НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	Завідувач кафедри
інформаційних технологій	комп'ютерних наук
(назва факультету (ННІ))	(назва кафедри)
_____ Ігор БОЛБОТ	_____ Белла ГОЛУБ
(підпис) (ПІБ)	(підпис) (ПІБ)
"___" _____ 20_ р.	"___" _____ 20_ р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему Програмне забезпечення платформи для управління нерухомістю та взаємодії між орендарями і орендодавцями

Спеціальність 121 «Інженерія програмного забезпечення»
(код і найменування)

Освітня програма Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми освітня-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми
к.ф.-м..н., доцент _____ Кириченко В.В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи
к.т.н., доцент _____ Вайганг Г.О.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Мотлюк О.П.
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) __інформаційних технологій_____

ЗАТВЕРДЖУЮ
Завідувач кафедри__комп'ютерних наук_
___доцент, к.т.н._____ _____ _Белла ГОЛУБ_
(науковий ступінь, вчене звання) (підпис) (ПІБ)
"_01_" ___листопада_____2024 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Мотлюк Олександр Петрович
(прізвище, ім'я, по батькові)

Спеціальність 121 «Інженерія програмного забезпечення»
(код і назва)

Освітня програма ____Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми_____освітньо-професійна_____

Тема магістерської кваліфікаційної роботи Програмне забезпечення платформи для управління нерухомістю і взаємодії між орендарями і орендодавцями
затверджена наказом ректора НУБіП України від "_1_" _листопада__2024р. №1963 «С»

Термін подання завершеної роботи на кафедру __14 листопада 2025_____
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи: чинна нормативно-правова база у сфері управління нерухомістю та цифрових сервісів, наявні датасети орендарів і орендодавців, статистична інформація про ринок оренди, алгоритми й базові методи проєктування, моделювання та розроблення програмного забезпечення, що забезпечують реалізацію функціональних і аналітичних компонентів дослідження.

Перелік питань, що підлягають дослідженню:
1. Аналіз предметної області та постановка задачі
2. Проєктування програмного забезпечення
3. Реалізація та тестування системи
4. Аналіз результатів, економічне обґрунтування та перспективи розвитку системи.

Перелік графічного матеріалу (за потреби) презентація, постер_____

Дата видачі завдання "__1__"__листопада_____2024_ р.

Керівник магістерської кваліфікаційної роботи _____ ___Вайганг Г. О.____
(підпис) (прізвище та ініціали)

Завдання прийняв до виконання _____ _____Мотлюк О. П._____
(підпис) (прізвище та ініціали студента)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AES - Advanced Encryption Standard – стандартизований алгоритм симетричного шифрування

API – Application Programming Interface – програмний інтерфейс прикладного програмування

ARPU – Average Revenue Per User – середній дохід на одного користувача

ARR – Annual Recurring Revenue – річний повторюваний дохід

AWS – Amazon Web Services – хмарна інфраструктура Amazon

BPM – Business Process Management – управління бізнес-процесами

CI/CD – Continuous Integration / Continuous Deployment – безперервна інтеграція та розгортання

CRUD – Create–Read–Update–Delete – операції створення, читання, оновлення й видалення даних

DFD – Data Flow Diagram – діаграма потоків даних

ECS – Elastic Container Service – сервіс контейнеризації AWS

ERD – Entity Relationship Diagram – діаграма «сутність–зв’язок»

FK – Foreign Key – зовнішній ключ

HTTP – HyperText Transfer Protocol – протокол передавання гіпертексту

ID – Identifier – ідентифікатор

JWT – JSON Web Token – формат токена для автентифікації

KPI – Key Performance Indicators – ключові показники ефективності

MRR – Monthly Recurring Revenue – щомісячний повторюваний дохід

OTP – One-Time Password – одноразовий пароль

PK – Primary Key – первинний ключ

PMS – Property Management System – система управління нерухомістю

REST – Representational State Transfer – архітектурний стиль веб-сервісів

RPO – Recovery Point Objective – допустима втрата даних у разі аварії

RTO – Recovery Time Objective – допустимий час відновлення після збою

ВСТУП

У сучасних умовах цифрової трансформації ринку нерухомості суттєво зростає потреба в інтегрованих інформаційних системах, здатних забезпечувати комплексну підтримку процесів управління об’єктами, орендарями, фінансовими операціями, аналітикою та взаємодією з користувачами. Розрізненість наявних програмних рішень, відсутність єдиного інформаційного простору та обмежені можливості локальних систем створюють низку проблем, серед яких фрагментація даних, недостатній рівень автоматизації, слабка аналітична складова та відсутність масштабованих механізмів інтеграції з мобільними сервісами й платіжною інфраструктурою. Ці чинники зумовлюють актуальність розроблення клієнт–серверної платформи управління нерухомістю, яка поєднує функції білінгу, ведення договорів, обліку платежів, комунікацій і моніторингу сервісних заявок.

Об’єкт дослідження – процеси інформаційного забезпечення управління нерухомістю в умовах використання сучасних цифрових сервісів. **Предмет дослідження** – методи, моделі та програмні засоби побудови інтегрованої клієнт–серверної системи управління нерухомістю.

Мета дослідження полягає у розробленні програмного забезпечення для комплексної підтримки операцій управління нерухомістю з використанням сучасних технологій веб- та мобільної розробки, а також методів структурного та об’єктно-орієнтованого моделювання.

Для досягнення поставленої мети необхідним є виконання таких дослідницьких **завдань**:

- обґрунтувати актуальність і проблематику предметної області;
- проаналізувати існуючі системи та визначити їх обмеження;
- сформувати вимоги до програмного забезпечення;
- побудувати моделі архітектури та структури даних;
- розробити клієнтські та серверні компоненти;
- виконати тестування та оцінку ефективності запропонованого рішення.

Методи дослідження ґрунтуються на застосуванні структурного, функціонального та об’єктно-орієнтованого моделювання, UML-діаграмування, методів проєктування баз даних, методів програмної інженерії, REST-підходу до побудови API, а також інструментів модульного, інтеграційного та навантажувального тестування.

Наукова новизна полягає в удосконаленні архітектури програмної платформи управління нерухомістю шляхом інтеграції мобільного клієнта, серверної частини та платіжних сервісів на основі єдиного REST-інтерфейсу; у розробленні узгодженої моделі даних, орієнтованої на автоматизацію процесів обліку й аналітики; у впровадженні підходу до поєднання функціонального моделювання бізнес-процесів із компонентним описом системи, що забезпечує гнучкість масштабування та підтримку модульності.

Практичне значення результатів роботи полягає у можливості безпосереднього впровадження розробленої системи в діяльність девелоперських компаній, керуючих фірм та житлово-комунальних організацій. Платформа дає змогу зменшити адміністративні витрати, мінімізувати кількість помилок у фінансових розрахунках і забезпечити прозорість взаєморозрахунків між мешканцями та власниками. Використання автоматизованої аналітики й інтегрованих засобів безпеки підвищує рівень довіри між сторонами та створює передумови для подальшого розвитку цифрових сервісів управління нерухомістю.

Теоретичне значення. Робота робить внесок у розвиток галузі інженерії програмного забезпечення, зокрема в частині застосування сучасних методів архітектурного проєктування, проєктування баз даних і безпечних вебсистем. Вона демонструє практичне застосування принципів системного підходу до побудови інформаційних платформ та аналітичних модулів.

Апробація результатів здійснювалася під час представлення матеріалів на наукових заходах НУБіП України. Основні положення роботи були опубліковані та презентовані на *XVI Міжнародній науково-практичній конференції молодих учених «Інформаційні технології: економіка, техніка,*

освіта», де у вигляді тез було представлено результати розроблення програмного забезпечення платформи для управління нерухомістю та цифрової взаємодії між орендарями й орендодавцями (електронний ресурс: http://econference.nubip.edu.ua/index.php/itete/XVI/paper/view/4059).

Додаткову апробацію отримано в межах *Інтернет-конференції НУБіП України «Теоретичні та прикладні аспекти розробки комп'ютерних систем – 2025»*, де було представлено тези «Інтелектуальна система автоматизації управління орендою нерухомості з підтримкою цифрової взаємодії» (О.П. Мотлюк), що підтверджує актуальність тематики та практичну значущість отриманих результатів (електронний ресурс: http://econference.nubip.edu.ua/index.php/taacsd/2025/paper/view/3638).

Структура магістерської роботи складається зі вступу, чотирьох змістових розділів, висновків, списку використаних джерел та додатків. У першому розділі проведено системний аналіз предметної області та сформовано постановку задачі. Другий розділ присвячено моделюванню та проєктуванню програмної системи. У третьому розділі подано реалізацію клієнтських і серверних компонентів та результати тестування. Четвертий розділ містить оцінку ефективності функціонування розробленої системи та обґрунтування перспектив її розвитку. Загальний обсяг роботи становить 112 сторінок, містить 36 рисунків, 23 таблиці та 45 джерел.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Сутність і особливості управління нерухомістю

Управління нерухомістю – це комплекс організаційних та техніко-економічних процесів, що охоплюють інвентаризацію об’єктів, ведення договорів оренди, комунікацію між учасниками (орендодавець, орендар, менеджер), облік платежів, технічне обслуговування, планування витрат і формування керівної звітності. Традиційні підходи базуються на розрізнених інструментах (таблиці, месенджери, бухгалтерські програми), що призводить до дублювання даних, затримок, помилок і відсутності цілісної аналітики. Сучасний підхід – інтегрована **Property Management System (PMS)**, що об’єднує всі процеси в єдиній платформі з ролями доступу, централізованою базою даних, автоматизованими платежами та дашбордом ключових показників ефективності (KPI).

У системі взаємодіють кілька груп користувачів із різними цілями та правами доступу. Орендодавець/власник потребує прозорої фінансової звітності, контролю заповнюваності та інструментів ціноутворення; орендар – передбачуваних платежів, зрозумілих рахунків і зручних каналів комунікації; керуюча компанія – засобів диспетчеризації заявок, планування технічних робіт і контролю SLA; фінансово-бухгалтерський блок – уніфікованого інвойсингу, звірки оплат і інтеграції з платіжними сервісами; адміністратор ІС – механізмів автентифікації, авторизації та аудиту дій.

Типовий життєвий цикл включає:

1. маркетинг і заселення (оверка потенційних орендарів, укладання договорів, депозити);
2. експлуатацію (ведення об’єктів і юнітів, облік показників, планові/позапланові роботи);

3. фінансове адміністрування (нарахування, інвойсинг, онлайн-оплати, облік прострочень);
4. комунікації та підтримку (заявки, оголошення, повідомлення);
5. аналітику та звітність (cashflow, окупність, заповнюваність, дисципліна платежів);
6. ревізію та розвиток (аудит доступу, оцінка ефективності, оновлення фонду). Інтегрована PMS забезпечує наскрізну підтримку кожної фази, зменшує кількість ручних операцій і скорочує час прийняття рішень.

До критичних процесів належать управління договорами (створення, пролонгація, індексація ставок), виставлення рахунків і приймання оплат (у т.ч. через Stripe/PayPal), облік і обробка заявок на обслуговування (пріоритезація, планування ресурсів, контроль виконання), а також формування управлінської звітності. Точки контролю – підписання договору, виписка інвойсу, отримання платежу, фіксація прострочення, закриття заявки, щомісячне формування звітів. Їхня автоматизація зменшує ризики помилок і підвищує прозорість взаємодії.

Осердям є узгоджена модель даних, яка охоплює сутності *Users, Properties, Units, Leases, Payments, WalletHistory* та їхні зв’язки. Така структура забезпечує цілісність і відстежуваність фінансових подій, підтримку історичності (audit trail) та можливість ефективних агрегувань для аналітики. Наявність уніфікованих ідентифікаторів, таймстампів, статусів операцій і політик валідації є обов’язковою умовою якості даних.

Для оперативного управління використовують: occupancy rate (заповнюваність фонду), delinquency rate (частка прострочених платежів), MRR/ARR (періодичний дохід), cashflow (баланс доходів і витрат), ARPU (середній дохід на орендаря), середній час обробки заявки (SLA). Їхня візуалізація на дашборді дозволяє відстежувати тренди, вчасно реагувати на ризики та обґрунтовувати управлінські рішення.

З огляду на чутливість персональних і фінансових даних, PMS має підтримувати багаторівневу безпеку: автентифікацію з токенами JWT, двофакторну перевірку (2FA) для критичних операцій, шифрування

конфіденційних полів (AES-256) і передавання даних через захищені канали (TLS). Журнали подій і аудит доступу забезпечують відтворюваність дій користувачів і відповідність внутрішнім політикам та зовнішнім вимогам.

Перехід від розрізнених інструментів до інтегрованої платформи дає вимірювані переваги: скорочення операційних витрат і кількості помилок, зростання швидкості обробки заявок, підвищення дисципліни платежів, поліпшення клієнтського досвіду та прозорості для всіх сторін. Завдяки уніфікованій моделі даних та автоматизованим регламентам PMS стає джерелом достовірної управлінської інформації й основою для подальшого впровадження аналітики прогнозування (AI/ML) і розширень інтеграцій.

1.2. Інформаційні системи управління нерухомістю: класифікація та аналіз існуючих рішень

Для систем управління нерухомістю характерна значна різноманітність за призначенням, архітектурою та масштабом застосування, тому доцільно виконати їх класифікацію за ключовими ознаками.

Таблиця 1.1

Класифікація інформаційних систем управління нерухомістю

Критерій	Типи систем	Характеристика
Рівень автоматизації	Локальні / Інтегровані	Локальні закривають окремі задачі; інтегровані – повний цикл
Цільова аудиторія	Орендодавці, Керуючі компанії, Орендарі	Відмінності у правах і сценаріях
Архітектура	Моноліт / Мікросервіси / Гібрид	Впливає на масштабованість і швидкодію
Розгортання	Он-преміс / Хмара	Хмара дає SLA/резервування/оновлення
Канали доступу	Веб, Мобільні, Десктоп	Mobile-first – сучасний стандарт
Функціонал	Фінанси, Комунікації, Техобслуговування, Аналітика	Універсальні рішення об’єднують модулі

У таблиці 1.1 подано узагальнену класифікацію інформаційних систем управління нерухомістю за рівнем автоматизації, цільовою аудиторією, архітектурним підходом, способом розгортання, каналами доступу та функціональною орієнтацією. Такий підхід дозволяє виокремити відмінності між локальними й інтегрованими рішеннями, системами для окремих груп користувачів (орендодавців, керуючих компаній, орендарів) та комплексними платформами, що охоплюють повний життєвий цикл управління об’єктами.

Запропоновані критерії класифікації є важливими для подальшого проєктування власної платформи, оскільки впливають на вибір архітектури (моноліт чи мікросервіси), технологій розгортання (он-преміс чи хмара), типів клієнтських застосунків (web, mobile, desktop) і необхідного набору функціональних модулів (фінанси, комунікації, техобслуговування, аналітика). Саме аналіз цих вимірів дозволяє обґрунтувати місце й тип розроблюваної системи серед існуючих рішень.

Ринок програмних систем управління нерухомістю представлений низкою потужних рішень, орієнтованих насамперед на США та ринки Західної Європи. До найвідоміших відносять Yardi Voyager, AppFolio, Buildium, SimplifyEm, Domopult, а також спеціалізовані рішення на кшталт RentSyst. Кожна з цих систем має власну цільову нішу, набір ключових можливостей і обмежень, що визначає їхню придатність для використання в певних умовах. Узагальнена порівняльна характеристика провідних PMS наведена в таблиці 1.2.

Таблиця 1.2

Порівняльна характеристика провідних PMS

Система	Країна	Ключові можливості	Переваги	Недоліки
Yardi Voyager	США	CRM, бухгалтерія, техобслуговування, аналітика	Максимально повний стек	Дуже висока ціна, складність впровадження
AppFolio	США	Онлайн-платежі, документообіг, мобільний доступ	Зручність, хмара	Немає локалізації/правил для України
Buildium	США	Оренда, платежі, заявки	Простота	Менша гнучкість кастомізації

Система	Країна	Ключові можливості	Переваги	Недоліки
SimplifyEm	США	Доходи/витрати	Низький поріг входу	Слабка аналітика
Domopult	ЄС	ЖКГ, платежі, обслуговування	Європейський ринок, мультимова	Не фокусується на приватній оренді
RentSyst	Україна	Флот/оренда авто	Гнучка архітектура	Не для житлової нерухомості

Проведений аналіз демонструє, що глобальні рішення, такі як Yardi Voyager та AppFolio, пропонують максимально широкий функціональний стек (CRM, бухгалтерія, технічне обслуговування, аналітика, мобільний доступ), однак мають високу вартість ліцензування та складність впровадження, а також відсутність локалізації під українське законодавство й практику. Buildium і SimplifyEm позиціонуються як більш прості та доступні системи, але поступаються гнучкістю налаштувань та глибиною аналітики. Domopult орієнтований переважно на сегмент ЖКГ у країнах ЄС і не фокусується на задачах приватної оренди. Український продукт RentSyst демонструє хорошу гнучкість архітектури, однак спеціалізується на управлінні автопарком, а не житловою чи комерційною нерухомістю.

Таким чином, аналіз систем-аналогів показує, що на локальному ринку бракує платформи, яка одночасно поєднувала б підтримку українського контексту, інтеграції з платіжними сервісами (Stripe/PayPal), розвинену аналітику KPI, адаптивний користувацький інтерфейс та прозорий облік фінансових операцій. Виявлена прогалина обґрунтовує доцільність розроблення власної інтегрованої платформи управління нерухомістю, орієнтованої на потреби вітчизняних орендодавців та орендарів.

1.3. Актуальні проблеми та вимоги до інтегрованої платформи управління нерухомістю

Сучасна практика управління нерухомістю страждає насамперед від фрагментації даних: інформація розкидана між електронними таблицями,

месенджерами, поштою та локальними базами різних менеджерів, унаслідок чого виникають дублювання записів, розбіжності у версіях і втрата історії змін. Запропонована платформа формує єдине джерело правди завдяки централізованій реляційній БД (PostgreSQL) з нормалізованою моделлю (Users, Properties, Units, Leases, Payments, WalletHistory), уніфікованими ідентифікаторами, часовими мітками та ролями доступу (Tenant/Landlord/Admin). Усі операції виконуються через формалізовані REST-інтерфейси бекенду на Node.js, що забезпечує контроль цілісності, а журнали подій дозволяють відтворити будь-яку зміну даних.

Другий блок проблем пов’язаний із непрозорістю фінансів: вручну сформовані інвойси надходять із запізненням, нагадування не автоматизовані, звірка оплат трудомістка, що породжує зростання простроченої дебіторки і непередбачуваний cashflow. Система усуває це завдяки вбудованому білінгу: автоматичному календарю нарахувань, генерації інвойсів з нумерацією, правилам пені/знижок, а також прямим інтеграціям зі Stripe і PayPal (обробка webhook подій для зміни статусів paid/pending/overdue без ручного втручання). Нагадування (email/пуш) та сценарії ескалації скорочують кількість прострочень, а звірка та експорт у стандартні формати спрощують роботу бухгалтерії.

Третя проблема – слабка аналітика. За відсутності централізованих даних складно оперативно відповісти на базові управлінські питання: яка заповнюваність фонду, яку частку займають прострочені платежі, як змінюються доходи й витрати по місяцях, які об’єкти недоотримують дохід. Платформа надає дашборд із ключовими KPI – occupancy rate, delinquency rate, MRR/ARR, cashflow, ARPU, середній час обробки заявок – та інтерактивні графіки (доходи/витрати, розподіл статусів платежів, структура зайнятості), що дозволяє переходити від реактивного до проактивного управління.

