніми, fuzzy-пошук для виправлення опечаток та складні булеві запити. Модуль також відповідає за налаштування аналізаторів токенів, що дозволяють адаптувати пошук під специфічні потреби ринку нерухомості [9].

Система кешування (Cache Manager) грає критично важливу роль у забезпеченні високої продуктивності пошукового сервісу. Цей компонент реалізує двоступеневу стратегію кешування: на рівні окремих запитів та на рівні проміжних результатів. Cache Manager визначає, чи можна використати закешовані результати для поточного запиту на основі аналізу параметрів пошуку, часу життя кешу та частоти оновлення відповідних даних. Для зберігання кешованих даних використовується Redis – швидке сховище ключ-значення в оперативній пам'яті. Стратегія кешування включає TTL (Time To Live) для автоматичного очищення застарілих даних та механізми інвалідації кешу при оновленні інформації про нерухомість [9].

Сервіс індексації (Indexing Service) забезпечує синхронізацію даних між основною базою даних PostgreSQL та пошуковим індексом Elasticsearch. Цей компонент працює в двох основних режимах: повна індексація (при первинному наповненні системи або переіндексації) та інкрементальна індексація (при оновленні окремих записів). Indexing Service відстежує зміни в таблицях нерухомості через тригери бази даних або подійну систему, витягує змінні дані, трансформує їх у формат, прийнятний для Elasticsearch, та відправляє на індексацію. Компонент також реалізує механізми відновлення після збоїв та контролю узгодженості даних між різними сховищами [9].

API Controller виступає як публічний інтерфейс пошукового сервісу, надаючи REST API для зовнішніх клієнтів. Цей компонент відповідає за валідацію вхідних параметрів, обробку аутентифікації та авторизації запитів, нормалізацію даних та формування стандартизованих відповідей. API Controller реалізує підтримку різноманітних форматів запитів: від простих текстових пошуків до складних комбінованих фільтрів з географічними обмеженнями, ціновими діапазонами та критеріями зручностей [9].

Процес обробки пошукового запиту починається з отримання запиту через API Controller, який перевіряє його коректність та перетворює у внутрішній формат. Далі запит передається до Search Engine, який першим ділом звертається до Cache Manager для перевірки наявності закешованих

результатів. У разі попадання в кеш, результати негайно повертаються клієнту, що дозволяє досягти часу відповіді менше 50 мілісекунд.

На рисунку 2.3 представлено діаграму розгортання, яка ілюструє фізичну та логічну організацію компонентів системи пошуку та оренди житла в продакшен-середовищі. Діаграма демонструє розподіл мікросервісів, баз даних та зовнішніх залежностей по віртуальних та фізичних серверах [9].

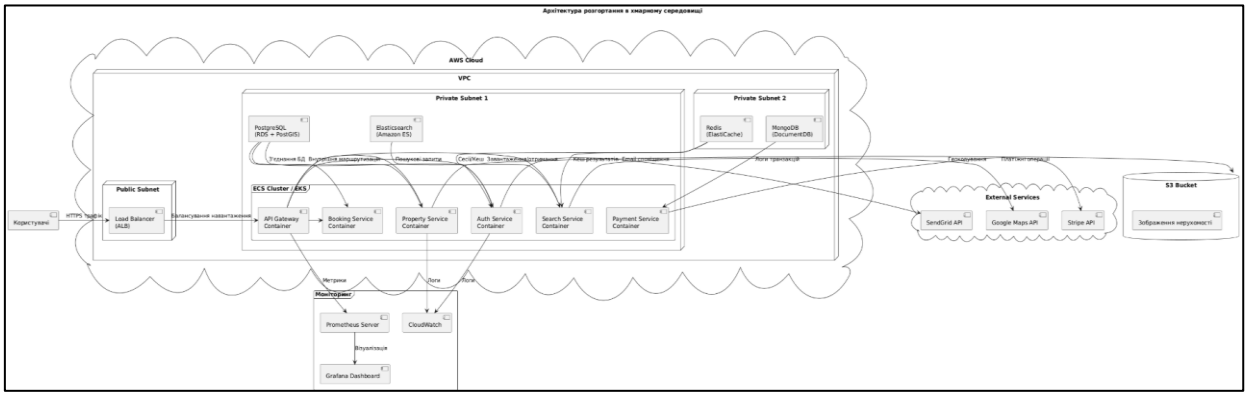


Рисунок 2.3 – Діаграма розгортання

Інфраструктурна архітектура базується на контейнеризації за допомогою Docker та оркестрації через Kubernetes. Кластер Kubernetes складається з master-ноди для управління та кількох worker-нод для запуску робочих навантажень. Master-нода відповідає за планування подів, управління конфігурацією, моніторинг стану та автоматичне масштабування. Worker-ноди розподілені між кількома фізичними серверами або віртуальними машинами для забезпечення високої доступності та відмовостійкості [9].

Мікросервіси розгорнуті в окремих контейнерах, згрупованих у Kubernetes Pods. Кожен Pod містить основний контейнер з мікросервісом та допоміжні sidecar-контейнери для логування, моніторингу та обробки конфігурації. Використання Kubernetes Deployments забезпечує декларативне управління версіями, автоматичне відновлення при збоях та можливість плавного оновлення через rolling updates. Сервіси експонуються через Kubernetes Services з типами LoadBalancer для зовнішнього доступу та ClusterIP для внутрішньої комунікації [10].

Бази даних розгорнуті в окремому сегменті інфраструктури для забезпечення ізоляції та безпеки.

PostgreSQL з розширенням PostGIS працює на виділених віртуальних машинах з реплікацією master-slave. Master-інстанс обробляє запити на запис, тоді як кілька slave-інстансів обслуговують читання, що забезпечує масштабування та підвищення продуктивності. Автоматичне перемикання при відмові master-ноди реалізоване через Patroni або Stolon. Для зберігання даних використовуються виділені SSD-диски з регулярним резервним копіюванням на окреме сховище.

Elasticsearch кластер розгорнуто на окремих вузлах з виділеними ролями master, data та ingest. Master-вузли відповідають за управління кластером, data-вузли зберігають індекси, а ingest-вузли обробляють вхідні дані для індексації. Кластер налаштовано на автоматичне розподілення шардів та реплік між різними фізичними серверами для забезпечення відмовостійкості та збалансованого навантаження. Моніторинг кластера здійснюється через Kibana та Prometheus експортери [10].

Redis розгорнуто у кластерній конфігурації з автоматичною сегментацією даних та реплікацією. Кластер складається з кількох master-нод, кожна з яких має одну або кілька slave-нод для реплікації. Redis Sentinel або Redis Cluster забезпечують автоматичне виявлення відмов та перемикання. Для збереження даних при перезавантаженні налаштовано персистентність через RDB снапшоти та AOF лог.

Моніторинг та логування реалізовані через централізовану систему на основі Prometheus, Grafana та ELK-стека. Prometheus збирає метрики з усіх мікросервісів, баз даних та системних компонентів через експортери. Алерт-менеджер налаштований на відправку сповіщень при порушенні порогових значень. Grafana дашборди надають реальний час огляду стану системи, продуктивності та бізнес-метрик. ELK-стек (Elasticsearch, Logstash, Kibana) збирає та аналізує логи з усіх компонентів системи [10].

2.3 Проєктування схеми бази даних PostgreSQL

Проєктування бази даних є одним із ключових етапів створення системи пошуку та оренди житла, оскільки саме структура збереження даних визначає цілісність інформації, ефективність доступу до неї та можливості подальшого масштабування всієї архітектури. На цьому етапі формується логічна модель даних, визначаються сутності предметної області, їх атрибути та взаємозв'язки в контексті ринку нерухомості та орендних транзакцій. Коректно спроєктована база даних забезпечує надійне зберігання критичних даних про нерухомість, користувачів та фінансові операції, підтримує виконання складних бізнес-процесів бронювання та оплати, і мінімізує ризик виникнення аномалій під час вставки, оновлення або видалення інформації в умовах високого конкурентного навантаження [10].

У межах системи пошуку та оренди житла база даних повинна зберігати комплексну інформацію про кілька ключових аспектів: дані користувачів системи (орендарів, орендодавців, адміністраторів) з їх ролями та правами доступу; детальні описи об'єктів нерухомості з географічними координатами, фотографіями, переліком зручностей та правил; календар доступності кожної нерухомості; історію бронювань та фінансових транзакцій; систему відгуків та рейтингів; а також логі комунікації між учасниками платформи. Модель даних має підтримувати сценарії багатокористувацької роботи з чітким розмежуванням доступу до конфіденційної інформації, що досягається шляхом встановлення відповідних зв'язків між сутностями та реалізації механізмів авторизації на рівні запитів до бази даних [10].

Структура бази даних є фундаментальною частиною будь-якої інформаційної системи, що забезпечує зберігання, керування та доступ до даних. У цьому проєкті використовується реляційна база даних PostgreSQL, спроєктована за допомогою ER-діаграми (діаграми «сутність-зв'язок»), що ілюструє логічні зв'язки між різними компонентами системи оренди житла. У контексті реляційних баз даних зв'язок між двома таблицями

характеризується кардинальністю відношень, що визначає кількість відповідних записів між пов’язаними сутностями. Для системи оренди житла ключовими є відношення між користувачами та нерухомістю, нерухомістю та бронюваннями, а також користувачами та транзакціями [10].

Найбільш поширеними типами відношень для реалізації логіки системи є зв’язок «один-до-багатьох», який дозволяє одному орендодавцю володіти необмеженою кількістю об’єктів нерухомості, та «багато-до-багатьох», що реалізується через проміжні таблиці для зв’язку бронювань із користувачами та нерухомістю. Для технічної реалізації такої структури використовуються зовнішні ключі, що гарантують посилальну цілісність і унеможливлюють існування бронювань без прив’язки до конкретної нерухомості та користувача. Важливим аспектом проєктування також є нормалізація таблиць до відповідних нормальних форм, що допомагає уникнути дублювання даних, зменшує ймовірність логічних помилок при їх модифікації та забезпечує консистентність даних при паралельних операціях [10].

Особливістю проєктування бази даних для системи пошуку житла є необхідність підтримки геопросторових запитів через розширення PostGIS. Це вимагає спеціалізованих типів даних (GEOGRAPHY, GEOMETRY) та індексів (GIST) для ефективного виконання запитів на відстань, пошуку в радіусі та інших просторових операцій. Додатково використовуються JSONB поля для зберігання напівструктурованих даних, таких як динамічні списки зручностей, правила будинку або конфігураційні параметри, що забезпечує гнучкість схеми без необхідності постійних модифікацій структури бази даних [11].

Окрім структуризації даних, схема бази передбачає створення спеціалізованих індексів для ключових полів, що критично необхідно для швидкого пошуку та сортування інформації у великих масивах даних, характерних для ринку нерухомості. Використовуються різні типи індексів: B-tree для стандартних полів фільтрації, GIST для геопросторових даних, GIN для JSONB полів та повнотекстового пошуку, BRIN для великих хронологічних таблиць. Також реалізуються часткові індекси для оптимізації

запитів до активних записів та унікальні обмеження для запобігання конфліктам даних [11].

Архітектура бази даних також враховує аспекти безпеки та відмовостійкості: реалізуються механізми резервного копіювання та реплікації, налаштовуються рівні ізоляції транзакцій для уникнення race conditions при одночасних бронюваннях, та використовуються тригери та правила для підтримки цілісності даних у розподіленій мікросервісній архітектурі. Усі ці компоненти разом формують надійну, масштабовану та високопродуктивну основу для системи пошуку та оренди житла, здатну ефективно обслуговувати тисячі користувачів та обробляти мільйони запитів до геопросторових даних.

На рисунку 2.4 представлено комплексну ER-діаграму (Entity-Relationship Diagram) бази даних системи пошуку та оренди житла, яка візуалізує логічну структуру всіх сутностей, їх атрибутів та взаємозв’язків. Діаграма демонструє, як організовані дані для підтримки всіх бізнес-процесів платформи, від управління користувачами до складних транзакцій бронювання та геопросторового пошуку [11].

Основу структури складають вісім ключових сутностей, які формують ядро предметної області системи оренди житла. Центральним елементом є сутність “users”, що містить інформацію про всіх учасників системи: орендарів, орендодавців та адміністраторів. Для кожного користувача зберігаються обов’язкові дані аутентифікації (email, password_hash), особиста інформація (first_name, last_name, phone), стан облікового запису (is_verified, is_active) та часові мітки для відстеження активності. Роль користувача визначається полем “role” з обмеженим набором значень, що впливає на доступні функції та права в системі.

Сутність “properties” є найбільш складною та інформаційно насиченою, оскільки містить повний опис об’єктів нерухомості, доступних для оренди. Кожен об’єкт характеризується базовими параметрами: типом нерухомості (apartment, house, villa, room), кількістю спалень та ванних кімнат, площею,

максимальною кількістю гостей. Критично важливим атрибутом є “location” типу GEOGRAPHY(Point, 4326), який зберігає географічні координати об’єкта та дозволяє виконувати просторові запити через PostGIS. Для зберігання динамічних та структурованих даних використовуються JSONB поля: “address” для повної адреси, “amenities” для списку зручностей, “house_rules” для правил проживання. Ці поля забезпечують гнучкість без необхідності зміни схеми бази даних при додаванні нових атрибутів [11].

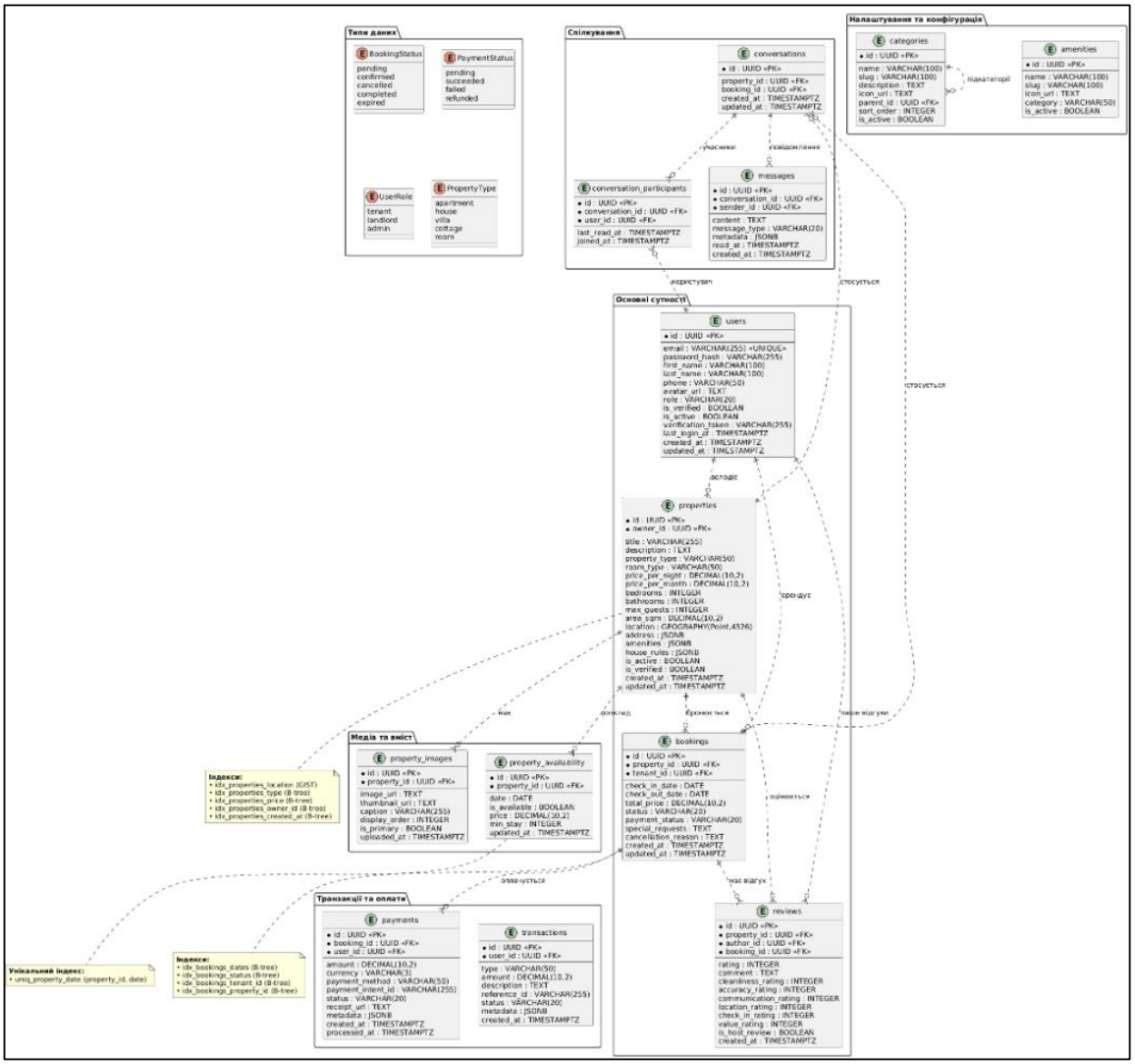


Рисунок 2.4 – Діаграма бази даних системи пошуку та оренди житла

Система бронювань реалізована через сутність “bookings”, яка встановлює зв’язки між користувачами та нерухомістю. Кожне бронювання

має обов’язкові дати заїзду та виїзду (check_in_date, check_out_date), розраховану загальну вартість (total_price), статус (pending, confirmed, cancelled, completed) та статус оплати. Особливістю є наявність поля “special_requests” для додаткових побажань орендаря та “cancellation_reason” для аналітики скасувань. Для гарантії цілісності даних реалізовані зовнішні ключі до таблиць “users” (tenant_id) та “properties” (property_id), що унеможливлює існування бронювань без чітко визначених учасників [11].

Медіа-вміст управляється через сутність “property_images”, яка дозволяє зберігати необмежену кількість фотографій для кожного об’єкта нерухомості. Кожне зображення має основне посилання (image_url), посилання на мініатюру (thumbnail_url) для оптимізації завантаження, опис (caption) та порядковий номер для відображення (display_order). Прапор “is_primary” визначає головне зображення, яке використовується в прев’ю пошуку. Ця структура дозволяє ефективно керувати візуальним контентом та оптимізувати завантаження сторінок через lazy loading мініатюр [11].

Календар доступності реалізований через сутність “property_availability”, яка забезпечує динамічне управління вільними датами та цінами для кожної нерухомості. Для кожної дати зберігається прапор доступності (is_available), актуальна ціна (price) та мінімальний термін оренди (min_stay). Унікальний індекс на полях (property_id, date) гарантує відсутність дублікатів дат для одного об’єкта та забезпечує швидкий пошук доступних дат. Ця архітектура дозволяє орендодавцям гнучко керувати розкладом, встановлювати сезонні ціни та блокувати дати для технічного обслуговування.

2.4 Проєктування серверної частини

Проєктування серверної частини системи пошуку та оренди житла передбачає розробку архітектури, що забезпечує високу продуктивність, масштабованість, безпеку та обробку складних бізнес-процесів. Серверна

частина реалізується на основі мікросервісної архітектури з використанням Node.js та Express.js, що дозволяє ефективно розподіляти навантаження та незалежно масштабувати окремі компоненти системи. На рисунку 2.5 представлено детальну структуру проєкту серверної частини системи пошуку та оренди житла, яка ілюструє організацію коду за принципами модульної архітектури та розділення відповідальностей. Структура побудована з використанням шаблону “мікросервіс в моноліті” (modular monolith), що дозволяє зберігати переваги монолітної архітектури для розробки з можливістю подальшого виділення окремих модулів у самостійні мікросервіси [11].

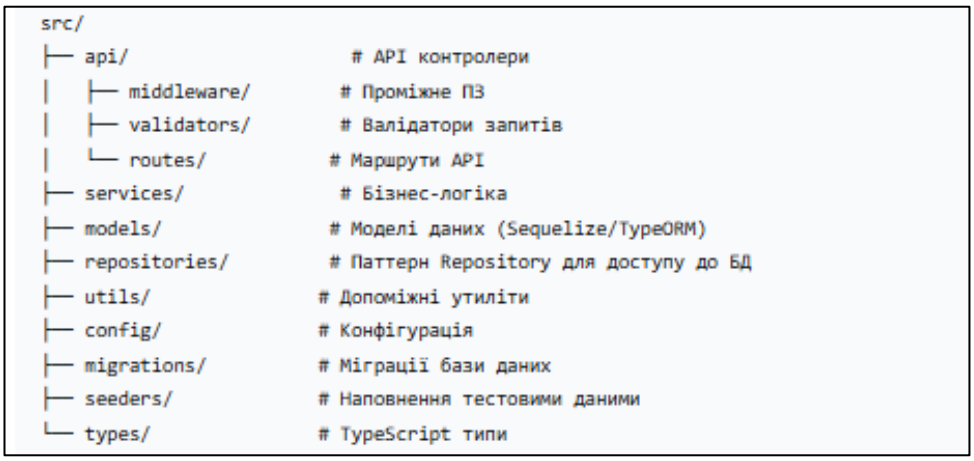


Рисунок 2.5 – Структура проєкту

Коренева структура проєкту організована у каталозі housing-rental-api/, який містить всі необхідні компоненти для розгортання, розробки та підтримки системи. Основним каталогом є src/, що включає всю бізнес-логіку та API додатку. Каталог tests/ містить тести різних рівнів зі структурою, що віддзеркалює основний код, що спрощує навігацію та підтримку тестового покриття. Для контейнеризації проєкту використовується каталог docker/ з конфігураційними файлами Docker та Docker Compose. Технічна документація, API специфікації та інструкції щодо розгортання зберігаються в каталозі docs/. Файл package.json містить всі залежності Node.js та скрипти для розробки, тестування та деплою [11].

Каталог src/ є ядром проєкту та розділений на кілька логічних груп. Каталог config/ містить усю конфігурацію додатку, розподілену за середовищами виконання. Тут знаходяться окремі файли для розробки, продакшену та тестування, а також специфічні налаштування для бази даних, кешу, аутентифікації та зовнішніх сервісів. Каталог common/ включає компоненти, що використовуються в різних модулях системи: TypeScript декоратори для валідації та авторизації, перехоплювачі для обробки запитів, фільтри для обробки винятків, охоронці для контролю доступу та проміжне ПЗ для CORS, компресії та rate-limiting [12].

Найважливішою частиною структури є каталог modules/, де кожен підкаталог представляє окремий бізнес-модуль системи. Модуль auth/ відповідає за аутентифікацію та авторизацію користувачів, містить контролери API, бізнес-логіку, репозиторії для доступу до даних, сутності бази даних та стратегії автентифікації. Модуль properties/ управляє об’єктами нерухомості, включаючи CRUD операції, обробку зображень та роботу з геопросторовими даними через PostGIS. Модуль search/ реалізує гібридний пошук, поєднуючи можливості PostGIS для геопошуку та Elasticsearch для повнотекстового пошуку [12].

Модуль bookings/ відповідає за всю логіку бронювань, включаючи створення, підтвердження, скасування бронювань та управління календарем доступності. Модуль payments/ інтегрується з платіжними системами та обробляє всі фінансові транзакції. Модуль notifications/ забезпечує відправку email, SMS та push-сповіщень через асинхронні черги. Модуль chat/ реалізує систему комунікації між користувачами через REST API та WebSocket. Модуль reviews/ управляє відгуками та рейтингами, включаючи розрахунок середніх оцінок та модерацію контенту [12].

Каталог shared/ містить спільні сервіси та утиліти, що використовуються в кількох модулях. Тут знаходяться обгортки для роботи з PostgreSQL та Redis, абстракції для відправки email, сервіси для роботи з S3-сумісним сховищем файлів, геокодування адрес, централізована система логування на основі

Winston та спільні схеми валідації. Каталог utils/ включає допоміжні утиліти: функції для роботи з датами та рядками, форматтери для цін і адрес, генератори унікальних ідентифікаторів та токенів, специфічні валідатори для бізнес-логіки та калькулятори цін, комісій та податків [12].

Каталог core/ містить фундаментальні компоненти системи, що формують основу архітектури. Тут знаходяться базові інтерфейси та типи TypeScript, абстрактні класи для уніфікації поведінки сервісів, кастомні винятки для обробки бізнес-помилок, системні декоратори та компоненти для реалізації пагінації. Каталог app/ включає конфігурацію додатку: кореневий модуль, який імпортує всі функціональні модулі, кореневий контролер для health checks та системних ендпоінтів, та кореневий сервіс для загальної логіки. Файл main.ts слугує точкою входу в додаток, де відбувається ініціалізація Express/Fastify, підключення middleware, реєстрація модулів та запуск сервера [12].

Каталог tests/ організований за типами тестування. Підкаталог unit/ містить модульні тести окремих компонентів, integration/ – інтеграційні тести взаємодії між модулями, e2e/ – end-to-end тести повних сценаріїв використання. Каталог fixtures/ включає тестові дані для різних сценаріїв, mocks/ – імітації зовнішніх сервісів для ізоляції тестів, а setup/ – налаштування тестового середовища та глобальні хуки.