Болісним є й вимір безпеки: у типових рішеннях відсутні 2FA, централізований аудит та наскрізне шифрування. У запропонованому рішенні реалізовано автентифікацію на основі JWT, двофакторну перевірку (2FA) для

критичних дій, шифрування конфіденційних полів за алгоритмом AES-256 та обов’язкове TLS для транспорту даних; додатково ведеться аудит доступу й дій користувачів. Нарешті, UX-проблематика: перевантажені або «захардкожені» інтерфейси без мобільного сценарію ускладнюють роботу користувачів і збільшують кількість помилок. Використання Flutter на фронтенді з Redux Toolkit і Styled Components забезпечує адаптивний, послідовний і швидкий інтерфейс для вебу та мобільних платформ, мінімізує ручні кроки та спирається на зрозумілі шаблони взаємодії. Сукупно ці рішення прибирають ключові больові точки предметної області та створюють передумови для масштабованого, прозорого й безпечного управління нерухомістю.

На основі виявлених проблем можна сформувати узагальнені вимоги до інтегрованої системи управління нерухомістю. До функціональних вимог належать підтримка повного життєвого циклу управління об’єктами й юнітами, ведення договорів оренди, автоматизований облік платежів, модуль аналітики з KPI, а також підсистема комунікацій і заявок. До нефункціональних – вимоги до продуктивності, доступності, безпеки, масштабованості, зручності інтерфейсу та сумісності із зовнішніми сервісами.

1.4. Концептуальна модель інтегрованої Property Management System

Концептуальна модель інтегрованої платформи управління нерухомістю (Property Management System, PMS) визначає загальні принципи організації інформаційних потоків, структуру доменних сутностей та логіку взаємодії користувачів із системою. Її побудова ґрунтується на необхідності об’єднати в єдиному цифровому середовищі процеси управління об’єктами та юнітами, ведення договорів оренди, контролю фінансових операцій, обробки технічних заявок, комунікацій між учасниками та формування аналітичної звітності.

У системі реалізовано рольову модель, що охоплює орендарів, орендодавців та адміністратора. Кожна роль має визначений набір прав доступу до функціональних компонентів платформи. Таке розмежування дозволяє забезпечити керованість бізнес-процесів, цілісність даних та захищеність

конфіденційної інформації. Діаграма прецедентів (рис. 1.1) відображає рольову модель платформи з акторами Tenant, Landlord, Admin (та гість для реєстрації) і межами доступу до підсистем.

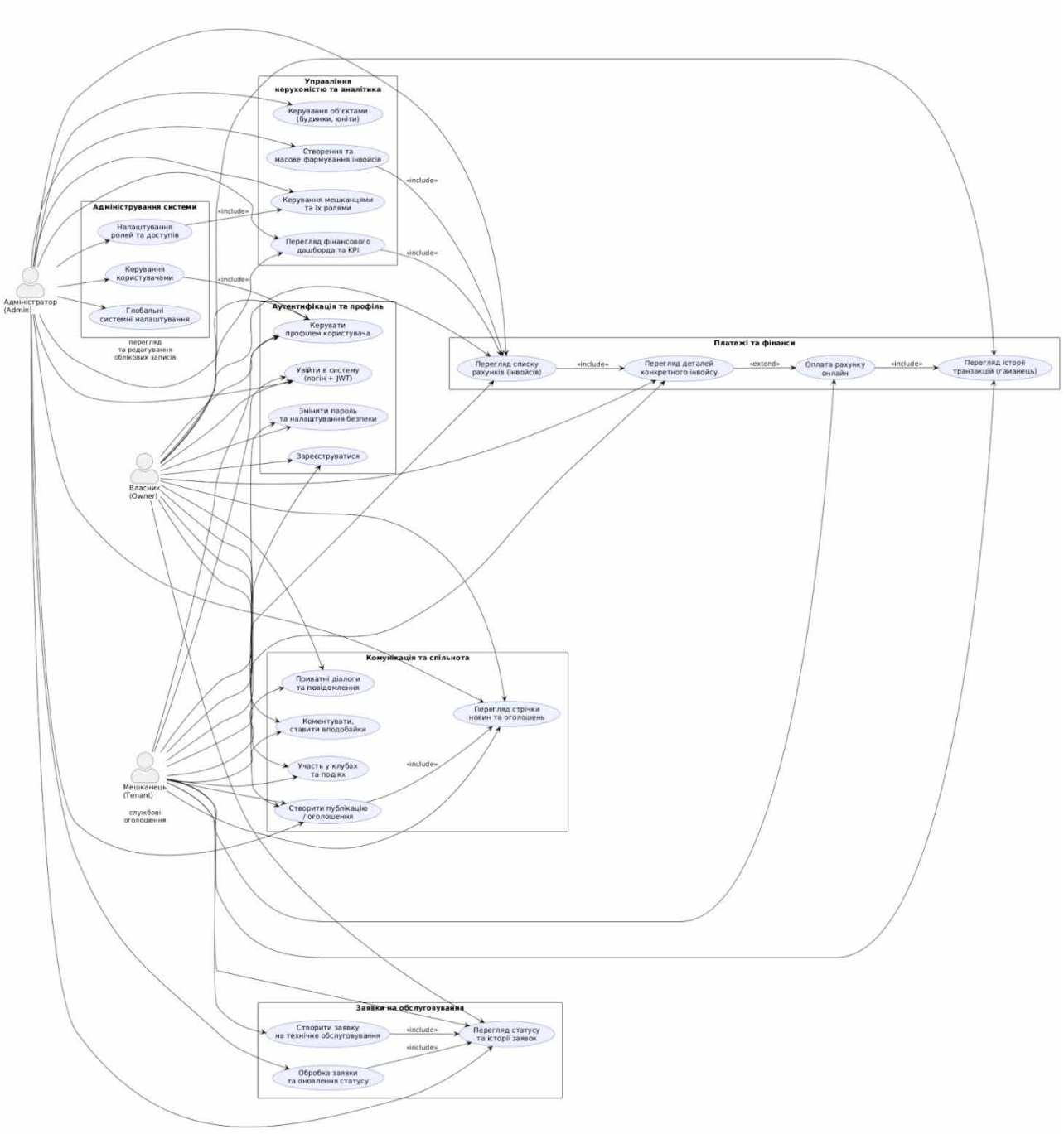


Рис. 1.1 Діаграма прецедентів (Use Case)

Система охоплює керування об’єктами та юнітами, життєвий цикл договорів, оплату інвойсів через Stripe/PayPal та перегляд фінансової історії, комунікації й заявки на обслуговування. Ланцюг include фіксує обов’язкові

кроки («перелік інвойсів → деталі → оплата → історія»), extend — опційні дії (напр., 2FA під час входу). Передумови — чинний обліковий запис і договір; результати — оновлення сутностей, зміна статусів, аудит і сповіщення. Модель покриває вимоги FR-01…FR-06 і слугує основою для подальшого трасування до БД та REST-API.

Для розуміння функціонування платформи на системному рівні доцільним є подання моделі у вигляді діаграми потоків даних. На схемі верхнього рівня (рис. 1.2) відображається взаємодія зовнішніх суб'єктів із центральним процесом «Інформаційна система управління житловими комплексами».

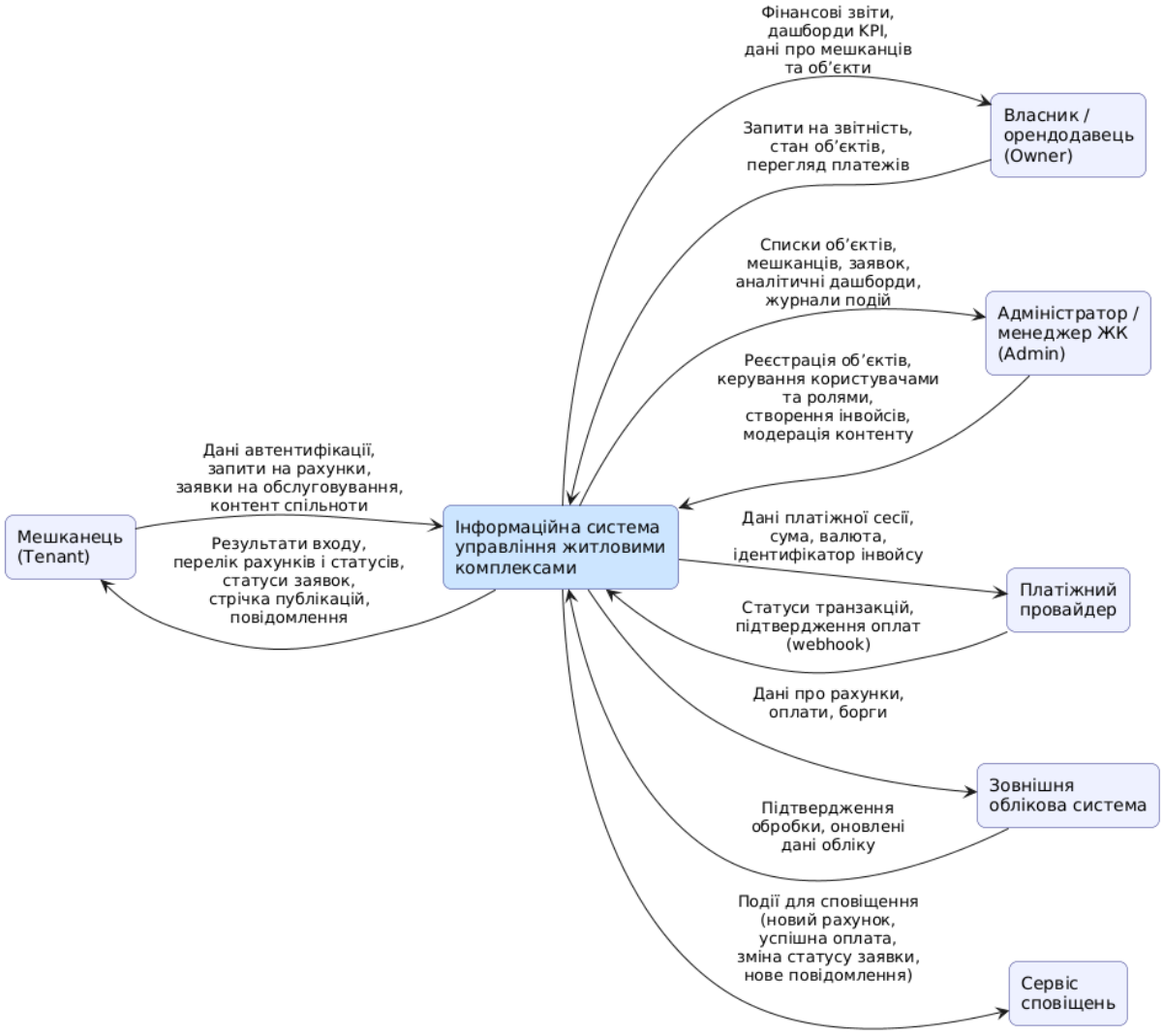


Рис. 1.2 DFD-схема верхнього рівня (Level 0)

Мешканець передає дані автентифікації, запити на рахунки та заявки й отримує результати входу, статуси рахунків і сповіщення. Власник формує

запити щодо стану об’єктів і платежів та одержує фінансові звіти й KPI-дашборди. Адміністратор реєструє об’єкти, керує користувачами й інвойсами та отримує зведені списки, аналітику і журнал подій. Інтеграційні потоки охоплюють платіжного провайдера (створення платіжних сесій і webhook-статуси), зовнішню облікову систему (обмін даними про рахунки й оплати) та сервіс сповіщень (доставка push/e-mail подій). Усі дані акумулюються в централізованому сховищі з журналюванням, що забезпечує цілісність процесу «договір → інвойс → оплата → звітність» і підтримує актуальність аналітики.

Розширена система взаємодій охоплює також модулі автоматичної перевірки даних, періодичні службові процеси та сервіс моніторингу. Під час роботи мешканець може ініціювати оновлення профілю, прикріплювати підтверджуючі документи та створювати запити на технічне обслуговування, які надходять менеджеру або адміністратору. Власник отримує деталізоване відображення руху коштів, прогнозний аналіз платежів та інформацію про відхилення у графіку надходжень. Адміністратор системи контролює коректність даних об’єктів, проводить ручні перевірки проблемних інвойсів і керує правами доступу для всіх ролей.

Інтеграції з платіжними сервісами доповнюються автоматизованими механізмами повторних запитів, перевіркою транзакцій у затримці та синхронізацією декількох статусів оплати. Зовнішня облікова система отримує зміни щодо договорів, стану квартир, балансу платежів і передає у платформу ключові фінансові дані для формування агрегованих звітів. Сервіс сповіщень не лише надсилає події, але й фіксує історію доставки, повторні відправлення та сегментацію користувачів.

Високорівнева архітектура (рис. 1.3) складається з клієнтського рівня (веб та Flutter-мобільний застосунок), що звертається до REST API з JWT, і серверного рівня на Node.js із модульною організацією (автентифікація, управління об’єктами, платежі, заявки, спільнота, аналітика). Дані зберігаються у централізованій БД PostgreSQL. Інтеграційний шар взаємодіє з платіжним провайдером (сесії, webhook-статуси), зовнішньою обліковою системою та

сервісом сповіщень, забезпечуючи цілісність і масштабованість платформи.

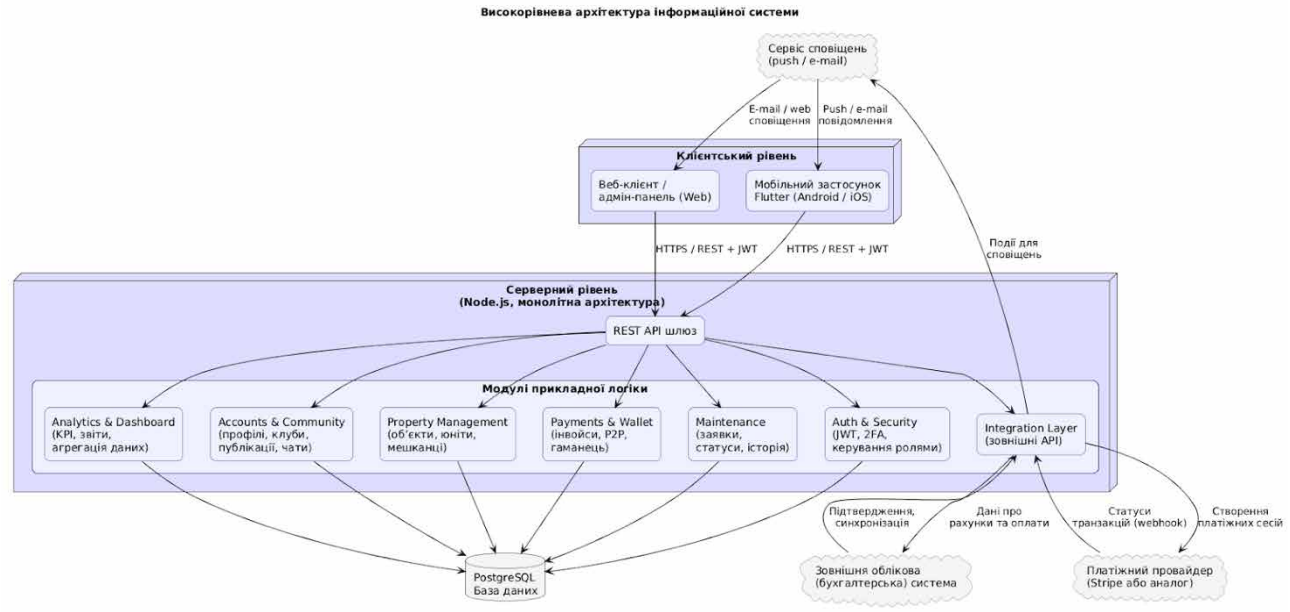


Рис. 1.3 Високорівнева архітектура платформи

Концептуальна модель також передбачає наявність окремих логічних модулів: підсистеми автентифікації та авторизації, модуля управління об’єктами та юнітами, підсистеми договорів і платежів, модуля аналітики з KPI, підсистеми комунікацій і заявок, а також адміністративного модуля для налаштування параметрів системи.

Реалізація концептуальної моделі спирається на принципи низького рівня зв’язності між компонентами, прозорості інтерфейсів, можливості масштабування та інтеграції з зовнішніми сервісами. Важливою складовою є підтримка аудиту дій користувачів і моніторингу подій, що підвищує рівень інформаційної безпеки та забезпечує контроль над критичними операціями. Описана модель є базою для подальшого проєктування логічної та фізичної архітектури платформи.

1.5. Інформаційна підтримка та показники ефективності в системах управління нерухомістю

Ефективне управління портфелем нерухомості неможливе без належної інформаційної підтримки, що забезпечує своєчасний доступ до даних про

зайнятість юнітів, стан розрахунків за договорами, динаміку доходів і витрат, а також структуру клієнтської бази. Інтегрована платформа формує єдине джерело даних, на основі якого обчислюються кількісні показники, що дають змогу об’єктивно оцінювати поточний стан об’єктів та результативність управлінських рішень. Сукупність ключових показників ефективності (KPI), які використовуються в системі, наведено в табл. 1.3.

Таблиця 1.3

Основні показники ефективності (KPI) платформи управління нерухомістю

KPI	Формула	Інтерпретація
Occupancy Rate	зайняті юніти / всі юніти	Заповнюваність фонду
Delinquency Rate	прострочені платежі / всі платежі	Дисципліна розрахунків орендарів (платежів)
MRR (Rent)	сума активних оренд / міс	Щомісячний повторюваний дохід
Cashflow	Σ(доходи − витрати)	Ліквідність та фінансова стійкість
ARPU	загальний дохід / кількість орендарів	Середній дохід на одного користувача

У табл. 1.11 подано основні KPI платформи: коефіцієнт заповнюваності фонду (Occupancy Rate), частку прострочених платежів (Delinquency Rate), щомісячний повторюваний дохід (MRR), чистий грошовий потік (Cashflow) та середній дохід на одного орендаря (ARPU). Occupancy Rate визначається як відношення кількості зайнятих юнітів до загальної кількості юнітів і характеризує рівень використання наявних площ. Delinquency Rate відображає дисципліну розрахунків орендарів, оскільки показує частку прострочених платежів серед усіх нарахованих. Показник MRR дає змогу оцінити стабільність регулярних орендних надходжень, Cashflow – здатність об’єкта або портфеля генерувати позитивний грошовий потік, а ARPU – середню монетизацію одного орендаря за певний період.

Графічна інтерпретація зазначених показників дає змогу швидко виявляти тенденції та відхилення. На рис. 1.4 відображено динаміку доходів і витрат за

місяцями, що дозволяє аналізувати сезонні коливання, оцінювати маржинальність діяльності та своєчасно реагувати на зростання витрат.

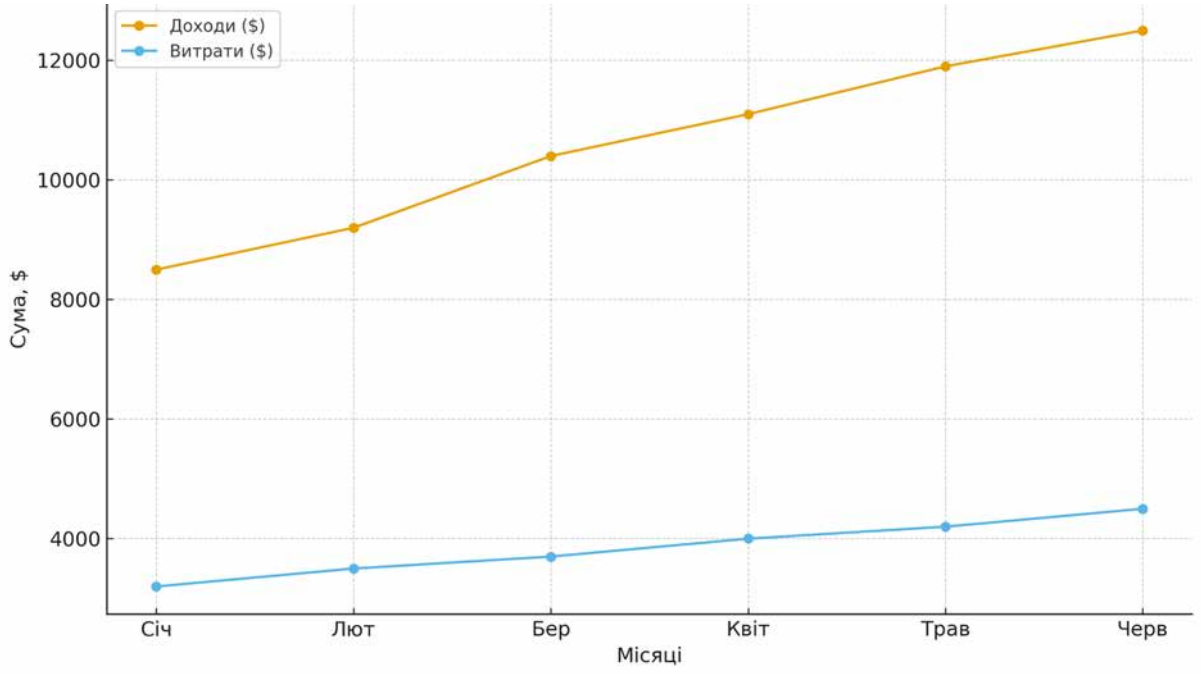


Рис. 1.4 Динаміка доходів та витрат за місяцями

Структуру заповнюваності фонду за станами «зайнято» та «вільно» продемонстровано на рис. 1.5; цей розподіл використовується для контролю ефективності використання площ і планування маркетингових заходів щодо залучення нових орендарів.