Каталог docker/ забезпечує контейнеризацію проєкту. Підкаталог development/ містить Docker конфігурацію для середовища розробки, production/ – оптимізовані Dockerfile для продакшену, compose/ – Docker Compose файли для оркестрації всіх сервісів, а nginx/ – конфігурацію reverse proxy для балансування навантаження та SSL термінації [12].

Ключовою особливістю структури є дотримання принципів чистої архітектури. Кожен модуль має чітко визначену відповідальність та мінімальну залежність від інших модулів. Використовуються шаблони проєктування «Репозиторій» для абстракції доступу до даних, «Сервіс» для інкапсуляції бізнес-логіки та «DTO» для типобезпечної передачі даних між

шарами. Middleware паттерн застосовується для перехоплення та обробки запитів на різних етапах виконання [12].

Архітектура забезпечує високу масштабованість та гнучкість. Кожен модуль може бути легко виділений в окремий мікросервіс при зростанні навантаження або необхідності незалежного розгортання. Незалежне тестування модулів дозволяє підтримувати високе якість коду, а можливість заміни конкретних реалізацій без змін інтерфейсів забезпечує адаптивність до змін технологічного стеку. Така структура спрощує розподіл завдань між командами розробників та підтримує високу якість кодової бази протягом всього життєвого циклу проєкту системи пошуку та оренди житла [12].

На рисунку 2.6 представлено комплексну діаграму компонентів серверної частини системи, яка візуалізує організацію основних модулів, їх взаємозв’язки та відповідальності в рамках архітектури додатку. Діаграма демонструє, як система розділена на логічно окремі компоненти, кожен з яких відповідає за певну бізнес-домену та взаємодіє з іншими компонентами через чітко визначені інтерфейси.

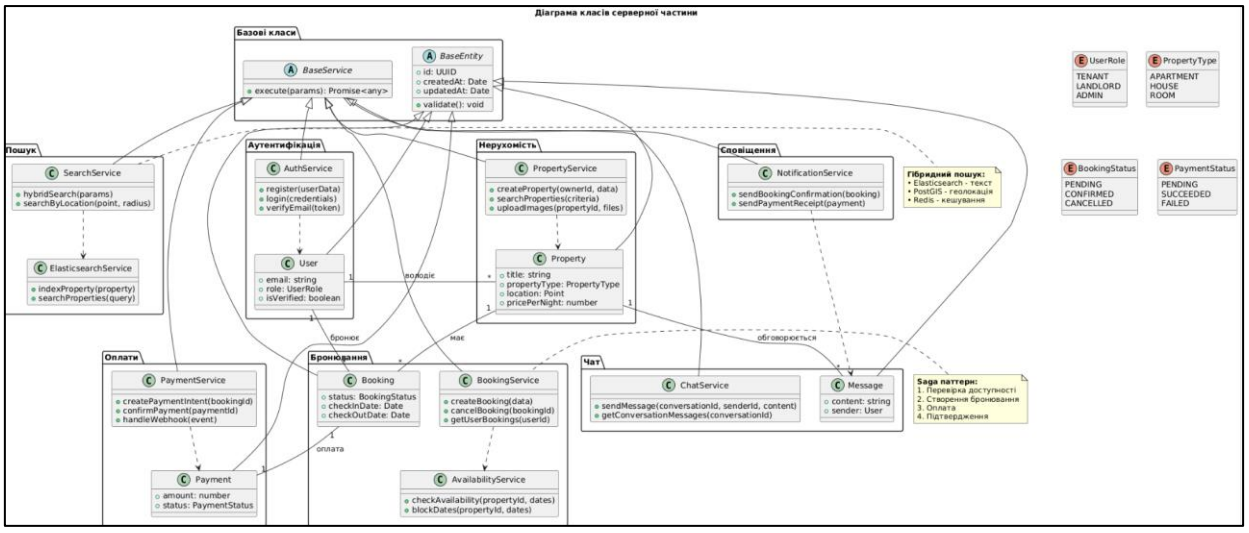


Рисунок 2.6 – Діаграма серверної частини

Фундаментом системи є базові абстрактні класи BaseEntity та BaseService, які визначають спільну структуру та поведінку для всіх компонентів. BaseEntity містить універсальні поля, такі як унікальний

ідентифікатор, часові мітки створення та оновлення, а також абстрактний метод валідації, що гарантує цілісність даних. BaseService надає базовий шаблон для всіх сервісних класів, включаючи механізми виконання операцій, обробки помилок та логування. Ці абстрактні класи забезпечують узгодженість архітектури та дозволяють уникнути дублювання коду в похідних компонентах [12].

Модуль аутентифікації складається з двох ключових компонентів: User та AuthService. Клас User представляє сутність користувача системи, зберігаючи інформацію про email, роль (орендар, орендодавець, адміністратор), статус верифікації та інші атрибути. AuthService відповідає за всю бізнес-логіку автентифікації та авторизації, включаючи реєстрацію нових користувачів, вхід у систему, верифікацію email, оновлення паролів та управління сесіями. Цей модуль забезпечує безпеку системи та контроль доступу до функціоналу відповідно до ролей користувачів.

Модуль управління нерухомістю представлений компонентами Property та PropertyService. Клас Property інкапсулює всі дані про об’єкт нерухомості: назву, тип (квартира, будинок, кімната), географічні координати, ціну за ніч, кількість спалень та ванних кімнат. Особливістю є зберігання географічних координат у форматі Point, що дозволяє виконувати просторові запити через PostGIS. PropertyService реалізує CRUD операції для нерухомості, обробку зображень, управління доступністю та пошук об’єктів за різними критеріями [12].

Пошуковий модуль включає SearchService та ElasticsearchService для реалізації гібридної системи пошуку. SearchService координує процес пошуку, комбінуючи результати від різних джерел: геопросторовий пошук через PostGIS, повнотекстовий пошук через Elasticsearch та кешування результатів через Redis. ElasticsearchService слугує адаптером для взаємодії з Elasticsearch, інкапсулюючи логіку індексації нерухомості, виконання пошукових запитів та управління пошуковими індексами. Така архітектура забезпечує високу продуктивність пошуку навіть при великих обсягах даних [12].

Модуль бронювань складається з трьох взаємопов'язаних компонентів: Booking, BookingService та AvailabilityService. Клас Booking представляє сутність бронювання з полями для дат заїзду/виїзду, статусу (очікує, підтверджено, скасовано) та зв'язками з нерухомістю та користувачем. BookingService управляє життєвим циклом бронювань, реалізуючи створення, підтвердження, скасування та продовження бронювань з дотриманням бізнес-правил. AvailabilityService спеціалізується на управлінні календарем доступності кожної нерухомості, перевірці вільних дат та блокуванні дат під час активних бронювань.

Платіжний модуль включає Payment та PaymentService для обробки фінансових транзакцій. Клас Payment містить інформацію про суму платежу, статус (очікує, успішний, неуспішний, повернений), метод оплати та ідентифікатор транзакції в зовнішній системі. PaymentService інтегрується з платіжними шлюзами (такими як Stripe), створює платежні наміри, обробляє вебхуки від платіжних систем та керує процесами повернення коштів. Цей модуль забезпечує безпечну обробку фінансових операцій та відповідність вимогам PCI DSS [12].

Модуль сповіщень представлений NotificationService, який відповідає за відправку повідомлень користувачам через різні канали: електронну пошту, SMS та push-сповіщення. Сервіс генерує персоналізовані повідомлення для різних подій: підтвердження бронювання, квитанції про оплату, нагадування про відгуки, сповіщення про нові повідомлення в чаті. Використання шаблонів повідомлень та асинхронної обробки через черги забезпечує надійну доставку сповіщень без впливу на продуктивність основного додатку [12].

Модуль комунікації складається з Message та ChatService для реалізації системи чату. Клас Message представляє окреме повідомлення в чаті з полями для контенту, відправника, часу відправлення та статусу прочитання. ChatService керує діалогами між користувачами: створює нові розмови, відправляє повідомлення, отримує історію повідомлень та відзначає повідомлення як прочитані. Модуль підтримує як приватні діалоги між двома користувачами, так і групові чати з кількома учасниками [12].

Відносини між компонентами будуються на принципах спадкування, асоціації та залежності. Спадкування від базових класів забезпечує узгодженість архітектури та повторне використання коду. Асоціації між сутностями (наприклад, зв’язок «користувач володіє нерухомістю» або «нерухомість має бронювання») відображають бізнес-логіку предметної області. Залежності між сервісами показують, як компоненти взаємодіють для виконання складних операцій: наприклад, BookingService залежить від AvailabilityService для перевірки доступності дат, а PaymentService залежить від Booking для отримання інформації про бронювання [12].

2.5 Висновки до розділу 2

В другому розділі роботи було виконано повний цикл проєктування серверної частини системи пошуку та оренди житла, результати якого становлять теоретичну основу для практичної реалізації.

Проведений аналіз вимог дозволив чітко визначити функціональні можливості системи та критерії її ефективності, зокрема щодо обробки геопросторових даних та транзакційної цілісності. Обґрунтування технологічного стеку показало оптимальність комбінації Node.js, PostgreSQL та мікросервісної архітектури для забезпечення високої продуктивності та масштабованості системи. Проєктування схеми бази даних призвело до створення оптимізованої моделі з використанням геопросторових типів даних для координат та JSONB полів для динамічних атрибутів, що забезпечує ефективну роботу з пошуковими запитами. Архітектурне проєктування визначило модульну структуру системи з чітким розподілом відповідальностей між сервісами аутентифікації, управління нерухомістю, пошуку, бронювання та оплат. Структурна організація проєкту заснована на принципах чистої архітектури, що забезпечує легкість розробки, тестування та подальшого масштабування системи.

Отримані результати формують міцну архітектурну основу для подальшої реалізації високопродуктивної, безпечної та масштабованої системи пошуку та оренди житла, що повністю відповідає встановленим функціональним та нефункціональним вимогам.

3 ПРОГРАМУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Налаштування середовища розробки та інфраструктури

Налаштування середовища розробки та інфраструктури є критично важливим першим кроком у процесі програмної реалізації серверної частини системи пошуку та оренди житла. Цей етап включає підготовку необхідних інструментів, конфігурацію систем залежностей, організацію структури проєкту та налаштування контейнеризованого середовища для забезпечення консистентності між різними етапами розробки. Спочатку було створено базову структуру проєкту Node.js з необхідними конфігураційними файлами.

На рисунку 3.1 представлено конфігураційні файли TypeScript та ESLint, які забезпечують типобезпеку, якість коду та стандартизацію розробки серверної частини системи пошуку та оренди житла.

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "lib": ["ES2022"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "moduleResolution": "node",
    "types": ["node", "jest"],
    "baseUrl": ".",
    "paths": {
      "@/*": ["src/*"],
      "@config/*": ["src/config/*"],
      "@modules/*": ["src/modules/*"],
      "@shared/*": ["src/shared/*"]
    }
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist", "tests"]
}
```

Рисунок 3.1 – Структура конфігурації файлів

Файл tsconfig.json визначає налаштування компілятора TypeScript для забезпечення строгої типізації та оптимальної продуктивності. Основним призначенням цієї конфігурації є забезпечення типобезпеки на етапі компіляції, що значно зменшує ймовірність помилок у виконанні та покращує якість коду. Ключовим параметром є “strict”: true, який активує всі строгі перевірки типів, включаючи обов’язкове оголошення типів для змінних, перевірку на null/undefined та інші важливі перевірки. Це гарантує, що код буде менш схильний до помилок часу виконання.

Важливим аспектом конфігурації є підтримка сучасних можливостей JavaScript через параметр “target”: “ES2022”, що дозволяє використовувати найновіші функції мови, такі як приватні поля класів, top-level await та інші покращення ES2022. Модульна система CommonJS ("module": "commonjs") обирається як найбільш сумісна з Node.js екосистемою. Для забезпечення чистоти архітектури налаштовані аліаси шляхів ("paths"), що дозволяє організовувати імпорти у вигляді @modules/auth замість відносних шляхів, що покращує читабельність та підтримку коду.

Експериментальні функції TypeScript активовані через параметри “experimentalDecorators”: true та “emitDecoratorMetadata”: true, що необхідно для роботи з TypeORM та іншими бібліотеками, які використовують декоратори для метапрограмування. Це дозволяє використовувати такі конструкції як @Entity(), @Column() для визначення моделей бази даних, що значно спрощує роботу з ORM. Налаштування "moduleResolution": "node" забезпечує правильне вирішення модулів у Node.js середовищі.

Файл .eslintrc.json містить правила для підтримки якості коду та дотримання кодстайлу проєкту. Використовується плагін @typescript-eslint для забезпечення коректної роботи ESLint з TypeScript. Правила поділені на кілька категорій: "recommended" для загальних рекомендацій, "plugin:@typescript-eslint/recommended" для специфічних перевірок TypeScript. Додатково включені правила для роботи з промісами, щоб запобігти поширеним помилкам асинхронного програмування.

Ключові правила ESLint включають вимогу явного повернення типів для функцій (explicit-function-return-type), що покращує читабельність та допомагає уникнути помилок типізації. Правило no-explicit-any обмежує використання типу any, сприяючи строгій типізації всього коду. Для забезпечення консистентності форматування використовуються правила щодо використання одинарних лапок, відступів та крапок з комою. Налаштовано автоматичне виправлення простих помилок через параметр --fix.

На рисунку 3.2 представлено конфігурацію Docker середовища для розробки серверної частини системи пошуку та оренди житла. Діаграма демонструє архітектуру контейнеризованого середовища, яка забезпечує консистентність розробки, простоти розгортання та ізоляцію залежностей проєкту.