На рис. 1.6 подано розподіл платежів за статусами «оплачені», «очікують» і «прострочені», що дозволяє оперативно оцінювати ризики неплатежів і приймати рішення щодо зміни умов договорів або посилення роботи з боржниками.

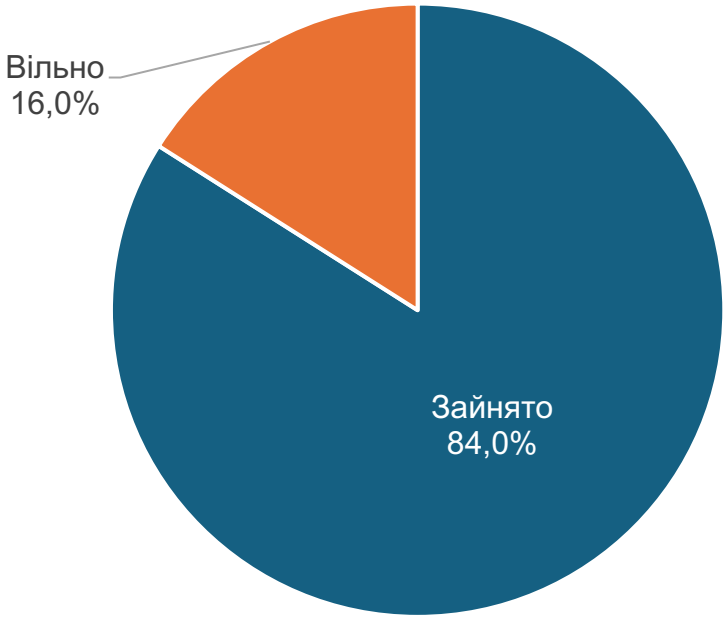


Рис. 1.5 Структура заповнюваності об’єктів, %

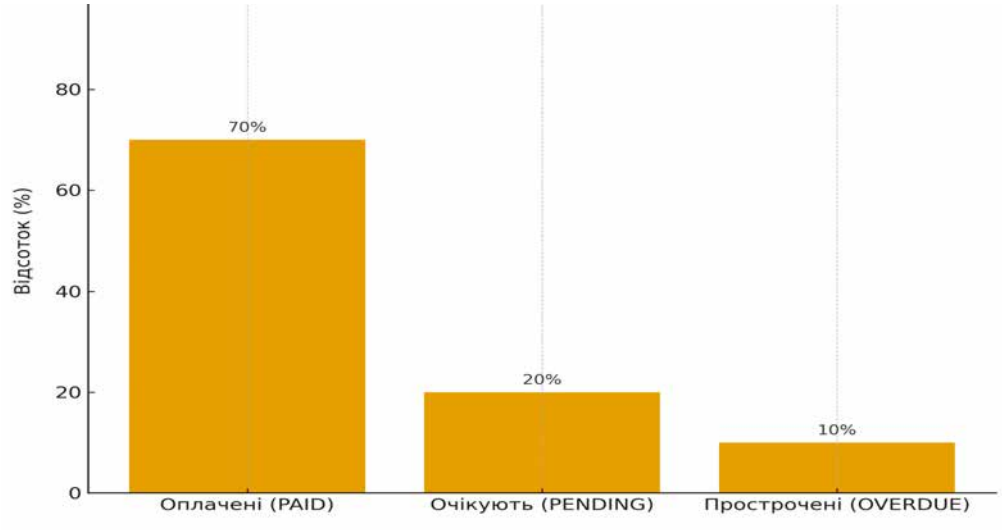


Рис. 1.6 Розподіл платежів за статусами, %.

Поєднання табличного та графічного подання інформації створює основу для побудови аналітичних панелей у системі управління нерухомістю. Користувачі платформи отримують можливість не лише переглядати поточні значення KPI, а й аналізувати їх динаміку, порівнювати різні об’єкти між собою та оцінювати вплив управлінських рішень на фінансові результати. Таким чином, інформаційна підтримка в поєднанні з формалізованими показниками ефективності перетворює платформу управління нерухомістю на інструмент обґрунтованого прийняття рішень.

1.6. Постановка задачі магістерського дослідження

Результати аналізу предметної області, розглянуті в попередніх підрозділах, засвідчили, що сучасні процеси управління житловою й комерційною нерухомістю характеризуються високою фрагментованістю даних, нерівномірністю інформаційних потоків, низьким рівнем автоматизації фінансових операцій та недостатньою аналітичною підтримкою управлінських рішень. Існуючі програмні продукти, попри широкий функціональний спектр, не забезпечують комплексної підтримки українського ринку оренди, не враховують національні регуляторні вимоги та потребують значної адаптації для інтеграції з локальними сервісами електронних платежів. Це створює потребу у розробленні цілісної інформаційної системи, здатної забезпечити наскрізну обробку даних, прозорість фінансових процесів, централізовану взаємодію між користувачами та можливість аналітичного моніторингу в реальному часі.

Об’єктом магістерського дослідження є процес управління нерухомістю в умовах застосування сучасних інформаційних технологій. Предметом дослідження виступають принципи побудови, методи функціонування та програмні засоби інтегрованої платформи управління нерухомістю, спрямованої на автоматизацію ключових бізнес-процесів, підвищення прозорості взаємодії між учасниками та підтримку аналітичного супроводу діяльності. Метою роботи є створення програмної платформи, що забезпечує централізоване ведення об’єктів і договорів оренди, автоматизовану обробку електронних платежів, формування аналітичних панелей із ключовими показниками ефективності, а також належний рівень інформаційної безпеки та зручності використання через веб- і мобільні інтерфейси.

Досягнення поставленої мети ґрунтується на послідовному розв’язанні комплексу підзавдань, що охоплюють формування бізнес-вимог, визначення функціональних та нефункціональних характеристик програмного продукту, проєктування концептуальної, логічної й фізичної моделей даних, розроблення архітектури програмної системи та реалізацію її основних модулів. У межах

дослідження також передбачається оцінювання якості програмних рішень, включно з перевіркою продуктивності API, точністю розрахунків, коректністю відображення ключових показників ефективності, відповідністю вимогам до доступності та інформаційної безпеки, а також визначенням готовності системи до експлуатації в реальних умовах.

Кінцевим результатом магістерського дослідження є створення інтегрованої платформи управління нерухомістю, яка поєднує централізоване ведення об’єктів та договорів, облік і проведення електронних платежів, розгорнуту аналітику та інструменти контролю доступу. Успішність розробленої системи визначається її здатністю забезпечувати стабільну роботу, відповідати встановленим вимогам щодо точності розрахунків і швидкодії, підтримувати необхідний рівень інформаційної безпеки та забезпечувати позитивний користувацький досвід у багатоакторному середовищі.

Висновки до першого розділу

У першому розділі проведено комплексний аналіз предметної області управління нерухомістю, що дало змогу окреслити специфіку сучасних практик, визначити потреби основних користувачів та ідентифікувати обмеження існуючих програмних рішень. Дослідження показало, що на ринку відсутня платформа, адаптована до українських умов, яка поєднувала б централізовану роботу з об’єктами та договорами, наскрізний білінг з підтримкою електронних платежів, аналітику ключових показників ефективності та інтегрований мобільний інтерфейс. У ході аналізу було виявлено низку системних проблем, характерних для традиційних підходів до управління нерухомістю: роз’єднаність даних, непрозорість фінансових потоків, обмеженість аналітичних засобів, недостатній рівень безпеки та застарілі моделі користувацької взаємодії.

Узагальнення результатів дозволило сформувати концепцію інтегрованої Property Management System із модульною структурою та чітко окресленими функціональними зонами: управління об’єктами й юнітами, життєвий цикл договорів і платежів, ведення фінансової історії, комунікації та технічні заявки,

аналітичний супровід. Сформульовано базові технологічні вимоги: використання сучасних інструментів фронтенду/бекенду, реляційної моделі даних, механізмів захисту інформації та автоматизованого розгортання.

На основі моделювання побудовано інформаційну структуру системи, що узгоджує взаємодію користувачів, об’єктів і процесів. Визначено первинний набір KPI (стан фонду, платіжна дисципліна, регулярність доходів, фінансова стійкість), який слугує підґрунтям для дашбордів оперативної підтримки управлінських рішень.

Систематизовано функціональні та нефункціональні вимоги: підтримка повного циклу договорів/платежів, рольове розмежування доступу, аналітика, продуктивність і доступність сервісу, відповідність вимогам безпеки. Окреслено орієнтири якості — точність фінансових розрахунків, стабільність платіжних інтеграцій, зменшення прострочень, зростання заповнюваності та скорочення часу реакції на заявки. Отже, розділ створює теоретико-методологічну основу для подальшого проєктування й розроблення платформи та обґрунтовує актуальність її впровадження.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Обґрунтування вибору технологічного стеку та формування вимог до системи

Проєктована програмна платформа призначена для підтримки повного життєвого циклу управління нерухомістю та організації взаємодії між орендарями, орендодавцями, менеджерами об’єктів і адміністраторами. Для такої системи характерні кросплатформний доступ, робота з фінансовими транзакціями в режимі, наближеному до реального часу, інтеграція з платіжними сервісами та підвищені вимоги до безпеки й відмовостійкості. Це зумовлює необхідність узгодженого вибору технологічного стеку та чіткого формування функціональних і нефункціональних вимог, які надалі визначатимуть архітектурні рішення, структуру даних і підходи до розгортання. На рисунку 2.1 схематично показано місце обраних технологій у загальній структурі платформи.

Клієнтська частина системи охоплює два основних канали доступу: мобільний застосунок для кінцевих користувачів (керування орендою, перегляд рахунків, подання заявок, отримання сповіщень) та веб-інтерфейс для бек-офісу (адміністративні кабінети, аналітика, управління об’єктами, договорами та тарифами). Для реалізації мобільного клієнта обрано фреймворк **Flutter**, який забезпечує розробку застосунків для Android і iOS з єдиної кодової бази, а за потреби – і веб-версії. Такий підхід істотно зменшує витрати на супровід, спрощує синхронізацію функціональності між платформами та забезпечує узгодженість інтерфейсів. Завдяки власному рушію рендерингу інтерфейсу Flutter забезпечує стабільну продуктивність при роботі з довгими списками об’єктів, історією транзакцій і інтерактивними елементами аналітики, що є критичним для системи управління нерухомістю.

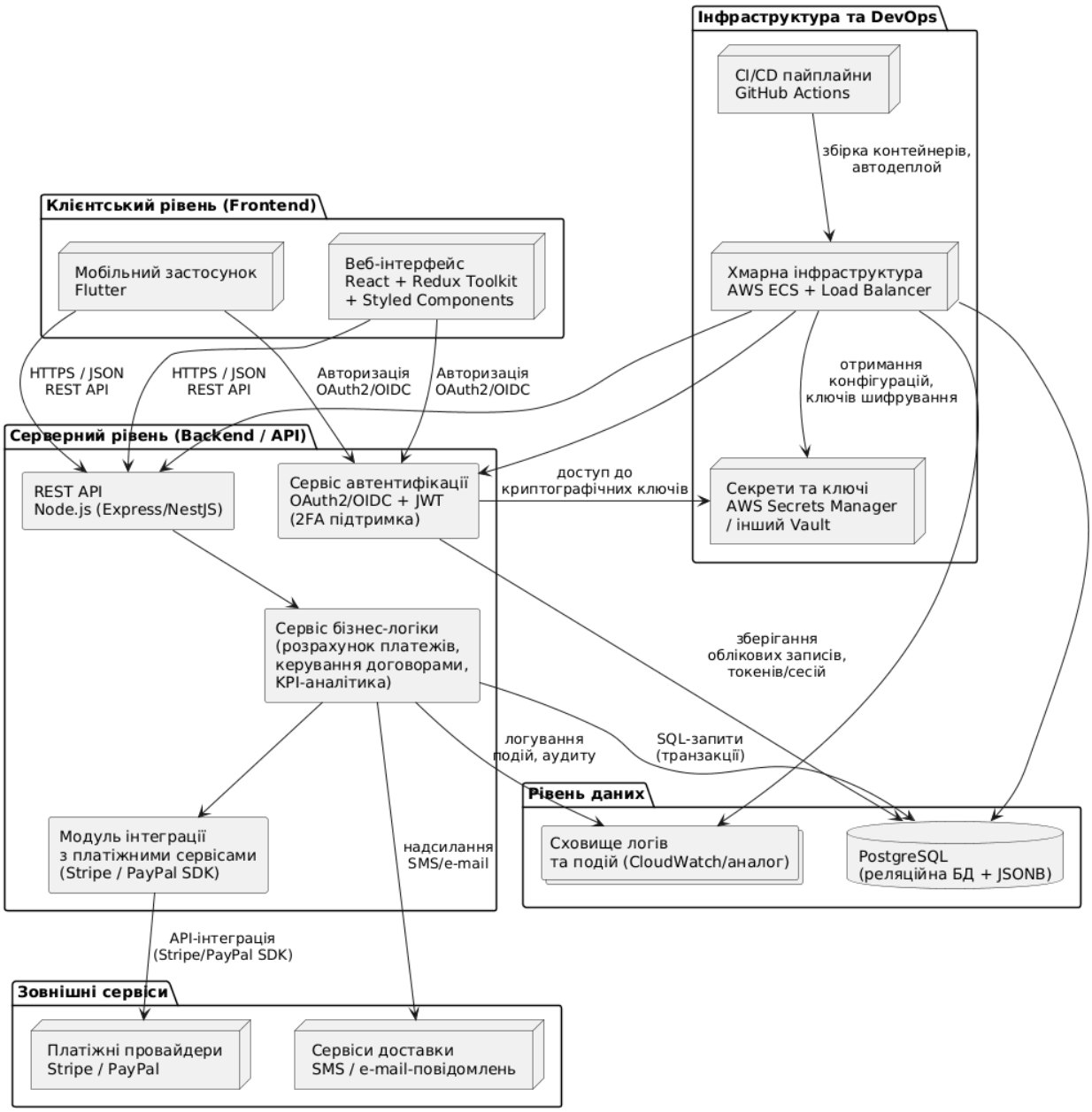


Рис. 2.1 Узагальнена схема розміщення технологічного стеку в архітектурі платформи управління нерухомістю

Для веб-інтерфейсу адміністраторів і менеджерів обрано зв’язку React + Redux Toolkit + Styled Components. React виступає базою для побудови односторінкового застосунку з насиченим інтерфейсом, який включає таблиці, фільтри, дашборди, модальні вікна та багатокрокові форми. Redux Toolkit забезпечує структуроване управління станом, що важливо за наявності численних сутностей (об’єкти, юніти, користувачі, договори, фільтри аналітики) та складних сценаріїв доступу. Використання Styled Components дозволяє

інкапсулювати стилі всередині компонентів, запобігаючи конфліктам та спрощуючи реалізацію єдиної дизайн-системи, включно з підтримкою різних тем і брендингу для окремих груп користувачів.

Серверна частина платформи реалізується на основі Node.js та реляційної СКБД PostgreSQL. Використання Node.js дає змогу застосовувати однакову мову (JavaScript/TypeScript) на фронтенді й бекенді, що спрощує обмін моделями даних, зменшує поріг входу для розробників і полегшує супровід проєкту. Подієво-орієнтована, неблокуюча модель обробки запитів у Node.js є придатною для сценаріїв із великою кількістю одночасних HTTP-звернень, інтеграцій із платіжними сервісами та зовнішніми API, де більшість операцій зводиться до викликів бази даних чи сторонніх сервісів.

Вибір PostgreSQL як основної системи керування базами даних зумовлений характером інформації, з якою працює платформа. Фінансові операції, договори, прив’язка платежів до об’єктів і користувачів, історія змін та аналітичні показники потребують гарантій цілісності, транзакційності та чітко визначених зв’язків між сутностями. PostgreSQL забезпечує підтримку ACID-транзакцій, розвинуті механізми цілісності (зовнішні ключі, обмеження, тригери) та розширену SQL-функціональність, необхідну для побудови звітів і розрахунку показників ефективності. Додаткова підтримка JSONB-полів дозволяє зберігати напівструктуровані дані (метадані транзакцій, відповіді зовнішніх сервісів, гнучкі конфігурації) без відмови від реляційної моделі.

Питання життєвого циклу розгортання та експлуатації вирішуються за рахунок використання GitHub Actions як інструменту безперервної інтеграції та доставки (CI/CD) та AWS ECS як платформи для контейнеризованого розгортання. GitHub Actions забезпечує автоматичний запуск тестів, статичний аналіз коду, збирання контейнерів і публікацію артефактів при кожній зміні в репозиторії, що підвищує якість програмного коду і зменшує ймовірність помилок під час ручного деплойменту. AWS ECS дає можливість розгортати контейнеризовані сервіси у вигляді кластерів, налаштовувати автоматичне масштабування за основними метриками навантаження та ізолювати середовища

(тестове, передексплуатаційне, промислове) при збереженні єдиних принципів конфігурації.

Окремий шар рішень стосується інформаційної безпеки. Платформа працює з персональними та фінансовими даними, тому обґрунтовано використання зв'язки OAuth2/OpenID Connect, JWT, 2FA та AES-256. Стандартизовані протоколи OAuth2/OIDC забезпечують уніфіковані механізми автентифікації та авторизації для мобільних і веб-клієнтів, а токени доступу у форматі JWT дозволяють ефективно передавати підтверджені атрибути користувача та його ролі між компонентами системи. Двофакторна автентифікація (2FA) застосовується для доступу до адміністративних функцій і виконання чутливих операцій (зміна платіжних реквізитів, підтвердження транзакцій), що знижує ризики компрометації облікових записів. Для зберігання конфіденційної інформації застосовується симетричне шифрування з використанням алгоритму AES-256, а ключі шифрування розміщуються у спеціалізованих сервісах керування секретами, інтегрованих з інфраструктурою хмарного провайдера. Сукупність цих заходів формує багатошарову модель захисту, у межах якої компрометація одного з компонентів не призводить до повного порушення безпеки системи.

На основі обраного технологічного стеку сформовано функціональні та нефункціональні вимоги до платформи. Функціональні вимоги відображають основні бізнес-процеси та сценарії використання системи, тоді як нефункціональні визначають цільові характеристики якості – продуктивність, доступність, безпеку, масштабованість і зручність використання. Узагальнений перелік ключових функціональних вимог наведено в таблиці 2.1.

Таблиця 2.1

Функціональні вимоги до програмної платформи управління нерухомістю

Код	Вимога	Опис / критерій приймання
FR-01	Управління об'єктами	Створення, редагування, видалення та перегляд об'єктів і юнітів; фіксація історії змін.
FR-02	Управління договорами	Оформлення, пролонгація та завершення договорів оренди; автоматизований розрахунок орендної ставки.

Код	Вимога	Опис / критерій приймання
FR-03	Платежі та інвойси	Генерація інвойсів, підтримка онлайн-оплат, відображення статусів проведення платежів.
FR-04	Аналітика та звітність	Обчислення та візуалізація показників cashflow, occupancy, delinquency та інших KPI.
FR-05	Комунікації з користувачами	Обробка заявок, повідомлень і оголошень; фіксація історії звернень.
FR-06	Ролі та доступ	Підтримка ролей Tenant / Landlord / Admin і політик доступу до функціоналу та даних.

Вимоги FR-01 – FR-06 охоплюють критично важливі підсистеми: управління об’єктами та юнітами, ведення договорів оренди, автоматизований інвойсинг і облік платежів, модуль аналітики з KPI (cashflow, occupancy, delinquency тощо), підсистему комунікацій із заявками та оголошеннями, а також рольову модель доступу з розмежуванням прав між орендодавцем, орендарем і адміністратором. Виконання цих вимог гарантує, що платформа зможе підтримувати базові бізнес-процеси користувачів і слугуватиме надійною основою для подальшого розширення функціоналу.

Нефункціональні вимоги визначають експлуатаційні характеристики платформи та встановлюють цільові значення для її продуктивності, доступності та надійності. Узагальнений перелік основних нефункціональних вимог подано в таблиці 2.2.

Таблиця 2.2

Нефункціональні вимоги до програмної платформи

Код	Вимога	Міра / критерій
NFR-01	Продуктивність	Середній час обробки типових API-запитів < 200 мс; обов’язкова пагінація для великих вибірок.
NFR-02	Доступність	Цільовий рівень доступності не нижче 99,9 % за рахунок використання хмарної інфраструктури та резервування.
NFR-03	Безпека	Автентифікація на основі OAuth2/OIDC і JWT, підтримка 2FA, шифрування чутливих даних за допомогою AES-256, обов’язкове логування доступу та критичних дій.
NFR-04	Масштабо-ваність	Можливість горизонтального масштабування сервісів засобами контейнеризації та оркестрації (AWS ECS).

Код	Вимога	Міра / критерій
NFR-05	Юзабіліті	Реалізація інтерфейсів за принципом mobile-first із використанням сучасних UX-патернів для мобільних і десктопних пристроїв.
NFR-06	Сумісність	Підтримка інтеграцій зі Stripe, PayPal та іншими сервісами через офіційні SDK і REST-інтерфейси.

Вимога NFR-01 задає орієнтир щодо продуктивності: час обробки типових запитів API має бути меншим за 200 мс, а для роботи з великими обсягами даних обов’язково застосовується пагінація. NFR-02 визначає цільовий рівень доступності 99,9 %, який досягається завдяки використанню хмарної інфраструктури та механізмів резервування. NFR-03 фокусується на безпеці: автентифікація на основі JWT, підтримка двофакторної перевірки (2FA), шифрування чутливих даних за допомогою AES-256, а також обов’язкове логування доступу та критичних дій користувачів.

Вимога NFR-04 передбачає можливість горизонтального масштабування сервісів за допомогою контейнеризації та оркестрації (наприклад, через AWS ECS), що важливо для роботи з ростом навантаження. NFR-05 регламентує показники юзабіліті: інтерфейс має бути реалізований за принципом *mobile-first* із використанням сучасних UX-патернів, що забезпечує зручність роботи як на десктопах, так і на мобільних пристроях. Нарешті, NFR-06 визначає сумісність із зовнішніми сервісами та API – зокрема, інтеграцію зі Stripe/PayPal через їхні офіційні SDK та підтримку REST-підходу для подальшого розширення екосистеми.

Виконання нефункціональних вимог гарантує, що платформа буде не лише функціонально завершеною, а й придатною до експлуатації в умовах реального ринку оренди нерухомості, де критичними є стабільність доступу, швидкість реакції системи та захищеність даних. Таким чином, обраний технологічний стек та сформовані вимоги створюють цілісне підґрунтя для подальшого моделювання архітектури, структури бази даних і механізмів взаємодії компонентів системи.

2.2 Моделювання та проєктування архітектури системи

Архітектура програмної платформи управління нерухомістю проєктується як цілісна, але модульна система, у якій чітко розмежовано відповідальність між серверною та клієнтською частинами, а також між окремими доменами всередині кожної з них. На цьому етапі моделювання визначаються основні компоненти, їхні шари, канали взаємодії та типові послідовності виконання операцій, що надалі використовуються при побудові UML-діаграм, ER-моделі та специфікацій API.

Серверна частина реалізована як монолітний застосунок на платформі Node.js з модульною організацією коду. Монолітний підхід означає єдину кодову базу, один процес виконання (із можливістю кластеризації та масштабування в контейнерах) та спільну реляційну базу даних PostgreSQL для всіх підсистем. Водночас застосовано принципи доменно-орієнтованого проєктування: файлову структуру організовано за предметними областями (account, building, payments, club тощо), а логіка кожного домену ізольована у власних маршрутах, контролерах, сервісах і ORM-моделях.

Логічно бекенд поділено на кілька шарів: шар ініціалізації та конфігурації, транспортний шар, шар бізнес-логіки, шар доступу до даних, а також набір спільних компонентів і проміжного ПЗ (middleware). Така структура зображена на рисунку 2.2 і забезпечує слабке зв’язування між підсистемами при збереженні простоти розгортання, характерної для моноліту.

На шарі ініціалізації розташовуються стартові файли (наприклад, server.js, main_*.js) та каталог config, у якому зосереджені параметри підключення до PostgreSQL, налаштування середовищ (sandbox, production), змінні оточення, ключі шифрування та конфігурація JWT. У цих модулях відбувається запуск HTTP-сервера, встановлення з’єднання з базою даних, реєстрація cron-задач та інших інфраструктурних служб, але ще не виконується прикладна бізнес-логіка.

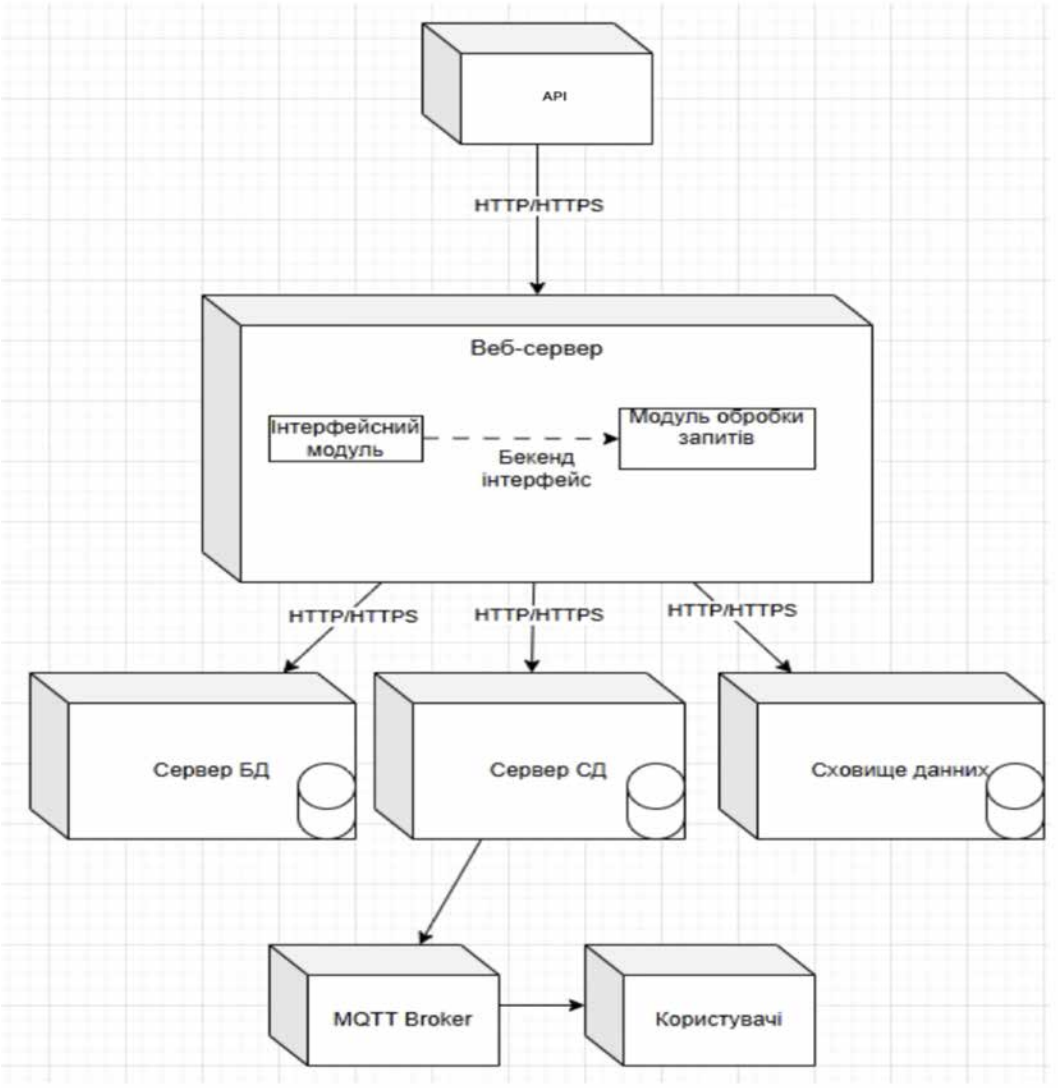


Рис. 2.2 Логічна архітектура бекенд-підсистеми платформи управління нерухомістю

Транспортний шар реалізується у вигляді маршрутів, згрупованих за доменами в каталозі lib/api/routes. Наприклад, гілка routes/account відповідає за всі операції з обліковими записами (реєстрація, логін, зміна профілю), а routes/building – за роботу з об’єктами нерухомості. Кожен маршрут визначає HTTP-метод, шлях, ланцюжок middleware (перевірка токена, обмеження частоти, логування, 2FA за потреби) та викликає відповідний контролер. При цьому транспортний шар не містить бізнес-правил: його завдання – прийняти запит, первинно його валідувати та передати управління на рівень бізнес-логіки.

Шар бізнес-логіки реалізується контролерами й сервісами, які оперують вже розібраними даними запиту. Контролери відповідають за координацію

операцій: звертаються до сервісів, агрегують результати, формують логічну відповідь, не залежачи від конкретних протоколів транспорту. Повторювана логіка (перевірка паролів, генерація токенів, робота з кошиками платежів, формування DTO) винесена у спеціалізовані сервіси.

Показовим є модуль автентифікації, де реалізується генерація та валідація токенів доступу. Для цього використовується бібліотека jose, яка забезпечує створення та перевірку JWT з асиметричним підписом. Типовий фрагмент коду, що імпортує необхідні засоби, наведено рис.2.3.

```
const {
  SignJWT,
  jwtVerify,
  compactVerify,
  errors,
  decodeProtectedHeader
} = require('jose');

const { ISSUER, JWKS, getSigningKey } = require('./keys');
const { issueRefreshToken } = require('./refresh_service');
```

Рис. 2.3 Фрагмент коду імпорту засобів роботи з JWT у модулі автентифікації

Функція генерації access-токена формує JWT із необхідними атрибутами (ідентифікатор користувача, email, роль, набір прав) та встановлює параметри безпеки – видавця, час випуску й закінчення дії, алгоритм підпису. Узагальнений вигляд такої функції подано на рис. 2.4.

Паралельно створюється refresh-токен, який зберігається в базі даних у хешованому вигляді разом із технічною інформацією (ідентифікатор облікового запису, IP-адреса, User-Agent, строк дії, ознаки відкликання). Відповідний сервіс може мати структуру, що представлено фрагментом коду в дод. А.1.

```
async function mintAccessToken(account) {
  const signingKey = await getSigningKey();

  return await new SignJWT({
      sub: String(account.id),
      email: account.email,
      role: account.role,
      roles_map: account.rolesMap
    })
    .setProtectedHeader({ alg: 'RS256', typ: 'JWT' })
    .setIssuer(ISSUER)
    .setIssuedAt()
    .setExpirationTime('15m')
    .sign(signingKey);
}
```

Рис. 2.4 Спрощена функція генерації access-токена

Контролер операції входу користувача (логіну) використовує описані сервіси та повертає клієнтові уніфіковану відповідь у стандартизованому форматі. Модуль response_structure відповідає за єдиний формат відповіді API (код, дані, помилка, додаткові метадані). Спрощений приклад коду такого контролера наведено в дод. А.2.

Доступ до захищених ресурсів реалізується через проміжне ПЗ, яке перевіряє заголовок Authorization, структуру та підпис JWT, а також правильність полів issuer, audience і строку дії токена. Типовий варіант коду такого middleware подано в дод. А.3.

Таким чином, шар бізнес-логіки та безпеки спирається на чітко відокремлені компоненти: контролери відповідають за сценарій обробки запиту, сервіси – за реалізацію доменних правил, а middleware – за попередні перевірки й перехоплення помилок.

Шар доступу до даних представлений ORM-моделями в каталозі struct_model, які відображають структуру таблиць у PostgreSQL. Для модуля автентифікації, зокрема, визначено модель користувача та модель токенів оновлення. Спрощений опис останньої наведено в дод. А.4.

Додатково в архітектурі бекенду передбачено спеціалізовані компоненти для логування (наприклад, надсилання критичних повідомлень у Telegram-канал), а також обробники Socket.IO/WebSocket подій, що забезпечують миттєві оновлення інтерфейсу (зміна статусу заявок, сповіщення про нові платежі тощо).

Основні домени серверної частини та їх призначення узагальнено в табл. 2.3.

Таблиця 2.3

Основні домени бекенд-підсистеми платформи

Домен	Основне призначення
Облікові записи (Account)	Реєстрація, автентифікація, керування профілем, ролями та двофакторною перевіркою.
Об’єкти нерухомості	Ведення переліку будинків і юнітів, прив’язка до користувачів і договорів.
Договори оренди	Створення, пролонгація та завершення договорів, зберігання умов оренди.
Платежі та інвойси	Генерація інвойсів, фіксація платежів, взаємодія з платіжними провайдерами.
Аналітика	Обчислення та збереження показників cashflow, occupancy, delinquency та інших KPI.
Комунікації	Обробка заявок, повідомлень, оголошень, історія звернень користувачів.

На прикладі домену автентифікації можна проілюструвати типову end-to-end-послідовність обробки запиту. Під час виклику операції входу користувача HTTP-запит надходить до транспортного шару, де проходить стандартні middleware-перевірки. Далі відповідний контролер звертається до сервісу автентифікації, який перевіряє облікові дані за даними бази, формує короткоживучий access-токен у форматі JWT, створює та зберігає refresh-токен із прив’язкою до облікового запису та середовища (IP-адреса, user-agent), а потім повертає уніфіковану відповідь. Аналогічна структура сценаріїв застосовується в інших доменах, що забезпечує єдність підходів і передбачуваність поведінки всієї системи.

Клієнтська частина платформи реалізована як кросплатформний Flutter-застосунок із модульною структурою та чітким розділенням на шари

представлення, керування станом і сервісів. Код організовано за принципом «feature-first»: для кожної функціональної області (обліковий запис, об’єкти, платежі, сповіщення, аналітика) створено окремий модуль, який містить екрани, контролери, моделі та сервіси.

У шарі представлення зосереджені екрани та віджети, що відповідають за візуалізацію стану та реакцію на дії користувача. Вони не виконують мережевих запитів і не взаємодіють безпосередньо з бекендом. Натомість екрани працюють із контролерами стану, реалізованими за допомогою зв’язки GetX і Provider. Контролери інкапсулюють бізнес-логіку модуля, зберігають поточний стан (списки, фільтри, прапорці завантаження), викликають сервіси для роботи з API та повідомляють UI про зміни. Приклад типового контролера екрана входу наведено в дод. Б.1. UI-екран при цьому зосереджується лише на відображенні поточного стану контролера, як показано в дод. Б.2 .

Сервісний шар Flutter-застосунку інкапсулює роботу з REST-API бекенду: формування HTTP-запитів, додавання JWT до заголовків, оброблення помилок та перетворення відповідей у типізовані моделі. Спрощений приклад сервісу автентифікації подано в дод. Б.3.

Моделі даних у клієнтській частині відтворюють структуру об’єктів, із якими працює сервер: користувачі, будинки, юніти, договори, транзакції, повідомлення, елементи аналітики тощо. Для кожної моделі реалізуються методи перетворення з/до формату JSON, що забезпечує узгодженість із контрактами REST-API. Такий підхід спрощує валідацію даних, зменшує кількість помилок під час еволюції серверної частини та підвищує прозорість коду.

Особливу увагу приділено підтримці різних платформ. Flutter-застосунок використовує спільні контролери, сервіси та моделі як для мобільного, так і для веб-інтерфейсу, тоді як відмінності полягають переважно в побудові навігації та компоновці екранів. Це дає змогу реалізувати адаптований UX для смартфонів і браузера без дублювання бізнес-логіки, що відповідає вимогам до кросплатформності та знижує витрати на супровід.

Узгоджене моделювання архітектури серверної та клієнтської частин забезпечує цілісність програмної платформи: монолітний, але модульний бекенд гарантує транзакційну цілісність і єдину точку керування бізнес-правилами, тоді як структурований кросплатформний клієнтський застосунок забезпечує зручний доступ користувачів до функцій системи. У результаті сформована архітектура задовольняє функціональні та нефункціональні вимоги, визначені на попередньому етапі, і створює надійну основу для подальшого масштабування та еволюції платформи.

2.3 Формалізація даних та проєктування структури бази даних

Проєктування інформаційної моделі платформи ґрунтується на принципах нормалізованого реляційного подання даних та забезпеченні цілісності бізнес-процесів. Основою є СКБД PostgreSQL, що підтримує ACID-транзакційність і надає гнучкі механізми роботи як із суворо структурованими, так і з напівструктурованими даними (JSONB). Структура бази даних охоплює декілька груп сутностей: користувачі, об’єкти нерухомості та їх юніти, договори оренди, фінансові транзакції, історія грошових потоків, заявки та комунікації. Узагальнений взаємозв’язок між основними сутностями наведено на ER-діаграмі (дод. В.1), яка демонструє ключові зв’язки «один-до-багатьох» та «багато-до-одного» між доменними таблицями.

Для подальшого опису магістерської роботи доцільно детальніше розглянути базові доменні таблиці, що безпосередньо забезпечують реалізацію основних бізнес-процесів платформи: Users, Properties, Units, Leases, Payments та WalletHistory. Саме ці сутності лежать в основі моделі користувачів, об’єктів, договорів, платіжного циклу та обліку змін балансу.

Таблиця 2.4 **Users** описує всіх учасників платформи (орендодавців, орендарів, адміністраторів) і задає базову модель користувача, яка є відправною точкою для більшості бізнес-процесів системи.

Таблиця 2.4

Структура таблиці Users

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Унікальний ідентифікатор користувача
name	text	NOT NULL	ПІБ користувача
email	text	UNIQUE, NOT NULL	Логін / контактна адреса
role	enum(tenant, landlord, admin)	NOT NULL	Роль користувача в системі
password_hash	text	NOT NULL	Хеш пароля
twofa_enabled	boolean	DEFAULT false	Ознака увімкнення двофакторної автентифікації
created_at	timestamptz	DEFAULT now()	Дата та час створення запису

Через зовнішні ключі на цю таблицю посилаються модулі оренди, комунікацій, платежів та історії грошових операцій. Наступним логічним кроком формалізації даних є опис сутностей, що відображають матеріальні активи платформи – об’єкти нерухомості, якими управляють користувачі (табл. 2.5). Саме вони формують верхній рівень ієрархії домену «нерухомість» та визначають структуру подальших залежних таблиць.

Таблиця 2.5

Структура таблиці Properties

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Ідентифікатор об’єкта нерухомості
owner_id	UUID	FK → Users(id)	Власник об’єкта (посилання на користувача)
address	text	NOT NULL	Поштова адреса об’єкта
units	int	CHECK ≥ 0	Загальна кількість юнітів
created_at	timestamptz		Дата реєстрації об’єкта в системі

Таблиця **Properties** (табл. 2.5) описує об’єкти нерухомості як цілісні адміністративні одиниці, однак для коректного ведення обліку та побудови договорів цього рівня деталізації недостатньо. Кожен об’єкт складається з окремих юнітів – квартир, офісних приміщень або інших функціональних блоків,

що можуть передаватися в оренду. Тому наступним елементом інформаційної моделі є таблиця **Units**, яка конкретизує внутрішню структуру кожного об'єкта (табл. 2.6).