```yaml
version: '3.8'

services:
  postgres:
    image: postgis/postgis:15-3.3
    container_name: housing-postgres
    environment:
      POSTGRES_DB: housing_rental
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secure_password
      POSTGRES_INITDB_ARGS: "--encoding=UTF8"
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d
    networks:
      - housing-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U admin"]
      interval: 10s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    container_name: housing-redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    networks:
      - housing-network
    command: redis-server --appendonly yes --requirepass redis_password

  elasticsearch:
    image: elasticsearch:8.8.0
    container_name: housing-elasticsearch
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ports:
      - "9200:9200"
      - "9300:9300"
    volumes:
      - elasticsearch_data:/usr/share/elasticsearch/data
    networks:
```

Рисунок 3.2 – Конфігурація Docker середовища

Основою конфігурації є файл docker-compose.yml, який описує чотири ключові сервіси системи: PostgreSQL з PostGIS, Redis, Elasticsearch та API додаток. Кожен сервіс має власні налаштування конфігурації, вольюми для збереження даних та специфічні параметри здоров'я для моніторингу стану. Сервіси об'єднані в єдину мережу housing-network, що забезпечує їхню взаємодію та ізоляцію від зовнішніх систем.

Сервіс PostgreSQL з PostGIS (postgres) використовує офіційний образ postgis/postgis:15-3.3, що забезпечує повну підтримку геопросторових операцій. Контейнер налаштований з параметрами середовища для автоматичного створення бази даних housing_rental, користувача admin та пароля. Особливістю є монтування двох вольюмів: postgres_data для збереження даних бази та ./init-scripts для виконання скриптів ініціалізації при першому запуску. Health check через pg_isready перевіряє доступність бази кожні 10 секунд.

Сервіс Redis (redis) розгортається на базі оптимізованого образу redis:7-alpine. Конфігурація включає включення persistence через --appendonly yes для гарантії збереження даних після перезавантаження та захист паролем через --requirepass redis_password. Redis слугує для кешування результатів пошуку, збереження сесій користувачів та роботи черг повідомлень. Використання alpine-версії забезпечує мінімальний розмір образу та швидкий запуск.

Сервіс Elasticsearch (elasticsearch) використовує версію 8.8.0 з вимкненим захистом (xpack.security.enabled=false) для спрощення розробки. Налаштовані обмеження пам'яті (-Xms512m -Xmx512m) для ефективного використання ресурсів на машинах розробників. Health check через curl перевіряє доступність REST API Elasticsearch кожні 30 секунд. Цей сервіс відповідає за повнотекстовий пошук та індексацію контенту нерухомості.

API додаток (api) будується з Dockerfile.dev, який оптимізований для розробки. Особливістю є монтування поточного коду (./src) та package.json безпосередньо в контейнер, що дозволяє здійснювати гаряче перезавантаження під час змін коду без перебудови образу. Залежності від

інших сервісів налаштовані через depends_on з умовами здоров'я, що гарантує послідовний запуск. Команда npm run dev запускає сервер у режимі розробки з nodemon для автоматичного перезапуску.

На рисунку 3.3 представлено конфігурацію та скрипти ініціалізації бази даних PostgreSQL для серверної частини системи пошуку та оренди житла. Діаграма демонструє архітектуру бази даних, необхідні розширення та початкові налаштування для ефективної роботи з геопросторовими даними та транзакціями оренди.

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "postgis";
CREATE EXTENSION IF NOT EXISTS "postgis_topology";
CREATE EXTENSION IF NOT EXISTS "fuzzystrmatch";
CREATE EXTENSION IF NOT EXISTS "pg_trgm";

-- Створення основних таблиць
CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(100) NOT NULL,
    last_name VARCHAR(100) NOT NULL,
    phone VARCHAR(50),
    role VARCHAR(20) NOT NULL CHECK (role IN ('tenant', 'landlord', 'admin')),
    is_verified BOOLEAN DEFAULT FALSE,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

-- Гібридний індекс для ефективного пошуку користувачів
CREATE INDEX IF NOT EXISTS idx_users_email_lower ON users (LOWER(email));
CREATE INDEX IF NOT EXISTS idx_users_phone ON users (phone);
```

Рисунок 3.3 – Конфігурація бази даних PostgreSQL

Основним компонентом конфігурації є SQL-скрипт ініціалізації, який автоматично виконується при першому запуску Docker-контейнера PostgreSQL. Скрипт розміщений у директорії init-scripts/ та монтується до контейнера через Docker Compose, що гарантує його виконання під час ініціалізації бази даних. Такий підхід забезпечує відтворюваність середовища та усуває необхідність ручного виконання скриптів налаштування.

Першим критично важливим кроком є активація необхідних розширень PostgreSQL. Розширення "uuid-ossp" дозволяє генерувати унікальні

ідентифікатори типу UUID, що використовуються як первинні ключі всіх сутностей системи для забезпечення глобальної унікальності та безпеки. Розширення "postgis" є ключовим для роботи з геопросторовими даними, надаючи підтримку геометричних типів даних, просторових індексів та функцій для обробки координат. "Postgis_topology" додає можливості для роботи з топологічними моделями, хоча в даному проєкті використовується обмежено.

Додаткові розширення "fuzzystrmatch" та "pg_trgm" покращують можливості текстового пошуку. "Fuzzystrmatch" надає функції для нечіткого пошуку та порівняння рядків, що корисне при пошуку за адресами з можливими помилками введення. "Pg_trgm" додає підтримку триграмного пошуку для ефективного пошуку за частковим збігом, що особливо важливо для функціоналу автодоповнення при пошуку нерухомості.

Створення таблиці користувачів (users) включає всі необхідні поля для роботи системи аутентифікації. Поле email має унікальне обмеження та валідацію на рівні бази даних через тип VARCHAR(255). password_hash зберігає захешовані паролі з використанням bcrypt. Поле role використовує перевірочне обмеження (CHECK) для обмеження допустимих значень до трьох ролей: орендар, орендодавець та адміністратор. Такий підхід забезпечує цілісність даних на рівні бази без необхідності додаткової валідації в коді.

На рисунку 3.4 представлено повну конфігурацію середовища розробки серверної частини системи пошуку та оренди житла. Діаграма демонструє архітектурні компоненти та їх взаємозв’язки в локальному середовищі розробки, що забезпечує консистентність та відтворюваність на всіх робочих станціях розробників.

Основним елементом конфігурації є файл environment.ts, який централізовано управляє всіма змінними середовища для різних стадій розробки та розгортання. Використання TypeScript для конфігурації замість традиційних .env файлів забезпечує типобезпеку та автоматичне завершення коду в IDE, що значно знижує ймовірність помилок при зверненні до

конфігураційних параметрів. Модуль dotenv завантажує змінні з .env файлу та робить їх доступними через process.env, але основний інтерфейс надається через типізований об’єкт environment.

```
import * as dotenv from 'dotenv';

dotenv.config();

export const environment = {
  nodeEnv: process.env.NODE_ENV || 'development',
  port: parseInt(process.env.PORT || '3000'),

  database: {
    host: process.env.DB_HOST || 'localhost',
    port: parseInt(process.env.DB_PORT || '5432'),
    username: process.env.DB_USERNAME || 'admin',
    password: process.env.DB_PASSWORD || 'secure_password',
    database: process.env.DB_DATABASE || 'housing_rental',
  },

  redis: {
    host: process.env.REDIS_HOST || 'localhost',
    port: parseInt(process.env.REDIS_PORT || '6379'),
    password: process.env.REDIS_PASSWORD || 'redis_password',
  },

  elasticsearch: {
    url: process.env.ELASTICSEARCH_URL || 'http://localhost:9200',
  },

  jwt: {
    secret: process.env.JWT_SECRET || 'default_jwt_secret',
    expiresIn: process.env.JWT_EXPIRES_IN || '1d',
    refreshExpiresIn: process.env.JWT_REFRESH_EXPIRES_IN || '7d',
  },

  stripe: {
    secretKey: process.env.STRIPE_SECRET_KEY || '',
    webhookSecret: process.env.STRIPE_WEBHOOK_SECRET || '',
  },

  email: {
    host: process.env.EMAIL_HOST || 'smtp.gmail.com',
    port: parseInt(process.env.EMAIL_PORT || '587'),
    user: process.env.EMAIL_USER || '',
    password: process.env.EMAIL_PASSWORD || '',
```

Рисунок 3.4 – Конфігурація середовища розробки

Структура конфігурації розділена на логічні секції для різних сервісів системи. Секція database містить параметри підключення до PostgreSQL: хост, порт, ім’я користувача, пароль та назву бази даних. Значення за

замовчуванням налаштовані для локальної розробки з використанням Docker Compose. Для продакшен-середовищ ці значення перевизначаються через змінні оточення на сервері розгортання.

Конфігурація Redis включає параметри для підключення до кеш-сервера: хост, порт та пароль для аутентифікації. Особливістю є підтримка кластерного режиму Redis, хоча для розробки використовується одинарний інстанс. Налаштування підключення оптимізовані для автоматичного перепідключення при тимчасових мережевих проблемах з експоненційною затримкою між спробами.

Elasticsearch конфігурація містить URL для підключення до пошукового кластера. Для розробки використовується локальний інстанс, тоді як для продакшену вказується адреса кластера в хмарній інфраструктурі. Додаткові параметри включають таймаути запитів, кількість повторних спроб та налаштування безпеки для продуктивних середовищ.

JWT (JSON Web Tokens) конфігурація містить секретний ключ для підпису токенів, час життя access та refresh токенів. Для безпеки в продакшен-середовищі секретний ключ генерується автоматично при кожному розгортанні та зберігається в секретному сховищі. Час життя токенів налаштований відповідно до політик безпеки: короткий для access токенів (1 день) та довший для refresh токенів (7 днів).

Інтеграція з платіжними системами через секцію stripe містить секретний ключ API та секрет для вебхуків. Для розробки використовуються тестові ключі Stripe, що дозволяє імітувати платежі без реальних фінансових операцій. Вебхук-секрет використовується для верифікації подій від Stripe, запобігаючи спробам шахрайських запитів.

Email конфігурація налаштовує параметри SMTP сервера для відправки сповіщень користувачам. Для розробки може використовуватись локальний SMTP сервер або сервіси на кшталт Mailtrap для перехоплення листів. У продакшені конфігурація включає TLS шифрування, аутентифікацію та додаткові параметри для підвищення deliverability.

Система логування налаштована через змінну LOG_LEVEL, що контролює деталізацію логів. Для розробки використовується рівень debug для максимальної інформативності, тоді як у продакшені – info або warn для зменшення обсягу логів. Додатково налаштовані формати логування: JSON для автоматичної обробки та кольорове виведення для консолі розробника.

Конфігурація безпеки включає параметри CORS (Cross-Origin Resource Sharing) для контролю доступу до API з різних джерел. Для розробки дозволено підключення з локального фронтенду, тоді як у продакшені вказуються конкретні домени. Додатково налаштовані обмеження розміру запитів, таймаути відповідей та параметри стиснення даних.

Налаштування середовища виконання містить порт сервера (port) та режим роботи (nodeEnv). Для розробки сервер працює на порту 3000 з режимом development, що включає додаткові перевірки, гаряче перезавантаження та детальні повідомлення про помилки. У продакшені використовується порт 80 або 443 з режимом production, що максимізує продуктивність.

3.2 Розробка сервісу аутентифікації та авторизації

Розробка сервісу аутентифікації та авторизації є фундаментальним компонентом системи пошуку та оренди житла, оскільки забезпечує безпеку даних користувачів, контроль доступу до функціоналу та управління сесіями. Цей сервіс реалізує повний цикл управління обліковими записами користувачів з підтримкою різних ролей та прав доступу. Сервіс побудований на основі мікросервісної архітектури з чітким розділенням відповідальностей між компонентами (див. рис. 3.5).