Таблиця 2.6

Структура таблиці Units

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Ідентифікатор юніта
property_id	UUID	FK → Properties(id)	Належність до об'єкта
unit_no	text	NOT NULL	Номер / індекс юніта
status	enum(vacant, occupied)	NOT NULL	Статус юніта (вільний / зайнятий)

Таблиця 2.6 **Units** деталізує внутрішню структуру об'єктів (квартири, офіси, паркомісця тощо) та використовується для побудови договорів оренди. Визначення складу юнітів у таблиці **Units** (табл. 2.6) створює підґрунтя для формування орендних правовідносин. Однак сам факт існування юніта не означає його передачу в користування, тому подальше моделювання стосується опису параметрів орендних договорів. Таблиця **Leases** фіксує юридичні та фінансові характеристики оренди, пов'язуючи конкретний юніт із відповідним орендарем (табл. 2.7).

Таблиця 2.7

Структура таблиці Leases

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Ідентифікатор договору оренди
unit_id	UUID	FK → Units(id)	Юніт, до якого прив'язаний договір
tenant_id	UUID	FK → Users(id)	Орендар (користувач системи)
start_date	date	NOT NULL	Дата початку дії договору
end_date	date		Дата завершення (може бути NULL для відкритих договорів)
monthly_rent	numeric(12,2)	NOT NULL	Місячна орендна ставка
deposit	numeric(12,2)		Сума завдатку / гарантійного платежу

Таблиця 2.7 **Leases** фіксує параметри договорів і є основою для генерації інвойсів і розрахунку регулярних платежів. Після формування договору оренди у таблиці **Leases** (табл. 2.7) виникає необхідність у відображенні реальних грошових операцій, які здійснюються протягом строку його дії. Саме ці транзакції становлять основу фінансового циклу платформи. Наступний елемент моделі – таблиця **Payments**, що містить інформацію про всі платежі за договором, їх статус, суму та канал проведення (табл. 2.8).

Таблиця 2.8

Структура таблиці Payments

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Ідентифікатор платіжної операції
lease_id	UUID	FK → Leases(id)	Договір, у межах якого здійснено платіж
amount	numeric(12,2)	NOT NULL	Сума платежу
status	enum(paid, pending, overdue)	NOT NULL	Статус платежу (оплачено / очікує / прострочено)
date	date	NOT NULL	Дата фактичної операції або нарахування
method	enum(stripe, paypal, cash)		Канал оплати (Stripe, PayPal, готівка тощо)
invoice_no	text	UNIQUE	Унікальний номер інвойсу

Таблиця 2.8 **Payments** описує всі фінансові операції, пов’язані з договорами, та використовується для формування звітності й розрахунку KPI (delinquency, cashflow тощо). Фіксація платіжних операцій у таблиці **Payments** (табл. 2.8) забезпечує облік виконання фінансових зобов’язань за окремими договорами.

Проте для повної прозорості грошових потоків цього рівня деталізації недостатньо, оскільки платформа підтримує також внутрішні взаєморозрахунки, корекції балансу, повернення коштів та інші технічні операції. Ці зміни повинні реєструватися незалежно від конкретного договору оренди. Саме тому в структурі бази даних виділено таблицю **WalletHistory**, яка фіксує історію всіх

debit і credit операцій для кожного користувача й забезпечує можливість повного аудиту транзакційної активності (табл. 2.9).

Таблиця 2.9

Структура таблиці WalletHistory

Поле	Тип	Ключ / Умова	Опис
id	UUID	PK	Ідентифікатор запису історії
user_id	UUID	FK → Users(id)	Користувач, до якого належить операція
amount	numeric(12,2)	NOT NULL	Величина зміни балансу
type	enum(debit, credit)	NOT NULL	Тип операції (списання / зарахування)
created_at	timestamptz	DEFAULT now()	Дата та час фіксації операції

Таблиця 2.9 **WalletHistory** забезпечує прозорий облік усіх змін балансу користувачів (нарахування, списання, повернення коштів) і є основою для фінансового аудиту.

Проведена формалізація даних дозволила сформувати цілісну інформаційну модель платформи, у якій кожна сутність точно відображає відповідний елемент предметної області та узгоджена з бізнес-процесами системи. Побудована структура бази даних спирається на нормалізовану реляційну схему PostgreSQL, що забезпечує цілісність зв’язків між користувачами, об’єктами нерухомості, юнітами, договорами оренди та фінансовими транзакціями.

Послідовне проєктування від загального рівня (користувачі та об’єкти) до більш деталізованих елементів (юніти, договори, платежі, історія змін балансу) дало змогу створити гнучку та масштабовану модель, придатну для розширення й подальшої інтеграції з аналітичними та сервісними модулями платформи. Такий підхід забезпечує узгодженість даних, підтримує вимоги безпеки та дозволяє ефективно реалізувати ключові функції системи управління нерухомістю в умовах реального експлуатаційного навантаження.

2.4. Розроблення ER-діаграми бази даних

Інформаційна модель платформи розроблена на основі реляційної схеми, реалізованої у СКБД PostgreSQL. Враховуючи багатокомпонентну структуру системи управління нерухомістю – облікові записи, фінансові операції, сервісні заявки, комунікації, соціальну взаємодію та аналітичні модулі – повна база даних включає понад сто таблиць, пов'язаних між собою численними зв'язками типу «один-до-багатьох», «багато-до-одного» та «багато-до-багатьох». Така модель забезпечує високу деталізацію даних і дозволяє масштабувати функціонал платформи без порушення її цілісності.

Оскільки повна ER-діаграма бази даних є значною за обсягом і містить багато службових сутностей, у даному підрозділі зосереджено увагу на її узагальненому фрагменті, що відображає ключові домени системи. Вибрані сутності охоплюють найважливіші бізнес-процеси платформи, а саме: управління обліковими записами, фінансами, сервісними заявками, комунікаціями та соціальною активністю користувачів. Узагальнений фрагмент ER-моделі наведено на рис. 2.5.

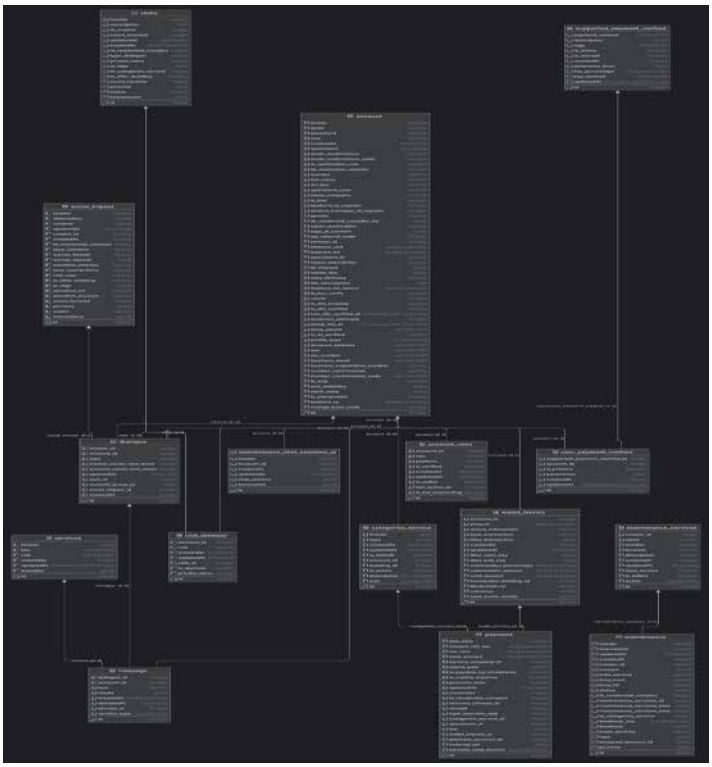


Рис. 2.5 Фрагмент ER-діаграми основних доменів бази даних

У представлений фрагмент входять такі групи сутностей: account, account_roles, wallet_sett, wallet_history, payment, user_payment_method, maintenance, maintenance_services, maintenance_ai_analysis, maintenance_chat_sessions_ai, maintenance_chat_mess_ai, dialogue, message, clubs, club_member, private_post.

Подальший опис подано за доменним принципом, що відповідає структурі логічної моделі.

Сутність account є центральним вузлом інформаційної моделі та акумулює відомості про всі категорії користувачів платформи, включаючи мешканців, власників, адміністраторів, представників керуючих компаній та зовнішніх підрядників. Структура таблиці охоплює кілька логічних груп полів, що забезпечують повноцінний облік ідентифікаційних даних, параметрів аутентифікації, фінансових характеристик і соціальної взаємодії. До ідентифікаційно-контактного блоку входять атрибути id, email, phone_number, full_name, account_address, business_email та реєстраційні дані користувача. Підсистема безпеки реалізована через такі поля, як password_hash, email_confirmation_code, is_2fa_enabled, is_2fa_verified, last_2fa_verified_at, incorrect_attempts, block_2fa_at та change_password_code; вони підтримують валідацію облікового запису, контроль активності та механізми двофакторної автентифікації.

Фінансовий профіль користувача визначається атрибутами balance_usd, balance_bb, balance_rg, balance_award, yardi_meta та unit_statistics, що також дозволяють зберігати агреговані дані у форматі JSON. Рольовий статус описується полями role, roleId, is_verification_role, is_kyc_verified, profile_type, is_placeholder, is_trial та is_ai_verified. Прив’язка до об’єктів нерухомості реалізована через id_residential_complex, id_residential_complex_list, apartment_id і apartment_num, тоді як соціально-медійні відомості зосереджені у полях avatar, bio_description, media_link, tags_ai_content та id_interest. Через велику кількість залежних таблиць сутність account має кардинальність зв’язку 1:N з більшістю

доменів, зокрема з ролями, платежами, історією гаманця, діалогами та соціальними активностями.

Похідною від account є сутність account_roles, що реалізує розширену рольову модель платформи. Вона дозволяє одному користувачу мати декілька ролей залежно від контексту – наприклад, одночасно бути мешканцем, модератором будинку та адміністратором сервісних заявок. Структура таблиці включає поля account_id, role, platform, is_verified, is_online, last_active_at та is_full_onboarding. Зв'язок account – account_roles має тип 1:N, що забезпечує масштабованість та гнучкість у керуванні правами доступу.

До фінансового домену належать таблиці payment, wallet_history, wallet_sett і user_payment_method. Сутність payment відображає інформацію про нарахування та фактичні платежі. Її структура охоплює параметри due_date, payment_date, amount_net_tax, tax_rate, total_amount, service_company_id, id_residential_complex, apartment_id і categories_service_id, а також дані про стан виконання платежу, зовнішні посилання на виконані транзакції та суми часткових оплат. Ця таблиця формує основу для фінансової аналітики та розрахунків.

Сутність wallet_history описує історію змін балансу користувача та відображає реальні грошові рухи, незалежно від того, чи пов'язані вони з конкретним договором. До її основних атрибутів належать account_id, amount, status_transaction, type_transaction, commission_percentage, total_amount, currency, blockchain_tx і data_transaction у форматі JSON. Завдяки цьому таблиця підтримує повноцінний аудит фінансових операцій і синхронізацію з зовнішніми платіжними сервісами. Таблиця wallet_sett містить конфігураційні параметри гаманця, у яких визначаються правила обміну внутрішніх одиниць, комісії, мінімальні ліміти та курси конвертації, що дозволяє оновлювати політику розрахунків без модифікації програмного коду. Сутність user_payment_method описує платіжні методи конкретного користувача – від прив'язки банківської картки до інтеграції зі сторонніми платіжними системами – і включає поля

supported_payment_method_id, account_id, is_primary та parameters у форматі JSON.

Домен сервісних заявок та аналітики охоплює кілька ключових сутностей. Таблиця maintenance відображає сервісні звернення мешканців і містить дані про назву проблеми, опис ситуації, автора звернення, запланований час візиту, статус виконання, виконавця, прив’язку до об’єкта та оцінку якості обслуговування. Сутність maintenance_services функціонує як довідник сервісних компаній та включає назву, контактні дані, опис послуг і тип сервісу. Для поглибленого аналізу стану об’єктів використовується таблиця maintenance_ai_analysis, у якій зберігаються прогнозні показники, оцінки стану, типові проблеми, рекомендації та інші агреговані результати, сформовані алгоритмами аналізу.

Підтримку взаємодії користувача з AI-асистентом забезпечують сутності maintenance_chat_sessions_ai та maintenance_chat_mess_ai. Перша з них містить інформацію про окрему чат-сесію, включаючи історію повідомлень і стислий підсумок, а друга – зберігає окремі повідомлення, медіавкладення та часові мітки. Разом вони формують архітектуру інтелектуальної підтримки рішень у межах сервісного домену.

Окремий домен моделює комунікації та соціальну активність користувачів. Сутність dialogue описує канал спілкування між учасниками платформи та містить дані про тип діалогу, статуси прочитання повідомлень і можливу прив’язку до клубів. Сутність message забезпечує зберігання повідомлень у межах діалогу та включає інформацію про автора, текст, медіадані та службові посилання на інші сутності. Соціальний компонент платформи моделюється таблицею clubs, у якій зберігаються тематичні спільноти мешканців, і таблицею club_member, що формує зв’язок «багато-до-багатьох» між користувачами та клубами. Сутність private_post доповнює модель можливістю публікації локальних повідомлень, прив’язаних до конкретного житлового комплексу, зберігаючи текст опису, авторство, медіадані та статистику взаємодій.

Узагальнена характеристика основних сутностей наведено в таблиці дод. Д.1.

Побудована ER-діаграма відображає предметну область у структурованому вигляді та забезпечує логічну узгодженість усіх компонентів бази даних. Її архітектура формує чіткий поділ системи на кілька доменів – обліковий, фінансовий, сервісний та соціальний, – кожен із яких репрезентує окрему групу бізнес-процесів. Таке розмежування дозволяє підтримувати організованість моделі й уникати надмірної залежності між сутностями різних підсистем.

Усередині моделі збережено високий рівень цілісності даних, що досягається застосуванням зовнішніх ключів, обмежень цілісності та коректно визначених типів зв’язків між таблицями. Завдяки цьому система гарантує узгодженість інформації як у межах окремих доменів, так і на перетині їх функціональних контекстів. Додатковою перевагою є гнучкість моделі, що надає можливість розширювати функціонал платформи без зміни ядра структури: нові довідники, модулі або аналітичні підсистеми можуть інтегруватися у вже сформовану архітектуру без порушення її логічної цілісності. Це досягається також підтримкою змішаних структур даних, зокрема JSONB-полів, які використовуються для зберігання складних метаданих, агрегованої статистики та аналітичних результатів.

У межах основного тексту роботи представлено фрагмент ER-моделі (рис. 2.5), що відображає ключові зв’язки між доменами й демонструє основні принципи побудови інформаційної структури. Повна ER-діаграма, яка містить довідникові таблиці, історичні журнали та службові структури, винесена до Додатку В. Вона може використовуватися як детальна технічна основа для подальшого розширення функціональності, удосконалення бізнес-процесів та супроводу платформи в процесі експлуатації.

2.5. UML-моделювання системи

Для забезпечення формального опису структури та поведінки програмної платформи застосовано мову UML (Unified Modeling Language), яка дає змогу відтворити ключові аспекти функціонування системи на різних рівнях абстракції. Моделювання здійснювалося послідовно — від представлення взаємодії користувачів із платформою до деталізації внутрішніх зв’язків між сутностями та опису динамічних сценаріїв роботи. Комплекс діаграм, що охоплює варіанти використання, класи, компоненти та послідовності, забезпечує цілісне уявлення про архітектуру та логіку системи. У цьому підрозділі подано фрагменти UML-моделей, які демонструють базові механізми платформи; повний набір діаграм наведено в додатках.

Початковим етапом моделювання стала побудова діаграми варіантів використання, що відображає основні сценарії взаємодії між категоріями користувачів та системою управління нерухомістю. На діаграмі (рис. 2.6) показано ключових акторів: мешканця, менеджера житлового комплексу, сервісного підрядника, адміністратора платформи, а також зовнішні сервіси — платіжного провайдера та службу сповіщень. Діаграма демонструє інтегрованість платформи з зовнішніми каналами платежів та системами доставки push-, email- і SMS-повідомлень.

У межах цього моделювання мешканець виконує такі дії, як автентифікація, перегляд рахунків, оплата послуг, керування гаманцем, створення сервісних заявок, участь у клубах та обмін повідомленнями. Менеджер комплексу опрацьовує заявки, контролює статуси, взаємодіє з мешканцями через діалоги та аналізує статистику звернень. Адміністратор керує користувачами, ролями та довідниками, а сервісна компанія отримує та виконує призначені їй заявки. Подібне структурування варіантів використання дає можливість оцінити зовнішню поведінку системи та її відповідність вимогам експлуатації.

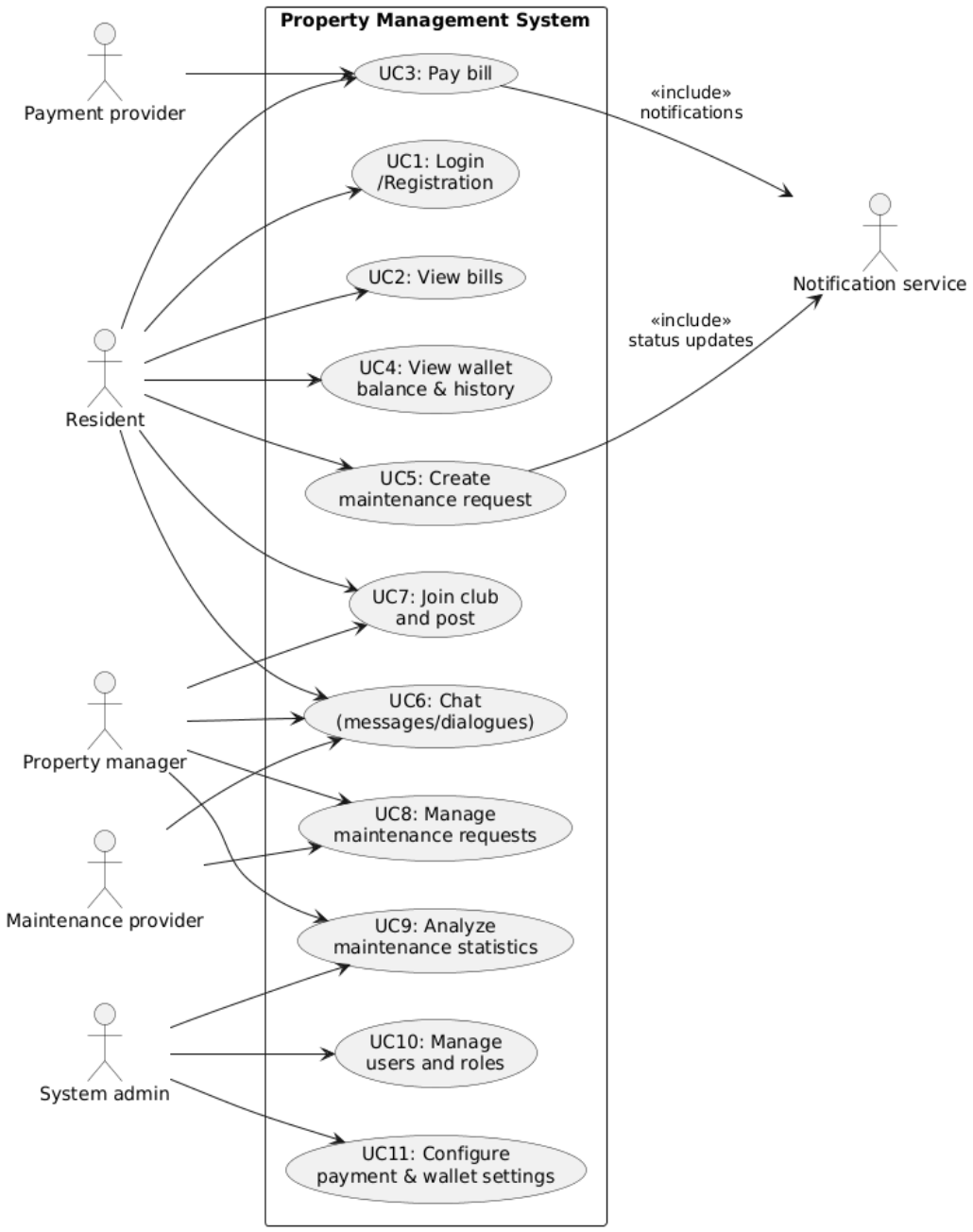


Рис. 2.6 Загальна діаграма варіантів використання системи

На основі структури бази даних та внутрішньої логіки бекенд-модуля побудовано діаграму класів, яка відображає доменну модель системи (рис. 2.7). Ця діаграма демонструє ключові сутності платформи, їх атрибути та асоціації. Клас Account описує облікові записи користувачів, включаючи контактні дані, фінансові атрибути, налаштування безпеки та методи автентифікації й керування профілем. Клас AccountRole забезпечує зв’язок одного користувача з кількома ролями, що підтримує контекстну модель доступу.

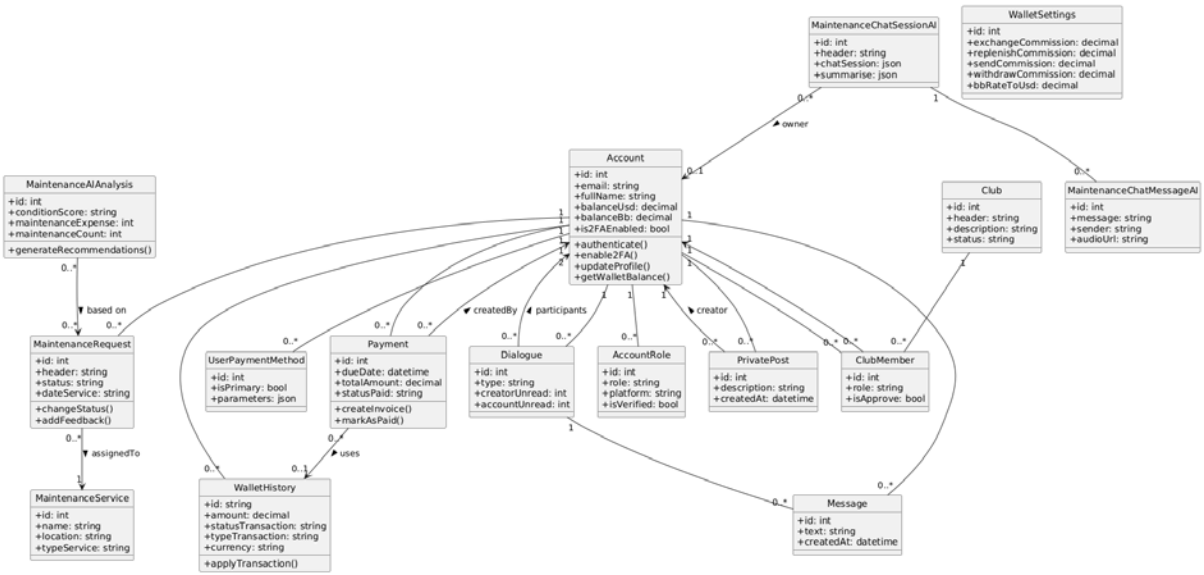


Рис. 2.7 Діаграма класів основних сутностей системи

Фінансовий модуль представлено класами Payment, WalletHistory, WalletSettings та UserPaymentMethod. Payment відображає операції, виконані у межах платіжного циклу, а WalletHistory фіксує зміни балансу користувача. Налаштування комісій і курсів конвертації зосереджено у WalletSettings, а UserPaymentMethod моделює прив’язані платіжні способи.

Домен сервісних звернень охоплює класи MaintenanceRequest, MaintenanceService, а також MaintenanceAIAnalysis та об’єкти AI-чатів, що забезпечують аналітику та напівавтоматизовану підтримку користувачів. Окремим блоком представлені класи Dialogue і Message, які моделюють внутрішню комунікацію між користувачами, а також Club, ClubMember та PrivatePost, що забезпечують соціальний функціонал системи. Діаграма містить різні типи зв’язків — асоціації 1:N, M:N та агрегації, що відтворюють реальну логіку взаємодії сутностей.

Для опису інфраструктурної архітектури створено діаграму компонентів і розгортання (рис. 2.8). Вона демонструє взаємодію клієнтських застосунків (Flutter mobile та Flutter web) із серверною частиною, яка реалізована у вигляді монолітного Node.js-застосунку з модульною організацією.

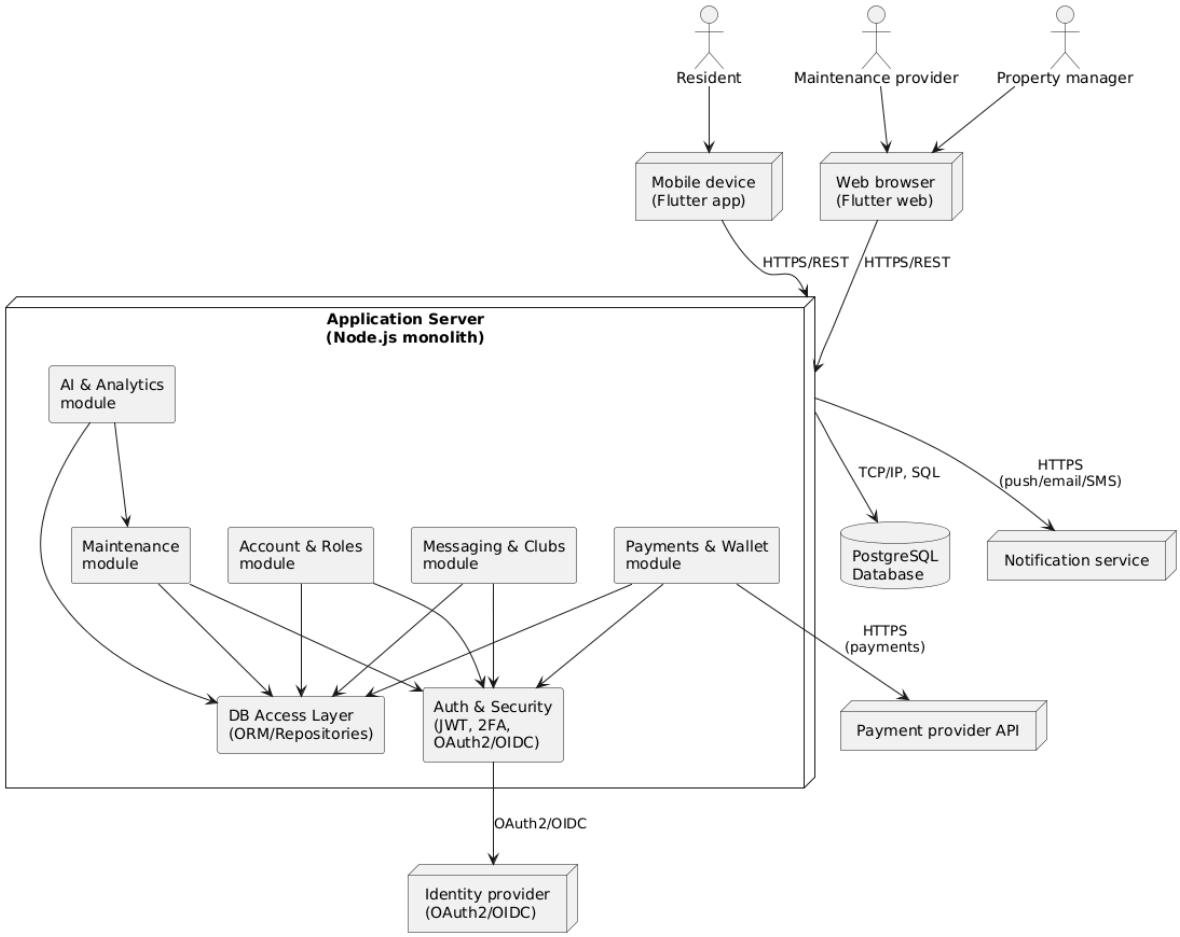


Рис. 2.8 Діаграма компонентів і розгортання системи

У структурі серверного вузла виділено модулі автентифікації та безпеки, управління користувачами, платіжний модуль, підсистему сервісних заявок, модуль обміну повідомленнями та соціальних клубів, а також модуль аналітики та AI-обробки. Доступ до бази даних PostgreSQL здійснюється через шар ORM-моделей, що забезпечує відокремлення логіки доступу від бізнес-логіки.

На діаграмі також відображено інтеграцію із зовнішніми сервісами: платіжним провайдером, системою сповіщень та провайдером ідентифікації (OAuth2/OIDC). Така архітектура демонструє повну схему розгортання та мережевих взаємодій системи.

Динаміка взаємодії між компонентами системи проілюстрована за допомогою діаграм послідовності. Вони відтворюють покроковий обмін повідомленнями між клієнтським застосунком, бекенд-сервером, базою даних та зовнішніми сервісами.

Перший сценарій – «Оплата рахунку» – показує, яким чином серверна частина перевіряє права доступу користувача, отримує налаштування гаманця, формує запит до платіжного провайдера, опрацьовує результат транзакції та оновлює стан платежу і балансу користувача у базі даних. Діаграма (рис. 2.9) узгоджено відображає логіку зміни станів та забезпечує прозорість бізнес-процесу.

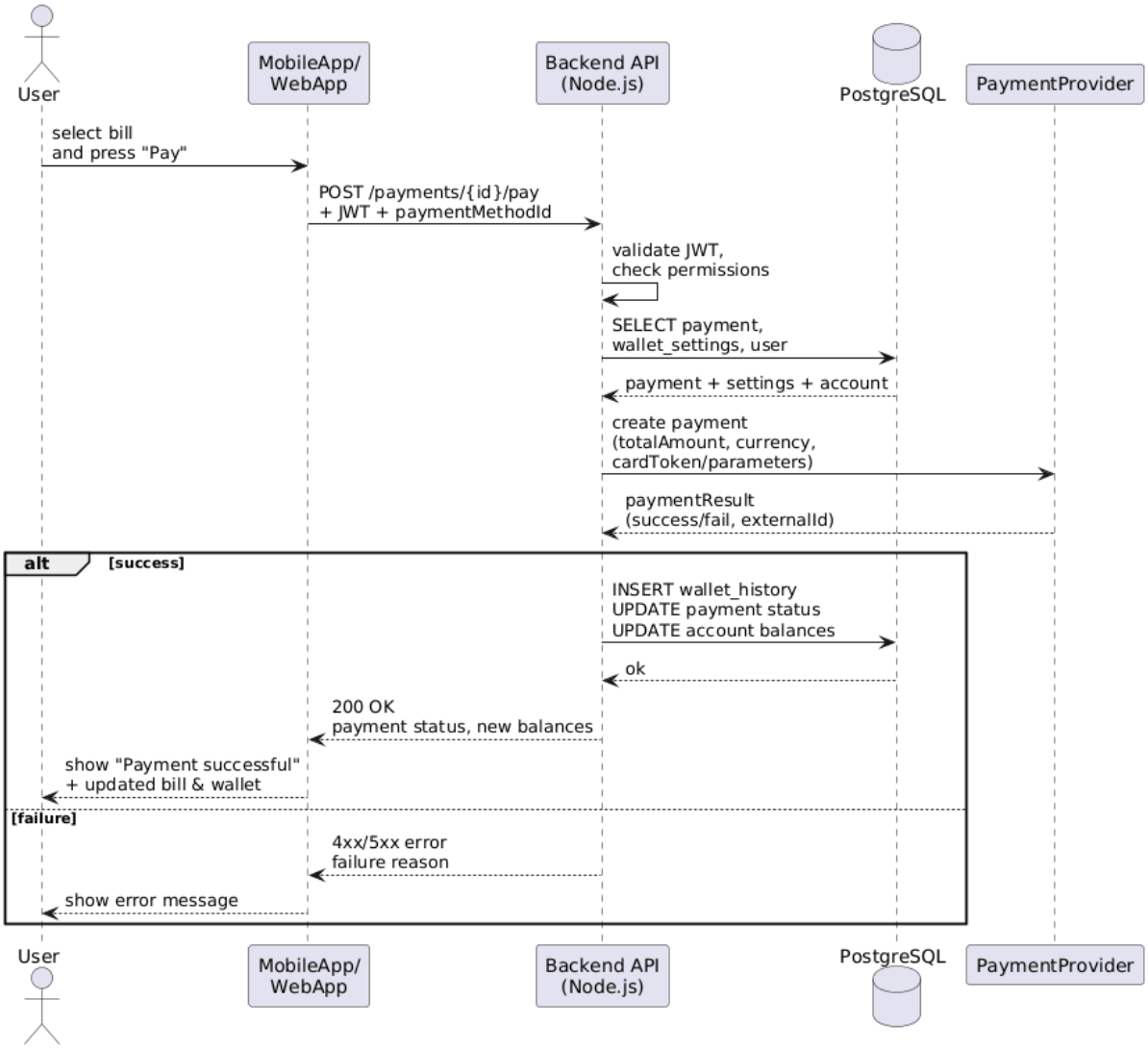


Рис. 2.9 Діаграма послідовності сценарію «Оплата рахунку»

Другий сценарій – «Створення сервісної заявки та її обробка» –демонструє повний життєвий цикл звернення: подання заявки мешканцем, її реєстрацію на сервері, подальшу обробку менеджером, обмін повідомленнями між учасниками та завершення процесу із залишенням відгуку (рис. 2.10). Така візуалізація дає

змогу чітко встановити розмежування відповідальності між компонентами та підтвердити внутрішню узгодженість системи.

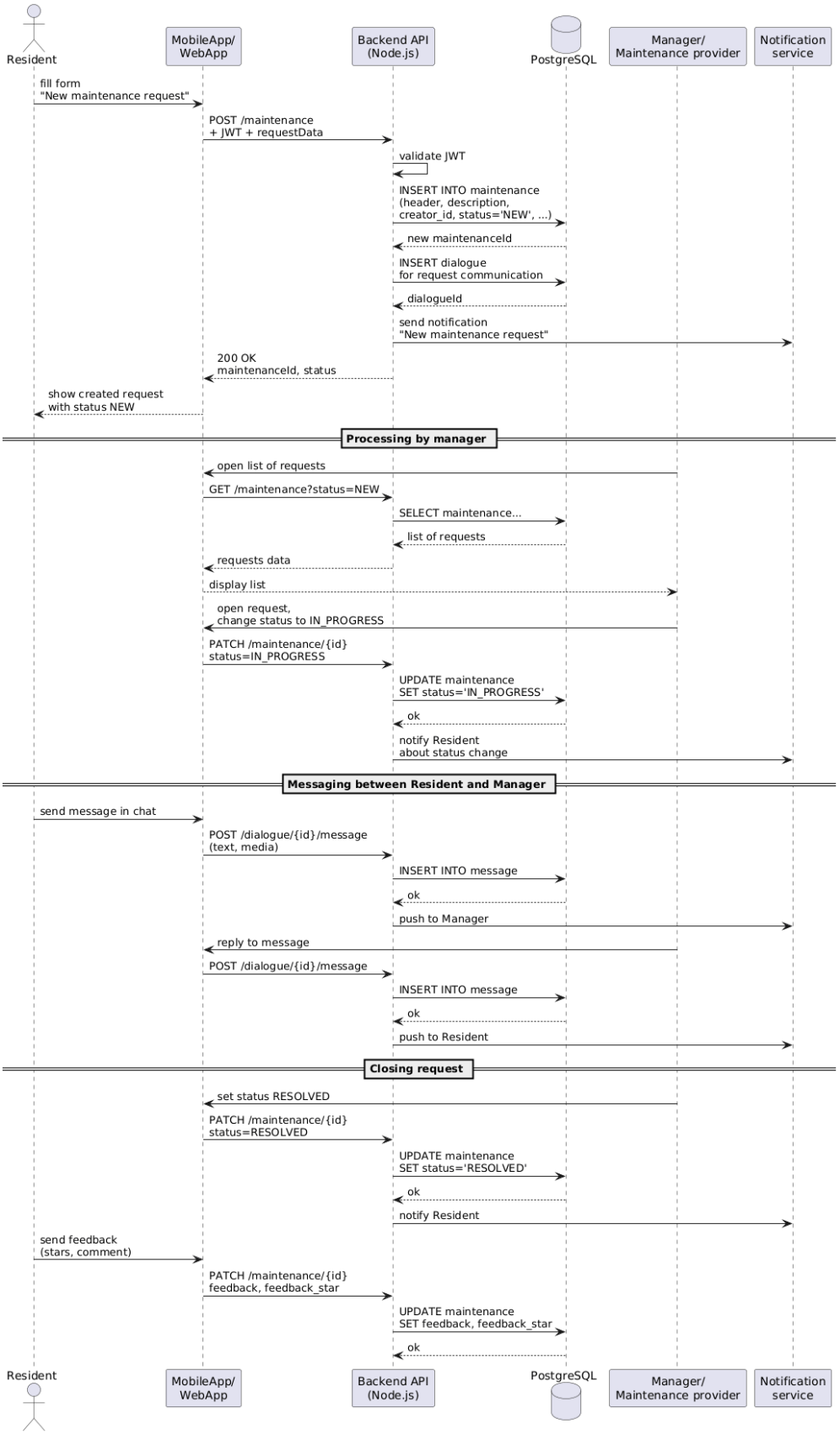


Рис. 2.10 Діаграма послідовності сценарію «Обробка сервісної заявки»

Отриманий набір UML-моделей забезпечує всебічне відтворення функціональної та технічної структури системи. Діаграми варіантів використання дозволяють сформулювати вимоги з позиції кінцевих користувачів; діаграма класів відображає логіку предметної області; діаграма компонентів демонструє архітектуру взаємодії модулів; діаграми послідовності деталізують поведінку системи у ключових сценаріях. Сукупність моделей є важливим інструментом комунікації між розробниками та аналітиками й виступає підґрунтям для подальшої еволюції платформи та розширення її функціональних можливостей.

2.6. Опис REST-API для взаємодії клієнт–сервер

Клієнтська частина платформи, реалізована як мобільний та веб-застосунок на Flutter, взаємодіє з сервером через REST-API поверх протоколу HTTPS. Обмін даними здійснюється у форматі JSON, сесії є безстановими, а автентифікація виконується за допомогою JWT-токенів, що передаються в заголовку Authorization: Bearer <token>. Такий підхід забезпечує чітке розділення відповідальностей між клієнтом і сервером, спрощує масштабування та дозволяє використовувати однакові API-контракти для різних типів клієнтів.

REST-інтерфейс побудовано як сукупність маршрутів, згрупованих за доменними областями. Для роботи з обліковими записами використовуються шляхи з префіксом /account, для квартир та житлових комплексів – /apartments, для сервісних заявок – /maintenance, для фінансових операцій – /wallet і /payment, для соціальної взаємодії – /clubs та /dialog, для процедур автентифікації – /auth. Такий модульний поділ відповідає внутрішній архітектурі бекенд-застосунку та полегшує подальшу еволюцію системи.

В основі проєктування REST-API лежить узгоджене використання HTTP-методів. Запити на читання інформації подаються методом GET, створення нових ресурсів або запуск операцій, що змінюють стан системи, реалізується через POST, часткове оновлення окремих атрибутів виконується за допомогою PATCH, а деактивація або логічне видалення ресурсів – за допомогою DELETE.

Узагальнений розподіл методів за ролями наведено в таблиці 2.10, яка відображає зв’язок між семантикою HTTP-методів та типовими сценаріями платформи.