Архітектура модуля побудована за принципами Domain-Driven Design (DDD) та розділення відповідальностей. Модуль розділений на шість основних каталогів, кожен з яких відповідає за конкретний аспект

функціоналу аутентифікації. Така структура забезпечує чистоту коду, легкість тестування та можливість незалежного розвитку окремих компонентів.

Каталог controllers/ містить API контролери, які обробляють HTTP запити від клієнтів. AuthController відповідає за основні операції аутентифікації: реєстрацію, вхід у систему, оновлення токенів, вихід, верифікацію email та скидання паролів. UserController управляє профілями користувачів: отримання інформації, оновлення даних, зміна пароля та управління сесіями. Контролери використовують декоратори для маршрутизації, валідації вхідних даних та документування API через Swagger.



Рисунок 3.5 – Архітектура сервісу аутентифікації

Каталог services/ включає бізнес-логіку модуля. AuthService є основним сервісом, який реалізує ключові операції: реєстрацію користувачів з валідацією паролів та відправкою email підтвердження, автентифікацію через порівняння паролів, генерацію JWT токенів, верифікацію email, управління скиданням паролів та оновлення токенів. UserService забезпечує операції з

профілями користувачів: оновлення інформації, зміну пароля, управління активністю акаунтів. PasswordService спеціалізується на роботі з паролями: хешування з використанням bcrypt, порівняння, валідація сили паролів та генерація випадкових паролів.

Каталог repositories/ реалізує шаблон Repository для абстракції доступу до даних. UserRepository забезпечує CRUD операції з сутністю користувача, включаючи спеціалізовані методи пошуку за email, телефоном, статусом активності та роллю. TokenRepository керує refresh токенами: зберігає, знаходить за токеном або користувачем, відкликає токени та очищає застарілі. Репозиторії використовують TypeORM Query Builder для оптимізованих запитів до бази даних.

Каталог entities/ містить моделі даних бази. User сутність визначає структуру таблиці користувачів з усіма необхідними полями: унікальний ідентифікатор UUID, email з унікальним обмеженням, хеш пароля, ім’я та прізвище, номер телефону, роль (орендар, орендодавець, адміністратор), статуси верифікації та активності, токени для верифікації та скидання пароля, часові мітки. RefreshToken сутність зберігає інформацію про refresh токени: сам токен, зв’язок з користувачем, термін дії, статус відкликання, User-Agent та IP адресу запиту.

Каталог dtos/ (Data Transfer Objects) визначає типи даних для передачі між шарами додатку. RegisterDto описує дані для реєстрації з валідацією email, пароля, імені, прізвища та телефону. LoginDto містить обмеження для даних входу в систему. TokenDto визначає структуру відповіді з токенами. UpdateProfileDto дозволяє оновлювати профіль користувача. Використання DTO забезпечує типобезпеку та автоматичну валідацію вхідних даних через class-validator.

Каталог guards/ реалізує механізми контролю доступу. AuthGuard перевіряє наявність та валідність JWT токена в запиті, витягує користувача з токена та додає його до об’єкта запиту. RolesGuard забезпечує авторизацію на основі ролей, перевіряючи, чи має користувач необхідні права для доступу до

ендпоінту. JwtGuard інтегрується з Passport.js для аутентифікації через JWT стратегію. Гарди використовуються через декоратори на контролерах та методах.

Каталог strategies/ містить стратегії Passport.js для різних методів аутентифікації. JwtStrategy обробляє JWT токени, витягає корисне навантаження та валідує користувача. LocalStrategy реалізує аутентифікацію за email та паролем. OAuth2Strategy (не показана на діаграмі) забезпечує вход через соціальні мережі. Стратегії інтегруються з сервісами для перевірки користувачів у базі даних.

Модульна конфігурація auth.module.ts організовує всі компоненти в єдину структуру. Модуль імпортує необхідні залежності: TypeOrmModule для роботи з сутностями, JwtModule для роботи з токенами, PassportModule для аутентифікації. Він реєструє контролери, сервіси, стратегії та гарди, а також експортує ключові компоненти для використання в інших модулях системи. Конфігурація JWT включає секретний ключ, час життя токенів та алгоритм підпису.

Взаємодія між компонентами відбувається через чітко визначені інтерфейси. Клієнти (веббраузери, мобільні додатки) взаємодіють з контролерами через REST API. Контролери викликають методи сервісів для виконання бізнес-логіки. Сервіси використовують репозиторії для доступу до даних та інші сервіси для допоміжних операцій (наприклад, відправки email). Гарди та стратегії інтегруються з пайплайном обробки запитів для автоматичної аутентифікації та авторизації.

На рисунку 3.6 представлено модульну конфігурацію системи Dependency Injection (DI) для сервісу аутентифікації в системі пошуку та оренди житла. Діаграма демонструє, як організована ін’єкція залежностей у фреймворці NestJS, що забезпечує слабку зв’язність компонентів, легкість тестування та централізоване управління залежностями. Основним компонентом конфігурації є AuthModule клас, який служить кореневим модулем для всіх компонентів аутентифікації. Цей клас реалізує декоратор

@Module() з NestJS, який визначає метадані модуля: імпортовані модулі, контролери, провайдери та експортовані компоненти. Модульна система NestJS забезпечує ієрархічну організацію додатку та автоматичне створення інстансів класів з усіма необхідними залежностями.

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { AuthController } from './controllers/auth.controller';
import { UserController } from './controllers/user.controller';
import { AuthService } from './services/auth.service';
import { UserService } from './services/user.service';
import { PasswordService } from './services/password.service';
import { User } from './entities/user.entity';
import { RefreshToken } from './entities/refresh-token.entity';
import { JwtStrategy } from './strategies/jwt.strategy';
import { LocalStrategy } from './strategies/local.strategy';
import { JwtAuthGuard } from './guards/jwt.guard';
import { RolesGuard } from './guards/roles.guard';

@Module({
  imports: [
    TypeOrmModule.forFeature([User, RefreshToken]),
    PassportModule.register({ defaultStrategy: 'jwt' }),
    JwtModule.registerAsync({
      useFactory: () => ({
        secret: process.env.JWT_ACCESS_SECRET,
        signOptions: {
          expiresIn: process.env.JWT_ACCESS_EXPIRES_IN || '15m',
        },
      }),
    }),
  ],
  controllers: [AuthController, UserController],
  providers: [
    AuthService,
    UserService,
    PasswordService,
    JwtStrategy,
    LocalStrategy,
    JwtAuthGuard,
    RolesGuard,
  ],
  exports: [
    AuthService,
    UserService,
    PasswordService
```

Рисунок 3.6 – Модульна конфігурація для Dependency Injection

Секція imports містить зовнішні модулі, необхідні для роботи модуля аутентифікації. TypeOrmModule.forFeature([User, RefreshToken]) реєструє сутності бази даних у локальному контексті модуля, надаючи доступ до репозиторіїв цих сутностей через Dependency Injection.

PassportModule.register({ defaultStrategy: ‘jwt’ }) налаштовує Passport.js з JWT стратегією за замовчуванням для обробки аутентифікації. JwtModule.registerAsync() асинхронно конфігурує JWT модуль з динамічними параметрами, що завантажуються з змінних середовища.

Секція controllers визначає API контролери модуля. AuthController обробляє всі HTTP запити, пов’язані з аутентифікацією: реєстрація, вхід, оновлення токенів, вихід, верифікація email та скидання паролів. UserController управляє операціями з профілями користувачів: отримання інформації, оновлення даних, зміна пароля. Контролери автоматично отримують необхідні сервіси через конструкторну ін’єкцію залежностей.

Секція providers містить всі сервіси, стратегії та гарди модуля. AuthService є основним сервісом бізнес-логіки аутентифікації. UserService управляє операціями з користувачами. PasswordService спеціалізується на роботі з паролями. JwtStrategy та LocalStrategy реалізують стратегії аутентифікації для Passport.js. JwtAuthGuard та RolesGuard забезпечують захист маршрутів та контроль доступу. NestJS автоматично створює інстанси цих класів та ін’єктує їх залежності.

Конфігурація JwtModule виконується асинхронно через registerAsync(), що дозволяє динамічно завантажувати конфігураційні параметри з змінних середовища або конфігураційних файлів. Фабрика useFactory повертає об’єкт конфігурації з секретним ключем (secret), часом життя токенів (expiresIn) та додатковими опціями. Використання асинхронної конфігурації забезпечує безпеку та гнучкість при розгортанні в різних середовищах.

Інтеграція з TypeORM через TypeOrmModule.forFeature() забезпечує автоматичне створення репозиторіїв для сутностей User та RefreshToken. Ці репозиторії потім ін’єктуються в сервіси через декоратор @InjectRepository(). TypeORM Query Builder надає потужний інтерфейс для створення оптимізованих SQL-запитів з підтримкою транзакцій, кешування та пагінації.

Passport.js інтеграція налаштовується через PassportModule.register(), що реєструє JWT стратегію за замовчуванням. Додаткові стратегії (Local, OAuth2)

можуть бути додані через додаткові конфігурації. Passport.js middleware автоматично обробляє аутентифікацію на основі стратегій та додає користувача до об’єкту запиту для подальшого використання в гардах та контролерах.

3.3 Реалізація модуля управління нерухомістю та геопошуку

Модуль управління нерухомістю та геопошуку є центральним компонентом системи пошуку та оренди житла, оскільки відповідає за зберігання, обробку та пошук інформації про об’єкти нерухомості з використанням геопросторових технологій. Модуль реалізований з використанням TypeORM для роботи з PostgreSQL та PostGIS для геопросторових операцій.

На рисунку 3.7 представлено детальну структуру модуля управління нерухомістю та геопошуку серверної частини системи пошуку та оренди житла. Діаграма демонструє організацію коду та взаємозв’язки між компонентами, що забезпечують повний цикл роботи з об’єктами нерухомості: від створення та редагування до геопросторового пошуку та індексації.

Архітектура модуля побудована на принципах Domain-Driven Design (DDD) з чітким розділенням відповідальностей між компонентами. Модуль організований у вісім логічних каталогів, кожен з яких спеціалізується на конкретній функціональній області. Така структура забезпечує масштабованість, легкість тестування та можливість незалежного розвитку окремих частин системи.

Каталог controllers/ містить API контролери для обробки HTTP запитів. PropertyController відповідає за CRUD операції з об’єктами нерухомості: створення нових оголошень, оновлення інформації, видалення, отримання деталей та управління станом. SearchController спеціалізується на пошукових

операціях: геопросторовий пошук за координатами та радіусом, фільтрація за параметрами, повнотекстовий пошук та автодоповнення. Контролери використовують декоратори для маршрутизації, валідації вхідних даних та інтеграції з системою авторизації.



Рисунок 3.7 – Структура модуля управління

Каталог services/ включає бізнес-логіку модуля. PropertyService є основним сервісом для управління нерухомістю: валідація даних, обробка зображень, управління доступністю, розрахунок цін та статистики. SearchService реалізує гібридний пошук, що комбінує можливості PostGIS для геопошуку та Elasticsearch для повнотекстового пошуку. GeospatialService спеціалізується на геопросторових операціях: обчислення відстаней, пошук у радіусі, геокодування адрес та оптимізація просторових запитів. ImageService управляє завантаженням, обробкою та зберіганням фотографій нерухомості.

Каталог repositories/ реалізує шаблон Repository для абстракції доступу до даних. PropertyRepository забезпечує CRUD операції з сутністю

нерухомості, включаючи спеціалізовані методи для геопросторових запитів через PostGIS функції. AvailabilityRepository управляє календарем доступності: перевірка вільних дат, блокування під час бронювань, обчислення цін за період. Репозиторії використовують TypeORM Query Builder для створення оптимізованих SQL-запитів з підтримкою транзакцій та пагінації.

Каталог entities/ містить моделі даних бази. Property сутність визначає структуру об'єкта нерухомості з усіма атрибутами: географічні координати (тип Point з SRID 4326), тип нерухомості, ціни, кількість кімнат, зручності, правила, адреса у JSONB форматі. PropertyImage зберігає інформацію про фотографії: URL, мініатюри, порядок відображення. PropertyAvailability управляє календарем доступності з динамічними цінами та обмеженнями. Amenity визначає стандартизований список зручностей для фільтрації.

Каталог dtos/ (Data Transfer Objects) визначає типи даних для передачі між шарами. CreatePropertyDto описує дані для створення нового оголошення з валідацією координат, цін та обов'язкових полів. UpdatePropertyDto дозволяє часткове оновлення інформації. SearchCriteriaDto визначає параметри пошуку: географічні обмеження, ціновий діапазон, фільтри за зручностями. GeoSearchDto спеціалізується на геопросторових критеріях пошуку. DTO забезпечують типобезпеку та автоматичну валідацію.

Каталог pipes/ містить кастомні пайпи для трансформації та валідації даних. CoordinateValidationPipe перевіряє коректність географічних координат (широта в межах -90 до 90, довгота -180 до 180) та конвертує їх у формат Point для зберігання в базі. PriceRangePipe нормалізує цінові діапазони для пошуку. AmenityFilterPipe перетворює рядок зручностей у масив для фільтрації.

Каталог interceptors/ реалізує перехоплювачі для обробки відповідей. PropertyResponseInterceptor трансформує відповіді API: додає метадані для пагінації, обчислює загальну кількість результатів, форматує географічні координати для клієнта. SearchResultInterceptor оптимізує результати пошуку: об'єднує дані з різних джерел, сортує за релевантністю, додає відстань до точки пошуку.

Модульна конфігурація properties.module.ts організовує всі компоненти в єдину структуру. Модуль імпортує необхідні залежності: TypeOrmModule для роботи з сутностями, ElasticsearchModule для пошуку, MulterModule для завантаження файлів. Він реєструє контролери, сервіси, пайпи та інтерцептори, а також експортує ключові компоненти для використання в інших модулях системи.

На рисунку 3.8 представлено детальну структуру сутності нерухомості (Property Entity) – основної моделі даних для зберігання інформації про об’єкти нерухомості в системі пошуку та оренди житла. Діаграма демонструє повну архітектуру сутності з усіма полями, типами даних, індексами та зв’язками, що забезпечують ефективну роботу з геопросторовими даними та складними бізнес-процесами оренди.