Таблиця 2.10

Використання HTTP-методів у REST-API платформи

Метод	Основне призначення	Приклад ресурсу
GET	отримання даних без зміни стану системи	перегляд профілю користувача, списку заявок
POST	створення ресурсу або запуск прикладної операції	створення заявки, ініціація платежу
PATCH	часткове оновлення атрибутів існуючого ресурсу	зміна статусу заявки, редагування профілю
DELETE	деактивація або логічне видалення ресурсу	відв’язування платіжного методу, вихід з клубу

Для всіх маршрутів застосовується уніфікована система кодів стану HTTP. У разі коректного виконання запиту повертаються коди 200 (успішне читання) або 201 (створення нового ресурсу), при помилках валідації використовується код 400, при відсутності або некоректності токена – 401, при недостатніх правах – 403, при зверненні до неіснуючого ресурсу – 404, при бізнес-конфліктах на кшталт спроби повторно оплатити рахунок – 422, а при внутрішніх збоях серверу – 500. Зведення типових кодів наведено в таблиці 2.11.

Таблиця 2.11

Стандартизовані коди відповідей REST-API

Код	Опис стану	Типова ситуація
200	успішне виконання	повернення профілю, списку об’єктів
201	ресурс створено	реєстрація акаунта, створення заявки
400	некоректні вхідні дані	помилки валідації форм
401	відсутня або недійсна автентифікація	прострочений чи пошкоджений JWT-токен
403	заборонено	спроба доступу до чужих даних
404	ресурс не знайдено	невірний ідентифікатор у шляху
422	бізнес-помилка	повторна оплата вже сплаченого рахунку
500	внутрішня помилка сервера	непередбачена ситуація на боці бекенду

Усі відповіді REST-API мають стандартизовану структуру, що полегшує обробку результатів на стороні клієнта. На рівні бекенду використовується єдиний модуль формування відповіді, який повертає JSON-об’єкт з полями code, data та error. У випадку успіху поле data містить корисне навантаження, а error має значення null; при помилці в data повертається null, а в полі error надається структурований опис проблеми, придатний для локалізації у клієнтському інтерфейсі.

Приклад типової відповіді для успішного запиту має вигляд, наведений на рис. 2.11.

```
{
  "code": 200,
  "data": {
    "id": 255,
    "email": "user@example.com",
    "fullName": "John Doe"
  },
  "error": null
}
```

Рис. 2.11 Фрагмент коду типової відповіді для успішного запиту

У разі порушення правил автентифікації сервер може повернути, наприклад (рис. 2.12).

```
{
  "code": 401,
  "data": null,
  "error": {
    "Bearer": "Authorization token expired"
  }
}
```

Рис. 2.12 Фрагмент коду при порушенні правил автентифікації

Захист доступу до ресурсів забезпечується багатошаровим механізмом авторизації. Для кожного запиту на рівні проміжного програмного забезпечення виділяється значення заголовка Authorization, перевіряється валідність форми

токена, відповідність алгоритму підпису та параметрів iss і aud, а також термін дії. Після криптографічної перевірки підпису з використанням відкритого ключа з JWKS-сховища дані токена (ідентифікатор акаунта, ролі, контекст платформи) передаються в контролер, де виконується прикладна перевірка прав доступу до конкретного ресурсу. Це дозволяє відокремити технічну валідацію токена від бізнес-логіки й забезпечити гнучку рольову модель.

Структура REST-API відображає модульну побудову серверної частини. Модуль /account відповідає за роботу з профілями користувачів, ролями та платіжними методами. Саме через нього клієнтський застосунок отримує повну інформацію про поточний акаунт, завантажує списки мешканців та менеджерів окремого житлового комплексу, оновлює профіль, ініціює процедури зміни пароля чи відновлення доступу та керує відповідними параметрами безпеки. На рис. 2.12 наведено приклад виконання запиту GET /api/account/info/my у середовищі Postman, де видно як сформований заголовок авторизації, так і структуру успішної відповіді сервера у форматі JSON.

Модуль /apartments забезпечує доступ до інформації про житлові одиниці та їх зв’язок із користувачами і житловими комплексами. Через відповідні маршрути клієнт може отримати деталізований перелік квартир конкретного будинку, побачити пов’язаних мешканців, перевірити статуси зайнятості та використати ці дані для побудови інтерфейсів керування об’єктами.

Фінансові операції реалізовано в модулях /wallet і /payment. Перший модуль повертає історію змін балансу та агреговані показники гаманця, другий – відповідає за створення платіжних документів і запуск процесу оплати. Під час виконання запиту на оплату бекенд зчитує з бази даних параметри рахунку та налаштування гаманця, формує узгоджений платіжний запит до зовнішнього платіжного провайдера, а після отримання результату оновлює записи у таблицях payment, wallet_history та account. Фрагмент послідовності викликів для цього сценарію було детально показано у діаграмі послідовності розділу 2.5.

Модуль /maintenance реалізує повний життєвий цикл сервісних заявок. Клієнтський застосунок надсилає до цього модуля підготовлені дані про

проблему, бажаний час візиту та супровідні матеріали. Бекенд створює відповідний запис у таблиці maintenance, пов’язує його з житловим комплексом та квартирою, за потреби підбирає сервісну компанію з таблиці maintenance_services, а також ініціює створення діалогу для подальшого листування. Подальше оновлення статусів заявки, додавання відгуків і оцінок здійснюється через інші HTTP-методи того самого модуля.

Соціальні функції платформи зосереджено в модулях /clubs і /dialog. Через них користувачі отримують доступ до клубів, приватних постів, діалогів і повідомлень. REST-контракти цих модулів побудовано за аналогічними принципами: GET-запити повертають стрічки клубів, учасників або повідомлень; POST-запити створюють нові клуби, вступ до них або надсилання повідомлень; PATCH-запити застосовуються для зміни ролей учасників чи позначення повідомлень як прочитаних.

Особливу роль відіграє модуль /auth, через який реалізовано первинну автентифікацію, видачу JWT-токенів, оновлення сесій за допомогою refresh-токенів та увімкнення двофакторної автентифікації. Клієнтський застосунок надсилає облікові дані, отримує у відповідь пару access/refresh токенів та надалі використовує короткоживучі access-токени у заголовках HTTP-запитів.

Валідація даних і обробка помилок реалізовані на рівні проміжного ПЗ. Перед передачею керування до бізнес-логіки всі запити проходять перевірку обов’язкових полів, типів, форматів дат, сум і контактних даних. У разі порушення будь-яких обмежень формується відповідь з кодом 400 або 422 і деталізованим описом помилки, що на клієнті може бути локалізований та відображений користувачу у вигляді зрозумілого повідомлення. Така централізована валідація зменшує дублювання коду та гарантує однакову поведінку для всіх модулів API.

Для внутрішньої документації та регресійного тестування використовується колекція запитів у середовищі Postman. На рис. 2.13 показано фрагмент цієї колекції з групами маршрутів /account та /apartments, згрупованих за доменами.

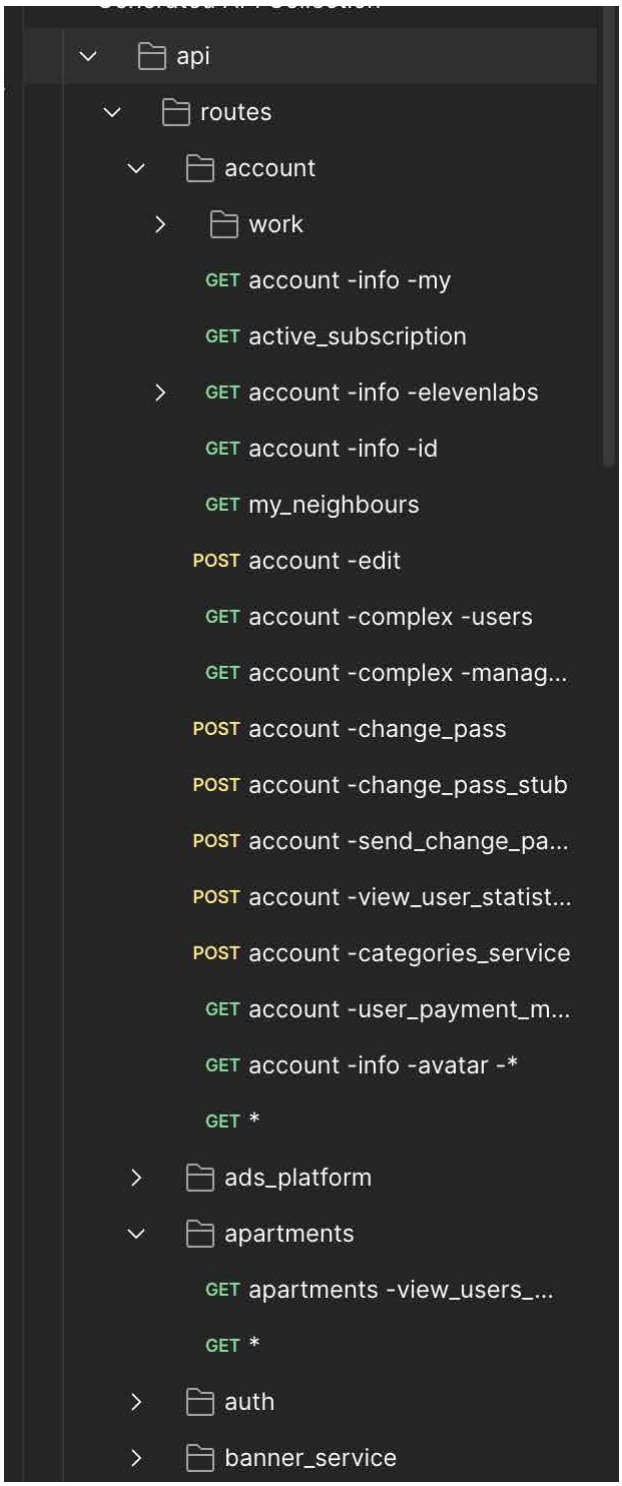


Рис. 2.13 Колекція API документації в Postman

Колекція відображає актуальну структуру REST-інтерфейсу, дозволяє налаштувати змінні середовища для різних оточень (sandbox та production), зберігає приклади валідних і помилкових запитів та використовується як робоча документація для розробників і тестувальників. На рис. 2.14 наведено приклад виконання запиту з авторизацією за допомогою JWT та переглядом розширеної

JSON-відповіді, що демонструє практичну форму використання описаних вище REST-контрактів.

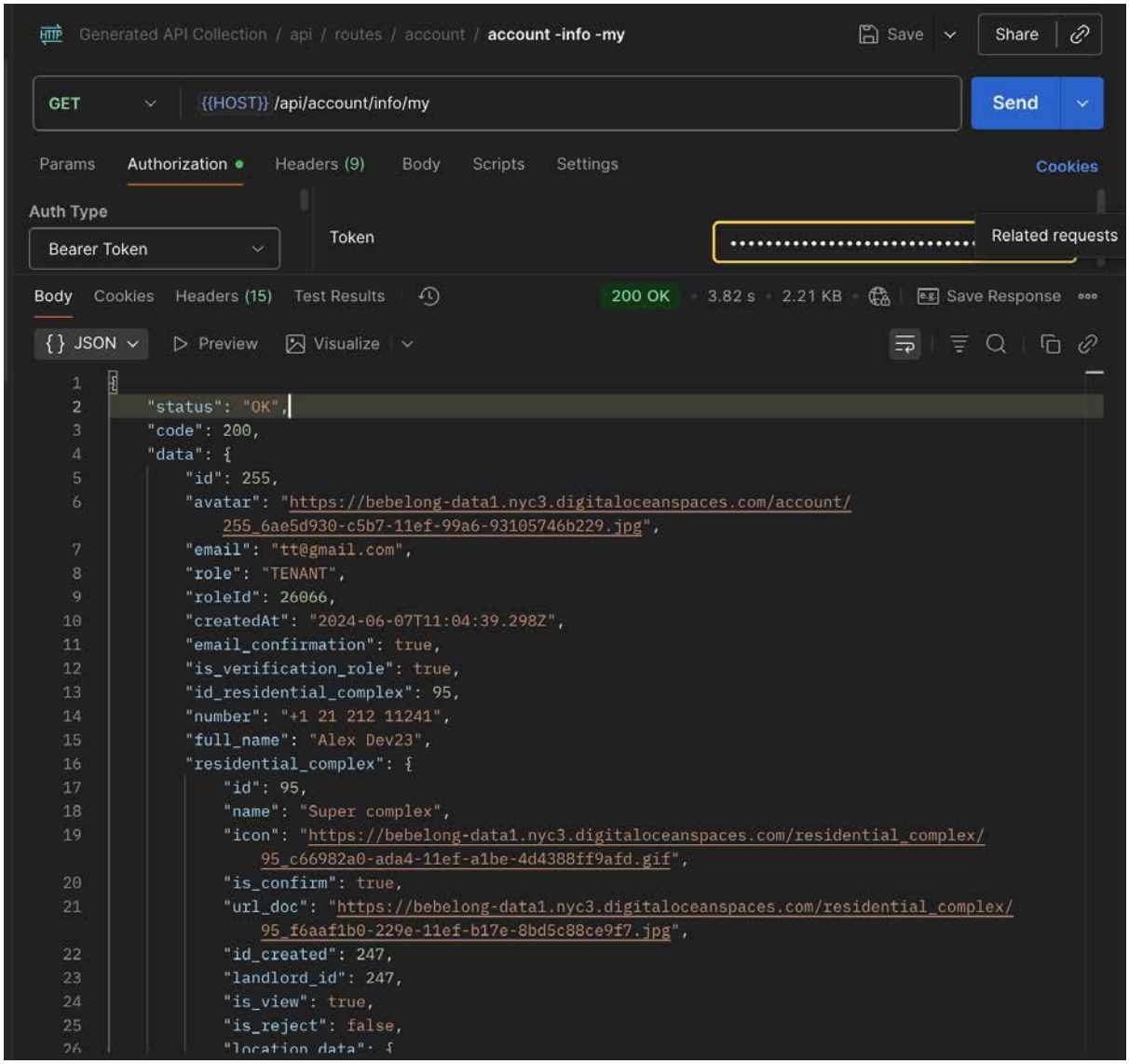


Рис. 2.14 Приклад формування запиту та відповіді від Серверу

Узгоджене використання HTTP-методів, єдиний формат відповідей, чітке розділення доменних модулів і централізована обробка помилок забезпечують прозору й передбачувану взаємодію між клієнтською та серверною частинами платформи. Це, у свою чергу, створює надійну основу для подальшого розширення функціоналу та інтеграції з зовнішніми сервісами без порушення існуючих API-контрактів.

Висновки до другого розділу

У другому розділі проведено системне обґрунтування вибраного технологічного стеку та сформовано вимоги до програмної платформи управління нерухомістю. Поєднання Flutter для клієнтської частини, React для веб-інтерфейсу та Node.js із PostgreSQL для серверної логіки забезпечує кросплатформність, масштабованість і відповідність вимогам інформаційної безпеки. На основі функціональних і нефункціональних вимог окреслено архітектурні принципи, що визначають структуру підсистем та їхній розподіл за доменами.

У ході проєктування архітектури побудовано модульну модель серверної частини з чітким шаруванням відповідальностей та виділенням доменів: управління акаунтами, платежами, сервісними заявками, повідомленнями та соціальними взаємодіями. Реляційна структура бази даних, розроблена на основі PostgreSQL, забезпечує цілісність і узгодженість даних та охоплює ключові сутності предметної області. Побудовані UML-діаграми – варіантів використання, класів, компонентів та послідовностей – формалізують логіку функціонування системи й відображають взаємодію між клієнтом, сервером і зовнішніми сервісами.

REST-API, описаний у підрозділі 2.6, визначає уніфікований механізм обміну даними, базований на HTTPS, JSON і JWT-автентифікації. Стандартизовані відповіді, централізована валідація та рольова авторизація забезпечують надійність і передбачуваність взаємодії. Сукупність отриманих моделей формує завершену архітектурну основу платформи та створює методологічну базу для подальшої реалізації та тестування програмного забезпечення.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Реалізація інтерфейсу користувача у Flutter

Клієнтська частина розробленої системи реалізована як кросплатформовий мобільний застосунок на основі технології Flutter, що забезпечує підтримку платформ iOS та Android із використанням єдиної кодової бази. Такий підхід дозволив досягти нативної продуктивності, плавності анімацій та узгодженої дизайн-системи для всіх екранів. Архітектура інтерфейсу побудована з чітким розділенням відповідальностей: графічні компоненти зосереджені на відображенні візуальних елементів, тоді як логіка взаємодії з REST-API, обробка даних і керування станом реалізовані у спеціалізованих контролерах GetX та провайдерах. Це дало змогу мінімізувати зв’язність компонентів, пришвидшити тестування та забезпечити можливість внесення змін у бізнес-логіку без модифікації інтерфейсних частин.

Структура UI ґрунтується на компонентному підході, у межах якого віджети карткового типу використовуються для представлення постів, оголошень, клубів, рахунків або профілів. Реактивність інтерфейсу забезпечується підпискою на зміни стану в контролерах; тому оновлення даних (наприклад, зміна статусу рахунку чи поява нового посту в стрічці) відображається негайно без додаткових викликів перерендерингу в усіх частинах дерева віджетів. Адаптивність макетів реалізована за допомогою гнучких контейнерів та механізмів врахування розмірів екрана, що дозволяє коректно відображати інтерфейс на пристроях різних форм-факторів. Єдині правила стилізації – палітра кольорів, шрифти, тіні й радіуси скруглення – формують цілісний дизайн-код системи.

На рис. 3.1 подано головний екран застосунку, який виступає центральною точкою входу користувача. Його структура включає верхню панель фільтрації, горизонтальну галерею аватарів сусідів, банер рекомендацій та основну динамічну стрічку контенту.

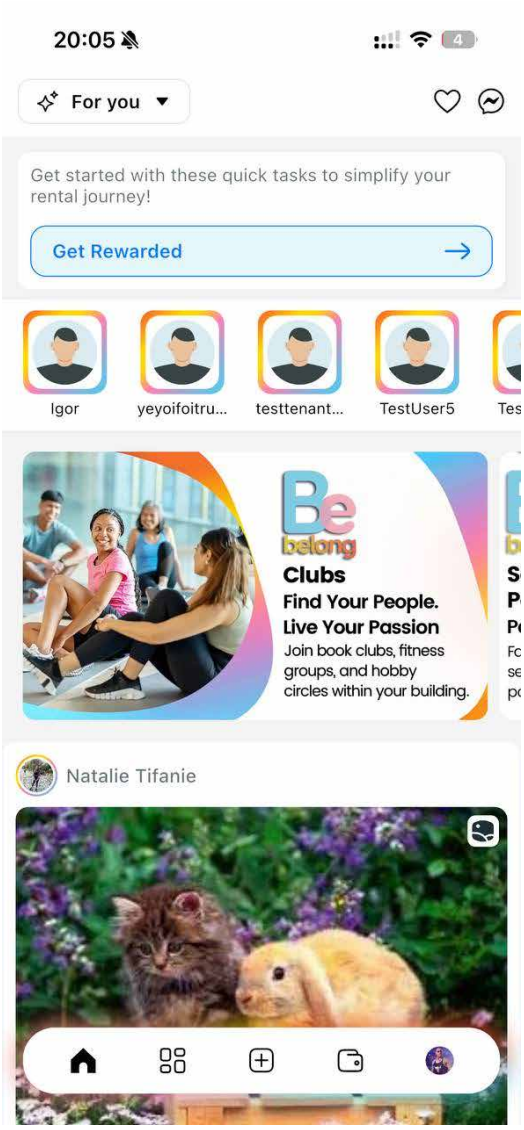


Рис. 3.1 Головний екран стрічки користувача

Елементи стрічки формуються контролером, що отримує дані з REST-API, після чого віджети автоматично відтворюють відповідні картки з фото, текстовою частиною, показниками взаємодії та кнопками дій. Такий підхід дозволяє використовувати однакову модель віджета для різних типів контенту, що зменшує дублювання коду та забезпечує стилістичну узгодженість.