```
import {
  Entity,
  Column,
  PrimaryGeneratedColumn,
  CreateDateColumn,
  UpdateDateColumn,
  ManyToOne,
  OneToMany,
  Index,
  JoinColumn
} from 'typeorm';
import { User } from '../../auth/entities/user.entity';
import { PropertyImage } from './property-image.entity';
import { PropertyAvailability } from './property-availability.entity';
import { Point } from 'geojson';

@Entity('properties')
@Index('IDX_PROPERTY_LOCATION', ['location'], { spatial: true })
@Index('IDX_PROPERTY_OWNER', ['ownerId'])
@Index('IDX_PROPERTY_TYPE', ['propertyType'])
@Index('IDX_PROPERTY_PRICE', ['pricePerNight'])
export class Property {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ length: 255 })
  title: string;

  @Column('text')
  description: string;

  @Column({
    name: 'property_type',
    type: 'enum',
    enum: ['apartment', 'house', 'villa', 'cottage', 'room'],
    default: 'apartment'
  })
  propertyType: PropertyType;

  @Column({
    name: 'room_type',
    type: 'enum',
    enum: ['entire_home', 'private_room', 'shared_room'],
    default: 'entire_home'
```

Рисунок 3.8 – Структура сутності нерухомості (Property Entity)

Сутність реалізована на основі TypeORM з використанням PostgreSQL та розширення PostGIS для роботи з географічними даними. Клас Property містить декоратори @Entity(‘properties’) для визначення імені таблиці в базі даних та @Index для створення оптимізованих індексів для швидкого пошуку. Використання TypeORM забезпечує об’єктно-орієнтовану роботу з даними та автоматичну генерацію міграцій.

Первинний ключ id типу UUID генерується автоматично через @PrimaryGeneratedColumn(‘uuid’), що забезпечує глобальну унікальність ідентифікаторів та безпеку через непередбачуваність значень. UUID формат дозволяє легко шардувати базу даних у майбутньому та уникнути конфліктів при реплікації. Для користувача цей ідентифікатор представлений у вигляді рядка формату «550e8400-e29b-41d4-a716-446655440000».

Основні атрибути нерухомості включають title (назва оголошення) типу VARCHAR(255) з обмеженням довжини для оптимізації зберігання та description типу TEXT для детального опису без обмежень довжини. Поле propertyType використовує ENUM тип з п’ятьма допустимими значеннями: apartment (квартира), house (будинок), villa (вілла), cottage (котедж), room (кімната). roomType визначає тип розміщення: entire_home (весь будинок), private_room (приватна кімната), shared_room (спільна кімната).

Фінансові параметри представлені полями pricePerNight та pricePerMonth типу DECIMAL(10,2) з точністю до двох знаків після коми для коректного обчислення вартості. Використання DECIMAL замість FLOAT гарантує точність фінансових розрахунків без помилок округлення. Поле pricePerMonth є опціональним і може розраховуватись динамічно на основі щоденної ціни при тривалому орендуванні.

Параметри розміщення включають bedrooms (кількість спалень), bathrooms (кількість ванних кімнат), maxGuests (максимальна кількість гостей) та areaSqm (площа в квадратних метрах). Ці поля використовуються для фільтрації та сортування результатів пошуку. Поле areaSqm є опціональним, оскільки не всі орендодавці знають точну площу нерухомості.

Геопросторові дані зберігаються у полі location типу GEOGRAPHY(Point, 4326) з використанням PostGIS розширення. Тип Point представляє географічну точку з координатами (довгота, широта). SRID 4326 вказує систему координат WGS 84, що є стандартом для GPS та більшості картографічних сервісів. Для швидкого геопросторового пошуку створюється просторовий індекс IDX_PROPERTY_LOCATION типу GiST.

Структуровані дані адреси зберігаються у полі address типу JSONB, що дозволяє гнучко зберігати різні компоненти адреси без необхідності створення окремих стовпців. Об’єкт адреси включає: country (країна), city (місто), street (вулиця), zipCode (поштовий індекс), district (район, опціонально) та formattedAddress (повна адреса для відображення). JSONB забезпечує індексацію та ефективний пошук за вкладеними полями.

Списки зручностей та правил представлені полями amenities (масив рядків) та houseRules (об’єкт з правилами) типу JSONB. Amenities містить стандартизований список зручностей: “wifi”, “parking”, “pool”, “air_conditioning”, “kitchen” тощо. houseRules включає логічні прапорці для дозволу паління, тварин, вечірок, а також час заїзду/виїзду та додаткові правила. Використання JSONB дозволяє легко розширювати список атрибутів без змін схеми бази.

Зв’язки з іншими сутностями реалізовані через @ManyToOne з користувачем-власником та @OneToMany з зображеннями та календарем доступності. ownerId зберігає зовнішній ключ до таблиці користувачів з каскадним видаленням (onDelete: ‘CASCADE’). images представляє колекцію фотографій нерухомості з автоматичним каскадним оновленням. availabilities зв’язує з календарем доступності для динамічного управління датами.

3.4 Тестування та оптимізація продуктивності системи

Тестування та оптимізація продуктивності є критично важливим етапом розробки серверної частини системи пошуку та оренди житла, оскільки

система повинна ефективно обробляти тисячі одночасних запитів, швидко виконувати геопросторові пошуки та забезпечувати стабільну роботу під час пікових навантажень. Юніт-тестування (Unit Testing) є фундаментальним рівнем тестування, що спрямований на перевірку окремих компонентів системи у ізоляції від зовнішніх залежностей. У контексті серверної частини системи пошуку та оренди житла, юніт-тести охоплюють тестування окремих сервісів, репозиторіїв, контролерів та утиліт.

Наприклад, тестування сервісу аутентифікації перевіряє коректність хешування паролів, генерації JWT токенів та валідації облікових даних без реального звернення до бази даних чи зовнішніх сервісів. Використовуються моки та стаби для імітації поведінки залежностей, що дозволяє тестувати бізнес-логіку в контрольованому середовищі. Юніт-тести характеризуються високою швидкістю виконання, великою кількістю тестових сценаріїв та можливістю раннього виявлення логічних помилок на етапі розробки.

На рисунку 3.9 представлено приклад юніт-тестування для компоненту SearchService у системі пошуку та оренди житла. Діаграма демонструє архітектуру тесту, взаємодію з мокованими залежностями та ключові тестові сценарії для перевірки гібридної системи пошуку.

Тестовий клас SearchService створюється за допомогою Test.createTestingModule() з NestJS Testing Module API. Цей підхід дозволяє ізолювати тестований сервіс від реальних залежностей, замінюючи їх контрольованими моками. Модуль налаштовується з чотирма провайдерами: SearchService (тестований компонент), ElasticsearchService, GeospatialService та RedisService. Кожен зовнішній сервіс замінюється mock-об’єктом з визначеними методами, що повертають контрольовані значення.

Мокування залежностей виконується через useValue з Jest функціями jest.fn(). ElasticsearchService містить мокові методи searchProperties та indexProperty. GeospatialService включає findPropertiesWithinRadius та calculateDistance. RedisService мокує операції кешування: get, set та invalidatePattern. Кожен мок ініціалізується пустою функцією, яка може бути

налаштована для конкретних тестових сценаріїв через mockResolvedValue, mockReturnValue або mockImplementation.



```
import { Test, TestingModule } from '@nestjs/testing';
import { SearchService } from '../../../src/modules/properties/services/search.service';
import { ElasticsearchService } from '../../../src/shared/search/elasticsearch.service';
import { GeospatialService } from '../../../src/modules/properties/services/geospatial.service'
import { RedisService } from '../../../src/shared/cache/redis.service';
import { Logger } from '@nestjs/common';

describe('SearchService', () => {
  let searchService: SearchService;
  let elasticsearchService: ElasticsearchService;
  let geospatialService: GeospatialService;
  let redisService: RedisService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [
        SearchService,
        {
          provide: ElasticsearchService,
          useValue: {
            searchProperties: jest.fn(),
            indexProperty: jest.fn(),
          },
        },
        {
          provide: GeospatialService,
          useValue: {
            findPropertiesWithinRadius: jest.fn(),
            calculateDistance: jest.fn(),
          },
        },
        {
          provide: RedisService,
          useValue: {
            get: jest.fn(),
            set: jest.fn(),
            invalidatePattern: jest.fn(),
          },
        },
        {
          provide: Logger,
          useValue: {
```

Рисунок 3.9 – Юніт-тест SearchService

Тестований сервіс SearchService отримує залежності через Dependency Injection. Після компіляції тестового модуля, сервіси отримуються через module.get() метод. Це забезпечує, що SearchService отримує саме мокові версії залежностей, а не реальні реалізації. Такий підхід дозволяє тестувати бізнес-логіку сервісу в повній ізоляції від зовнішніх систем.

Перший тестовий сценарій should return cached results if available перевіряє оптимізацію продуктивності через кешування. Тест імітує ситуацію,

коли результати пошуку вже знаходяться в Redis кеші. redisService.get налаштовується на повернення закешованих даних через mockResolvedValue(cachedResult). Параметри пошуку включають координати Києва (50.45, 30.52), радіус 5 км, текстовий запит “apartment near center” та параметри пагінації. Ключ кешу генерується на основі всіх параметрів пошуку для унікальності.

Інтеграційне тестування (Integration Testing) зосереджене на перевірці взаємодії між різними компонентами системи. У системі пошуку та оренди житла це включає тестування взаємодії між сервісами аутентифікації та управління нерухомістю, інтеграції бази даних PostgreSQL з додатком, комунікації між Elasticsearch та пошуковим сервісом, а також взаємодії з зовнішніми сервісами (Stripe для оплати, SendGrid для email). Інтеграційні тести використовують тестові бази даних, мокові зовнішні сервіси та Docker контейнери для створення ізольованого середовища, що наближене до продуктивного. Ці тести перевіряють коректність трансформації даних між шарами, правильність SQL-запитів, обробку помилок при збоях зовнішніх систем та відповідність API контрактам.

На рисунку 3.10 представлено приклад інтеграційного тестування API для модуля пошуку в системі пошуку та оренди житла. Діаграма демонструє повний цикл end-to-end тестування API ендпоінтів з використанням реальних HTTP запитів, аутентифікації, тестової бази даних та перевірки бізнес-логіки в інтегрованому середовищі.

Тестовий файл search.e2e-spec.ts організований за допомогою Jest та Supertest для виконання HTTP запитів до запущеного екземпляра додатку NestJS. На відміну від юніт-тестів, інтеграційні тести працюють з реальною або тестовою інфраструктурою, перевіряючи взаємодію між різними компонентами системи. Тести виконуються у порядку: ініціалізація тестового середовища, виконання тестових сценаріїв, очищення ресурсів.

Передусім тестуванням виконується ініціалізація тестового середовища через beforeAll хук. createTestDatabase() створює ізольовану тестову базу

даних PostgreSQL з розширенням PostGIS для геопросторових операцій. Тестовий модуль NestJS компілюється через Test.createTestingModule() з імпортом повного AppModule, що забезпечує наявність всіх залежностей та middleware. Додаток ініціалізується через app.init(), запускаючи всі сервіси та підключення до бази даних.

```
import { Test, TestingModule } from '@nestjs/testing';
import { INestApplication } from '@nestjs/common';
import * as request from 'supertest';
import { AppModule } from '../../src/app.module';
import { createTestDatabase, dropTestDatabase } from '../utils/test-database';

describe('Search API (e2e)', () => {
  let app: INestApplication;
  let accessToken: string;

  beforeAll(async () => {
    await createTestDatabase();

    const moduleFixture: TestingModule = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleFixture.createNestApplication();
    await app.init();

    // Аутентифікація для отримання токена
    const authResponse = await request(app.getHttpServer())
      .post('/auth/login')
      .send({
        email: 'test@example.com',
        password: 'TestPass123!',
      });

    accessToken = authResponse.body.data.accessToken;
  });

  afterAll(async () => {
    await dropTestDatabase();
    await app.close();
  });
```

Рисунок 3.10 – Інтеграційне тестування API

Отримання токена аутентифікації є критичним кроком для тестування захищених ендпоінтів. Виконується POST запит до /auth/login з тестовими обліковими даними (email: test@example.com, password: TestPass123!). Успішна відповідь містить JWT access token, який зберігається у змінній accessToken для подальшого використання в заголовках Authorization. Це імітує реальний потік користувача, який спочатку входить у систему перед виконанням пошуку.

Перша група тестів POST /api/search/properties перевіряє основний ендпоінт пошуку нерухомості. Тестовий запит містить комплексні критерії пошуку: координати центру Києва (50.4501, 30.5234), радіус 5 км, тип нерухомості “apartment”, ціновий діапазон 50-200 USD, зручності “wifi” та “parking”, дати бронювання 1-7 грудня 2024, параметри пагінації. Запит відправляється з заголовком Authorization, що містить отриманий JWT токен.

Перевірки для успішного пошуку включають: статус відповіді 200, наявність поля success: true, структуру даних відповіді з полями properties, total, page, limit. Додатково перевіряється, що properties є масивом об’єктів, що дозволяє переконатись у коректному форматі відповіді та наявності даних. Це забезпечує, що API правильно обробляє складні критерії пошуку та повертає структуровану відповідь.

Тест валідації параметрів пошуку перевіряє обробку некоректних вхідних даних. Відправляється запит з невалідними координатами (широта 100 замість допустимих -90..90, довгота 200 замість -180..180) та негативним радіусом (-5 км). Очікується статус відповіді 400 (Bad Request), що демонструє правильну роботу валідаційних пайпів та DTO. Це критично важливо для безпеки та стабільності системи.

Навантажувальне тестування (Load Testing) спрямоване на оцінку продуктивності системи під різним навантаженням. Для системи пошуку та оренди житла це включає тестування здатності обробляти одночасні запити пошуку нерухомості, обробляти бронювання під час пікового навантаження та підтримувати стабільну роботу при збільшенні кількості користувачів. Навантажувальні тести виконуються за допомогою інструментів на кшталт k6, JMeter або Locust, які імітують тисячі одночасних користувачів. Ключові метрики включають час відповіді API, кількість запитів за секунду, використання ресурсів (CPU, пам’ять), виявлення вузьких місць та визначення максимальної пропускної здатності системи.

На рисунку 3.11 представлено навантажувальне тестування системи пошуку та оренди житла за допомогою інструменту k6. Діаграма демонструє

архітектуру тестового сценарію, стратегію навантаження, метрики моніторингу та перевірки продуктивності для оцінки масштабованості та стабільності системи під різними рівнями навантаження.

Тестовий сценарій реалізований на мові JavaScript з використанням k6 – сучасного інструменту для навантажувального тестування з відкритим вихідним кодом. k6 дозволяє створювати складні тестові сценарії, імітуючи поведінку реальних користувачів, з підтримкою HTTP/2, WebSocket та інших протоколів. Сценарій організований як ES6 модуль з експортованими функціями та конфігураційними об'єктами.

```
import { check, sleep } from 'k6';
import http from 'k6/http';

export const options = {
  stages: [
    { duration: '30s', target: 50 },  // Поступове збільшення до 50 користувачів
    { duration: '1m', target: 100 },  // Збільшення до 100 користувачів
    { duration: '2m', target: 200 },  // Пікове навантаження 200 користувачів
    { duration: '30s', target: 50 },  // Поступове зниження
    { duration: '30s', target: 0 },   // Завершення тесту
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'], // 95% запитів мають виконуватись менше ніж за 500мс
    http_req_failed: ['rate<0.01'],   // Менше 1% неуспішних запитів
  },
};

const BASE_URL = __ENV.API_URL || 'http://localhost:3000';
const ACCESS_TOKEN = __ENV.ACCESS_TOKEN;

export default function () {
  // Тест пошуку нерухомості
  const searchPayload = JSON.stringify({
    location: {
      lat: 50.4501 + (Math.random() - 0.5) * 0.1,
      lng: 30.5234 + (Math.random() - 0.5) * 0.1,
    },
    radius: 2 + Math.floor(Math.random() * 8),
    propertyType: ['apartment', 'house', 'room'][Math.floor(Math.random() * 3)],
    priceRange: {
      min: Math.floor(Math.random() * 100),
      max: 100 + Math.floor(Math.random() * 400),
    },
    page: 1,
    limit: 20,
  });

  const searchParams = {
    headers: {
      'Content-Type': 'application/json',
      Authorization: `Bearer ${ACCESS_TOKEN}`,
    },
  };

  const searchResponse = http.post(
```

Рисунок 3.11 – Навантажувальні тести

Конфігурація тесту визначається об’єктом options, який містить стратегію навантаження та порогові значення метрик. Стратегія stages визначає поетапне збільшення навантаження для імітації реального сценарію використання системи. Перший етап тривалістю 30 секунд збільшує навантаження до 50 одночасних віртуальних користувачів, що імітує поступове зростання активності в системі.

Другий та третій етапи збільшують навантаження до 100 та 200 користувачів відповідно. Етап з 200 користувачами триває 2 хвилини, що дозволяє оцінити стабільність системи під тривалим піковим навантаженням. Останні два етапи поступово знижують навантаження до нуля, імітуючи спад активності та дозволяючи системі відновитись. Така стратегія дозволяє виявити проблеми з пам’яттю, з’єднаннями та іншими ресурсами, які можуть накопичуватись з часом.

Порогові значення метрик визначаються в thresholds. http_req_duration: [‘p(95)<500’] встановлює вимогу, щоб 95% HTTP запитів виконувались менше ніж за 500 мілісекунд. http_req_failed: [‘rate<0.01’] вимагає, щоб частота неуспішних запитів була менше 1%. Ці порогові значення базуються на SLA (Service Level Agreement) системи та вимогах до користувацького досвіду. При порушенні порогових значень тест вважається невдалим, що дозволяє автоматично виявляти деградацію продуктивності.

На рисунку 3.12 зображена головна сторінка вебінтерфейсу системи пошуку та оренди житла.

Головна сторінка інтерфейсу користувача розроблена з урахуванням принципів юзабіліті та містить усі необхідні елементи для комфортного пошуку та перегляду варіантів житла. Вона складається з кількох логічних блоків, кожен з яких виконує певну функціональну роль у системі.

У верхній частині сторінки розташована панель навігації, яка містить логотип та назву системи “RentHome”, основні навігаційні посилання (Пошук житла, Для орендодавців, Допомога, Про нас), а також кнопки авторизації для входу та реєстрації користувачів. Ця панель залишається доступною на всіх сторінках системи, що забезпечує зручну навігацію.

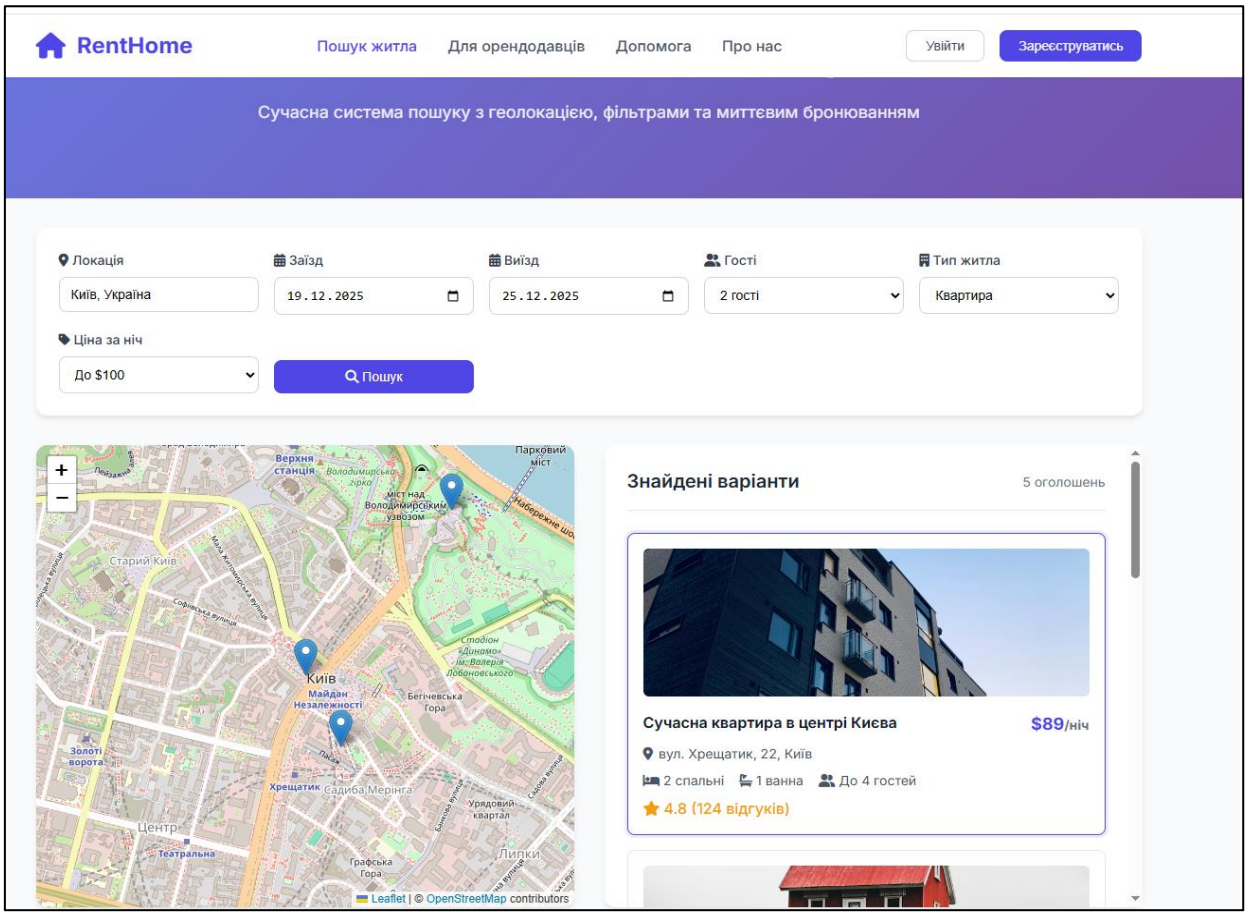


Рисунок 3.12 – Головна сторінка

Під панеллю навігації знаходиться герой-секція (hero section), яка містить основний заголовок «Знайдіть ідеальне житло для оренди» та короткий опис можливостей системи. Цей блок виконує презентаційну функцію та мотивує користувача до використання сервісу.

Основну площу сторінки займає інтерактивний блок, який поєднує в собі мапу з маркерами доступних об’єктів та список знайдених варіантів оренди. Мапа реалізована за допомогою бібліотеки Leaflet із використанням OpenStreetMap, що дозволяє візуалізувати географічне розташування об’єктів нерухомості. Кожен маркер на мапі відповідає певному об’єкту оренди та містить попап з основною інформацією про нього.

Список об’єктів праворуч від мапи відображає картки нерухомості з такою інформацією: фотографія об’єкта, назва, ціна за ніч, адреса, кількість спалень та ванних кімнат, максимальна кількість гостей, рейтинг та кількість відгуків. Картки є інтерактивними – при кліку на картку або маркер на мапі

відбувається синхронізація виділення, а також відкривається модальне вікно з детальною інформацією про об’єкт.

Модальне вікно з деталями об’єкта містить розширену інформацію: галерею фотографій, детальний опис, список зручностей, а також форму бронювання з можливістю вибору дат, кількості гостей та автоматичним розрахунком загальної вартості оренди. Вікно також відображає деталізований розрахунок вартості з урахуванням сервісних зборів.

У нижній частині головної сторінки представлена секція «Архітектура системи», яка описує технологічний стек серверної частини. Ця інформація включає три основні групи компонентів: серверна частина (Node.js, Express.js, PostgreSQL з PostGIS, Elasticsearch, Redis, JWT), модуль геопошуку (PostGIS для роботи з геоданими, пошук у радіусі, розрахунок відстаней) та модуль безпеки та масштабування (Docker, мікросервісна архітектура, захист від SQL-ін’єкцій, rate limiting, HTTPS). На рисунку 3.13 зображено модальне вікно перегляду детальної інформації про об’єкт оренди та бронювання.

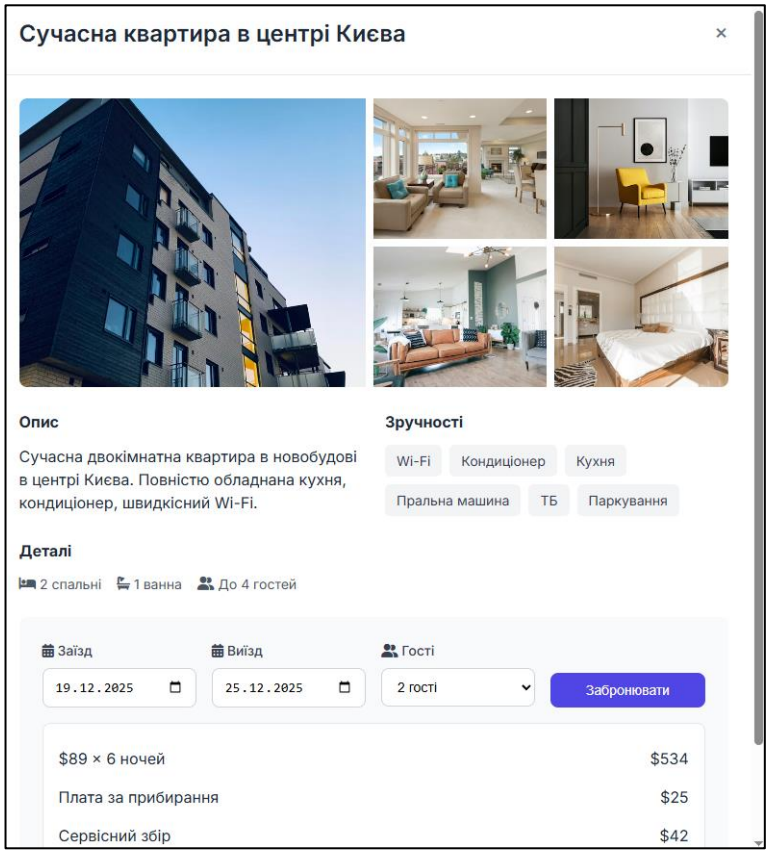


Рисунок 3.13 – Модальне вікно перегляду детальної інформації

Модальне вікно відкривається при виборі конкретного об’єкта нерухомості зі списку результатів пошуку або при кліку на маркер на мапі. Воно містить повну інформацію про обраний варіант оренди та функціональність для оформлення бронювання.

У верхній частині вікна розташований заголовок з назвою об’єкта «Сучасна квартира в центрі Києва» та кнопка закриття вікна. Під заголовком знаходиться галерея зображень об’єкта, яка включає одне головне зображення великого розміру та чотири додаткових фотографії меншого формату. Галерея реалізована за принципом сітки, що дозволяє ефективно використовувати простір та надавати користувачу візуальне уявлення про об’єкт.

Основний контент вікна організований у три інформаційні блоки. Перший блок містить детальний опис об’єкта, де наведено повну характеристику квартири: її розташування в центрі Києва, тип будівлі (новобудова), кількість кімнат, а також перелік основних особливостей (повністю обладнана кухня, кондиціонер, швидкісний Wi-Fi).

Другий блок містить деталі об’єкта у структурованому вигляді: кількість спалень (2), кількість ванних кімнат (1) та максимальну кількість гостей (до 4 осіб). Ця інформація представлена з використанням інтуїтивно зрозумілих іконок, що покращує сприйняття даних.

Третій блок містить перелік зручностей, доступних у об’єкті оренди. У прикладі наведено шість основних зручностей: Wi-Fi, кондиціонер, кухня, пральна машина, телевізор та паркування. Кожна зручність представлена у вигляді окремого бейджа з закругленими кутами, що візуально групує елементи та покращує читабельність списку.

Нижня частина модального вікна містить функціональний блок для бронювання, розділений горизонтальною лінією. У лівій частині розташована форма для введення параметрів бронювання, яка включає поля для вибору дати заїзду (19.12.2025) та дати виїзду (25.12.2025), а також випадаючий список для вибору кількості гостей (2 гості). Під формою розташована кнопка «Забронювати», яка ініціює процес оформлення замовлення.

У правій частині знаходиться блок з розрахунком вартості оренди, який автоматично оновлюється при зміні дат перебування. Розрахунок включає: базову вартість (89$ × 6 ночей = 534$), плату за прибирання (25$) та сервісний збір (42$). У нижній частиці блоку виділено загальну суму до сплати, яка в даному прикладі становить 601$. Такий розподіл вартості на складові елементи забезпечує прозорість розрахунків та довіру користувача до системи.

3.5 Висновки до розділу 3

У третьому розділі магістерської роботи було реалізовано серверну частину системи пошуку та оренди житла засобами Node.js та PostgreSQL. Було налаштовано повноцінне середовище розробки з використанням Docker та Docker Compose, що дозволило створити ізольовану інфраструктуру для розгортання бази даних PostgreSQL з PostGIS та серверних компонентів.

Було розроблено та впроваджено сервіс аутентифікації та авторизації на основі JWT токенів з підтримкою ролей користувачів (орендарі, орендодавці, адміністратори). Сервіс реалізує повний цикл управління користувачами: реєстрацію, вхід, оновлення профілю та контроль доступу до API ендпоінтів.

Ключовим елементом розробки став модуль управління нерухомістю та геопошуку, який інтегрував можливості PostgreSQL/PostGIS для зберігання та обробки геопросторових даних. Було реалізовано функціонал пошуку об’єктів нерухомості в заданому радіусі, фільтрації за різними параметрами та сортування за відстанню. Модуль також забезпечує CRUD операції для об’єктів нерухомості з прив’язкою до географічних координат.

Для забезпечення якості та надійності системи було проведено комплексне тестування, що включало модульне тестування окремих компонентів, інтеграційне тестування взаємодії між сервісами та системне тестування продуктивності. Було оптимізовано запити до бази даних через створення індексів для геопросторових даних та кешування часто використовуваних результатів.

В результаті розробки було створено повнофункціональну серверну частину системи, що підтверджує практичну реалізацію теоретичних концепцій, розглянутих у попередніх розділах. Система демонструє ефективне поєднання сучасних вебтехнологій для вирішення задач пошуку та оренди житла з підтримкою географічного пошуку.

ВИСНОВКИ

На етапі дослідження предметної області було проаналізовано специфіку сучасних систем пошуку житла та виявлено потребу у створенні масштабованого рішення, здатного ефективно обробляти геопросторові дані, забезпечувати швидкий пошук у великих обсягах інформації та надавати надійний API для клієнтських застосунків. Результати аналізу визначили архітектурні вимоги до майбутньої системи та обґрунтували вибір технологічного стеку.

Проєктування системи включало розробку модульної мікросервісної архітектури, проєктування схеми бази даних з урахуванням геопросторових потреб за допомогою PostGIS, визначення ключових компонентів: сервісу аутентифікації, модуля управління нерухомістю, геопошуку та системи бронювань. Застосування Node.js дозволило забезпечити високу продуктивність та асинхронну обробку запитів, а використання PostgreSQL з розширенням PostGIS забезпечило надійне зберігання та ефективну обробку географічних даних.

У процесі розробки серверної частини значна увага приділялася принципам безпеки, масштабованості та продуктивності. Було реалізовано сервіс аутентифікації на основі JWT токенів з підтримкою ролей користувачів, що забезпечило захищений доступ до API. Модуль геопошуку було побудовано з використанням можливостей PostGIS для пошуку об’єктів у заданому радіусі, розрахунку відстаней та оптимізованої фільтрації. Система управління нерухомістю забезпечила повний цикл CRUD операцій з прив’язкою до геолокації.

Етап програмної реалізації продемонстрував ефективність обраних технологій та архітектурних рішень. Серверна частина була реалізована з урахуванням принципів чистої архітектури, що сприяє легкій підтримці та розширенню системи. Було досягнуто високої продуктивності геопошуку

через створення спеціальних індексів та кешування результатів. Інтеграція всіх модулів забезпечила узгоджену роботу системи та стабільну взаємодію з клієнтськими застосунками.

Для забезпечення якості системи було проведено комплексне тестування, що включало модульне тестування окремих компонентів, інтеграційне тестування взаємодії сервісів та тестування продуктивності геопошуку. Оптимізація запитів до бази даних та впровадження кешування дозволили досягти високої швидкодії навіть при роботі з великими обсягами даних.

У результаті проведеної роботи досягнуто поставлену мету – створено серверну частину системи пошуку та оренди житла, яка характеризується високою продуктивністю, масштабованістю та надійністю. Система забезпечує ефективний геопошук, безпечну обробку транзакцій бронювання та може обслуговувати тисячі одночасних користувачів.

Розроблена система має потенціал для подальшого вдосконалення та розширення. Перспективними напрямами розвитку можуть бути інтеграція з системами онлайн-платежів, впровадження рекомендаційних алгоритмів на основі машинного навчання, розширення функціоналу аналітики для орендодавців, підтримка додаткових геопросторових операцій та масштабування за допомогою кластеризації бази даних.

Таким чином, кваліфікаційна робота реалізувала повноцінну серверну частину сучасної системи пошуку та оренди житла, яка відповідає вимогам продуктивності, безпеки та масштабованості. Робота демонструє ефективність поєднання Node.js та PostgreSQL з PostGIS для вирішення складних задач обробки геоданих та може слугувати основою для подальших досліджень і практичних впроваджень у сфері геоінформаційних систем та онлайн-бронювань.

ПЕРЕЛІК ПОСИЛАНЬ

1. Node.js Official Documentation. URL: https://nodejs.org/en/docs/ (дата звернення: 12.09.2025).

2. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 15.09.2025).

3. PostGIS Documentation. URL: https://postgis.net/documentation/ (дата звернення: 18.09.2025).

4. Express.js – Node.js web application framework. URL: https://expressjs.com/ (дата звернення: 20.09.2025).

5. Herron D. Node.js Web Development : 3rd ed. Packt Publishing, 2016. 809 p. URL: https://www.everand.com/book/365187139/Node-js-Web-Development-Third-Edition (дата звернення: 25.10.2025).

6. Biehl M. RESTful API Design – Best Practices in API Design with REST: API-University Series, #3. API-University Press, 2016. 475 p. URL: https://www.everand.com/book/407678610/RESTful-API-Design-Best-Practices-in-API-Design-with-REST-API-University-Series-3 (дата звернення: 01.11.2025).

7. Obe R. O., Hsu L. S. PostgreSQL: Up and Running – A Practical Guide to the Advanced Open Source Database. O'Reilly Media, 2017. 314 p.

8. Kleppmann M. Designing Data-Intensive Applications. O'Reilly Media, 2017. 345 p.

9. Poulter A. J., Ossont S. J., Cox S. J. Using the MEAN Stack to implement a RESTful service for an Internet of Things Application. *IEEE World Forum on Internet of Things (WF-IoT).* Milan, Italy, 2015. P. 280-285. URL: https://doi.org/10.1109/WF-IoT.2015.7389066 (дата звернення: 05.11.2025).

10. Building Scalable APIs with Node.js and PostgreSQL. *Smashing Magazine*. URL: https://www.smashingmagazine.com/2023/05/building-scalable-apis-nodejs-postgresql/ (дата звернення: 05.11.2025).

11. JWT Authentication Tutorial for Node.js. *Auth0 Developer Resources*. URL: https://developer.auth0.com/resources/guides/nodejs/authentication/jwt (дата звернення: 10.11.2025).

12. Geospatial Queries in PostgreSQL/PostGIS. *GIS Stack Exchange*. URL: https://gis.stackexchange.com/questions/31628/fastest-way-to-find-points-within-a-distance-from-a-lat-lon (дата звернення: 15.11.2025).