Далі на рис. 3.2 наведено приклад відображення посту клубу або спільноти. Візуальний блок побудований у вигляді картки, що містить обкладинку, назву клубу, тематичні хештеги, індикатори взаємодії та кнопку переходу до повної інформації. Усі дії користувача – лайк, відкриття деталей клубу, додавання коментаря – обробляються через відповідні REST-ендпоїнти та

супроводжуються оновленням стану у Flutter, що забезпечує миттєве відображення результатів взаємодії.

Рис. 3.2 Екран перегляду посту клубу та взаємодії з ним

Особливе значення для зручності навігації має панель сервісів, реалізована у вигляді модального вікна (рис. 3.3). Вона складається з сітки інтерактивних плиток, кожна з яких відповідає окремому функціональному модулю системи: маркетплейс, клуби, події, сервісні заявки, спільнота, поїздки, фінансовий

модуль та управління нерухомістю. Натискання на відповідний елемент ініціює навігаційний перехід, що здійснюється засобами GetX або стандартного маршрутизатора Flutter. Завдяки цьому ключові функції системи зібрано у єдиному доступному просторі, а навігація стала інтуїтивною навіть для нових користувачів.

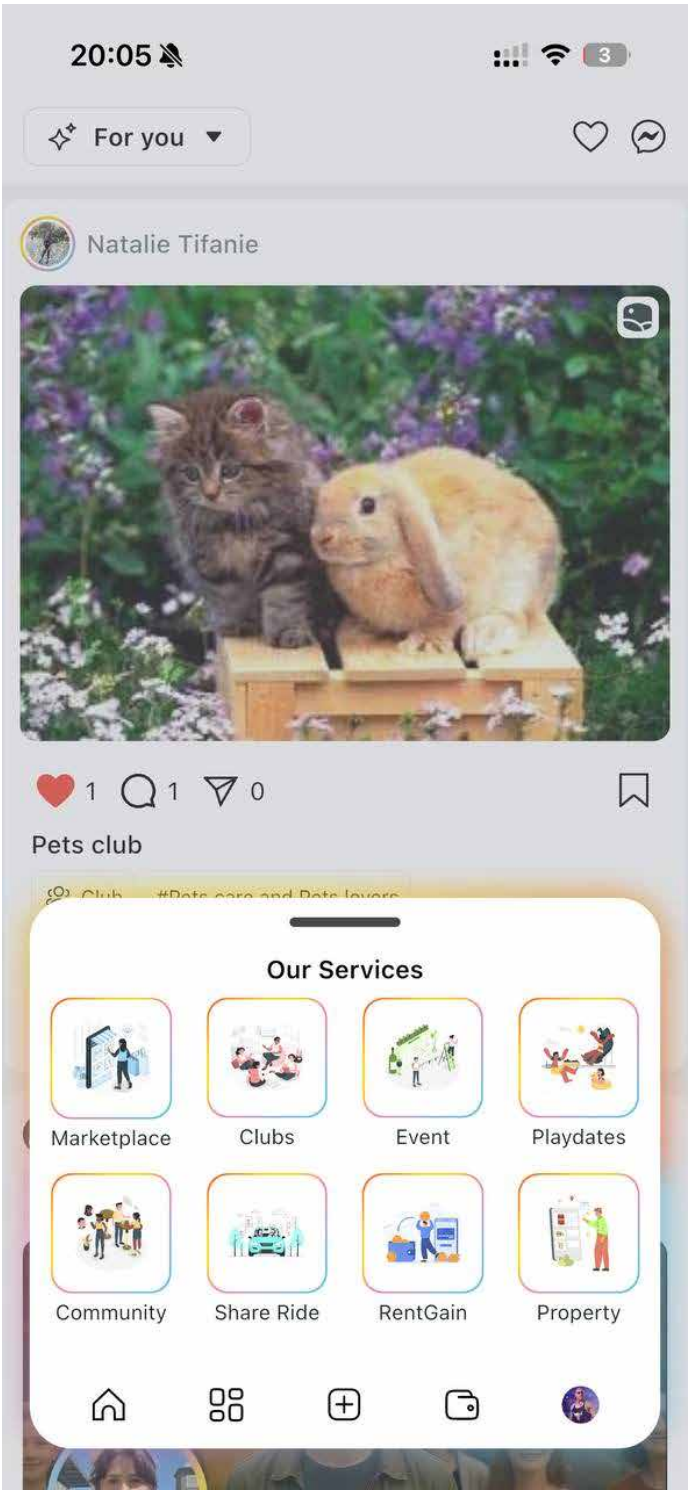


Рис. 3.3 Модальне вікно «Our Services» з основними сервісами платформи

На рис. 3.4 показано екран публічного профілю користувача, який відображає персональну інформацію, біографічні дані та інтереси. Профіль побудовано як набір вкладених карток, що формують логічно структуровану сторінку зі скролом. Контактні дані завантажуються із сервера під час відкриття екрана, а відображення списку інтересів реалізовано за допомогою динамічних елементів типу Chips, що адаптуються під доступний простір. Такий підхід дозволяє забезпечити як інформативність, так і зручність взаємодії з даними користувача.

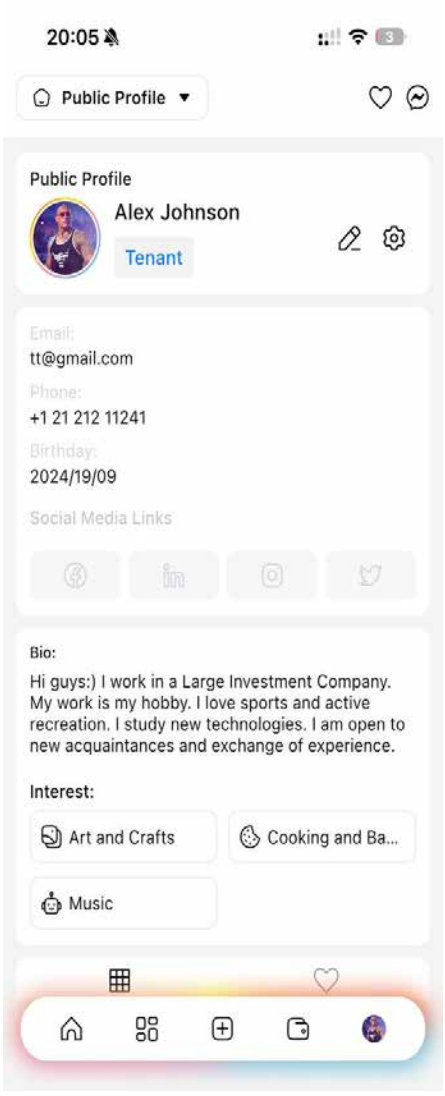


Рис. 3.4 Екран публічного профілю користувача з відображенням інтересів

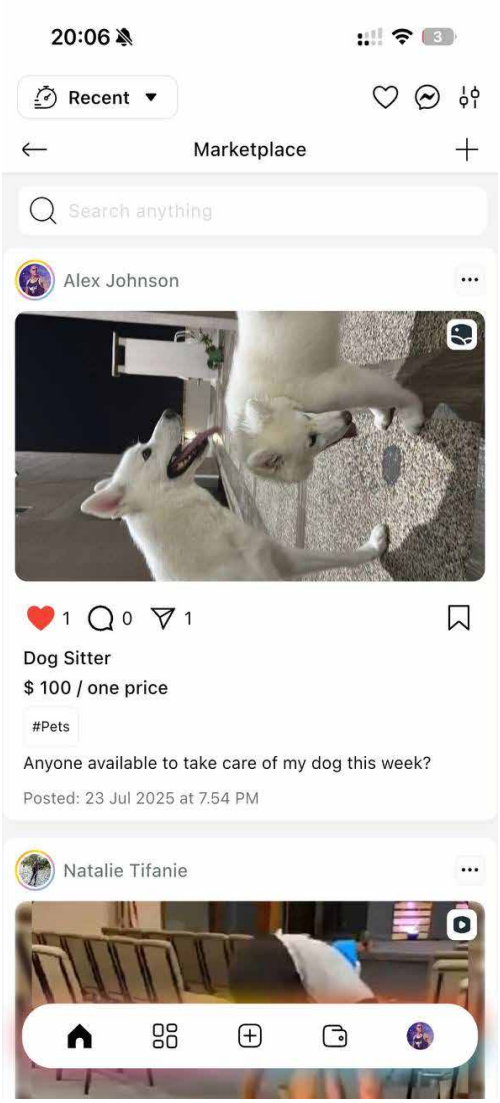


Рис. 3.5 Екран маркетплейсу з переліком оголошень мешканців

Функціонал маркетплейсу (рис. 3.5) демонструє застосування шаблонної карткової структури для представлення оголошень і сервісів, що публікують мешканці. Екран містить панель фільтрації, поле пошуку та вертикальну стрічку оголошень. Кожне оголошення подано у вигляді картки з фото, заголовком, описом, зазначенням ціни та категорії. Після натискання здійснюється перехід до детальної сторінки, де відображається повна інформація про пропозицію. Список оголошень синхронізується з бекендом, а інтерфейс автоматично реагує на зміну даних.

Фінансовий модуль системи представлено двома ключовими екранами, схема яких наведена на рис. 3.6 та 3.7. Перший відображає перелік рахунків користувача у стандартизованих картках, що містять номер інвойсу, суму, категорію, статус та кінцеву дату сплати.

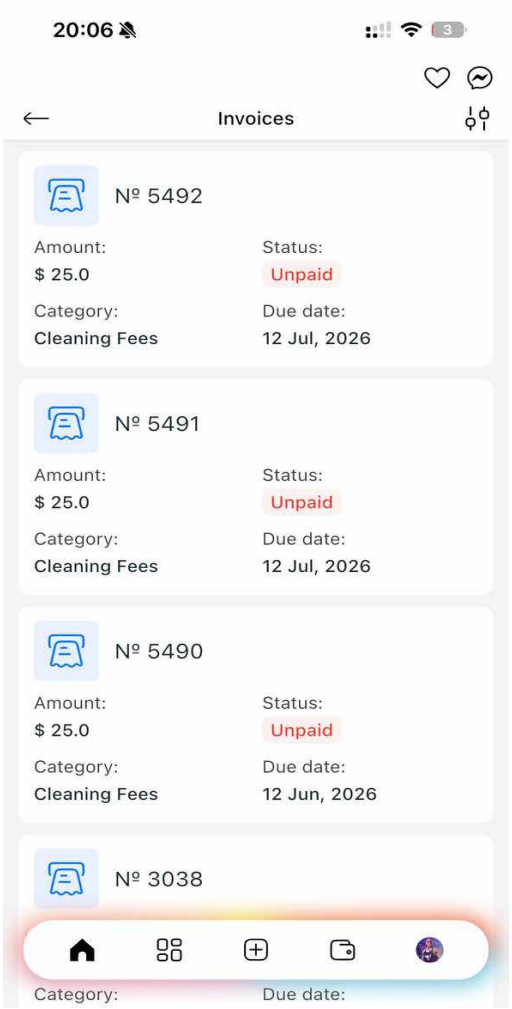


Рис. 3.6 Екран зі списком рахунків користувача

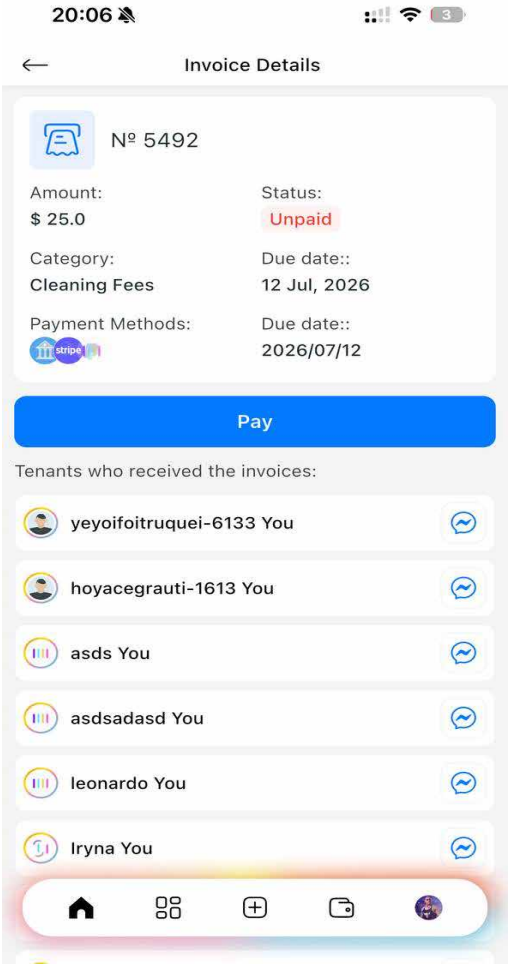


Рис. 3.7 Екран деталей інвойсу з можливістю оплати

Другий подає деталізовану інформацію про окреме нарахування та забезпечує можливість здійснення оплати через інтегрований платіжний сервіс. Після виконання транзакції контролер оновлює відповідний запис у стані, що миттєво відображається на екрані.

Узагальнюючи результати реалізації інтерфейсу у Flutter, можна зазначити, що технологія дозволила сформувати цілісний, адаптивний і продуктивний UI, який відповідає вимогам кросплатформеності та забезпечує узгоджене відтворення всіх основних функцій системи. Використання єдиної дизайн-системи та компонентного підходу спростило підтримку застосунку, тоді як реактивна архітектура та інтеграція з REST-API забезпечили оперативне оновлення даних і стабільність роботи. Завдяки цьому інтерфейс став повноцінною частиною комплексної програмної системи управління нерухомістю.

3.2. Серверна логіка Node.js + PostgreSQL

Серверна частина програмної системи реалізована на платформі Node.js з використанням фреймворка типу Express та реляційної системи керування базами даних PostgreSQL. Обраний технологічний стек забезпечує неблокуючу обробку великої кількості HTTP-запитів, підтримує масштабування за рахунок асинхронної моделі введення–виведення та гарантує коректність збереження даних завдяки транзакційним механізмам СУБД. Логіка бекенду побудована як монолітний застосунок із чітко виділеними модулями, що спрощує підтримку та розгортання на цільовій інфраструктурі.

Структура вихідного коду організована за доменним принципом. У каталозі api/routes розміщено файли, які описують REST-маршрути для основних підсистем, зокрема акаунтів, об’єктів нерухомості, автентифікації, фінансових операцій, клубів та сервісних заявок. Каталог api/middlewares містить проміжні обробники, відповідальні за перевірку токенів доступу, двофакторної автентифікації, ролей користувачів, а також базове логування та нормалізацію запитів. У api/service зосереджено бізнес-логіку окремих модулів, що працюють

із балансами, інвойсами, внутрішнім гаманцем, чатами та іншими доменними об’єктами. Моделі доступу до даних згруповано в каталозі api/struct_model; кожен файл описує відповідність між таблицями бази даних (account, payment, wallet_history, maintenance тощо) та сутностями прикладного рівня. Додаткові каталоги на кшталт cron_task, house_history_logger, ssl_cert, telegram-logs використовуються для планувальників періодичних задач, логерів та інтеграції із зовнішніми сервісами. Узагальнену структуру серверної частини подано в табл. 3.1.

Таблиця 3.

Основні каталоги серверної частини застосунку

Каталог	Призначення
api/routes	Опис REST-маршрутів і прив’язка HTTP-методів до контролерів
api/middlewares	Проміжні обробники безпеки, автентифікації, логування
api/service	Реалізація бізнес-логіки та взаємодія з моделями даних
api/struct_model	Оголошення ORM-моделей таблиць PostgreSQL
cron_task	Набір фонових задач для періодичного виконання
house_history_logger, telegram-logs	Сервіси збору та передавання логів у зовнішні системи
ssl_cert	Сертифікати та налаштування захищеного з’єднання

Типовий життєвий цикл HTTP-запиту організовано таким чином, що після надходження на один із маршрутів у api/routes він послідовно проходить крізь набір middleware, де перевіряються токени доступу, статус двофакторної автентифікації, ролі користувача та інші атрибути безпеки. Далі запит спрямовується до контролера, який взаємодіє із сервісним шаром, передаючи параметри, тіло запиту та ідентифікатор користувача, отриманий з JWT-токена. Сервіс виконує бізнес-операції, звертаючись до ORM-рівня, зовнішніх API (наприклад, платіжного провайдера) і, за потреби, відкриваючи транзакцію у

PostgreSQL. На завершальному етапі результат перетворюється у стандартизований JSON-відповідь, яка повертається клієнтові.

Робота з PostgreSQL здійснюється через ORM-шар, який відображає таблиці бази даних на класи JavaScript. Для кожної сутності, представленої в ER-діаграмі, створено модель із набором полів, що відповідають стовпцям таблиці: сутність Account відображається на таблицю account, Payment – на payment, WalletHistory – на wallet_history, MaintenanceRequest – на maintenance, а також визначені моделі для таблиць maintenance_services, dialogue, message, club, club_member, private_post та інших. У сервісному шарі ці моделі використовуються для пошуку даних, створення нових записів, оновлення станів і видалення застарілої інформації. Окрему увагу приділено підтримці транзакцій, що особливо важливо для фінансових операцій, де необхідно узгоджено оновлювати кілька взаємопов’язаних таблиць.

Приклад використання ORM для вибірки інвойсів користувача з урахуванням фільтрації за статусом оплати, категорією послуги та часовим інтервалом наведено в дод. Ж.1. У наведеному фрагменті формується об’єкт умов where, який враховує ідентифікатор квартири, тип платежу, статус оплати, вибрану категорію та діапазон дат, після чого виконується запит до моделі PaymentDb із застосуванням пагінації та сортування за часом створення.

Одним із ключових компонентів серверної частини є модуль автентифікації, який реалізує видачу та перевірку JWT-токенів і взаємодіє з таблицею account та таблицею зберігання refresh-токенів. Після успішної перевірки облікових даних формується корисне навантаження токена, що включає ідентифікатор користувача, адресу електронної пошти, роль та інші службові поля. Далі за допомогою криптографічної бібліотеки генерується підписаний токен із алгоритмом RS256, зазначенням видавця, суб’єкта, часу створення та часу закінчення дії. Перевірка токена у middleware передбачає валідацію формату, сигнатури, очікуваних значень полів iss і aud, а також контроль часу життя. У випадку порушення будь-якої з умов ініціюється виняток, який перетворюється на відповідь сервера з кодом 401 Unauthorized.

В дод. Ж.2 наведено фрагмент коду, що демонструє функцію генерації токена доступу та функцію його перевірки. У процесі генерації використовується ключ із сховища JWKS, а при перевірці – валідація заголовка й корисного навантаження з урахуванням налаштувань видавця, аудиторії та допустимого відхилення системного часу.

Довготривалі сесії користувачів підтримуються через механізм refresh-токенів, які зберігаються у спеціальній таблиці разом із відомостями про пристрій, IP-адресу, час створення та статус відкликання. Під час ротації refresh-токена попередній запис позначається як недійсний, а новий фіксується із оновленими параметрами. Це забезпечує контроль за активними сесіями та дає змогу централізовано їх завершувати у разі втрати пристрою або зміни політики безпеки.

Важливим напрямом є реалізація бізнес-логіки оплати рахунків. Цей функціонал охоплює таблиці payment, wallet_history, wallet_sett, user_payment_method та відповідні сервіси, які координують взаємодію із зовнішнім платіжним провайдером. Під час обробки запиту на оплату контролер отримує ідентифікатор рахунку, завантажує відомості з таблиці payment разом із пов’язаними даними, зчитує налаштування комісій із wallet_sett та параметри обраного платіжного методу. Далі в межах транзакції PostgreSQL виконується звернення до платіжного сервісу; у разі успішної відповіді створюється запис у wallet_history з детальною інформацією про транзакцію, оновлюється статус рахунку та відповідні поля в таблиці account, що відповідають зміні балансу. У випадку помилки будь-який із кроків спричиняє відкат транзакції, а клієнт отримує повідомлення про неможливість завершити операцію.

В дод. Ж.3 показано фрагмент сервісу, який формує платіжну сесію в Stripe, розраховує комісію платформи, передає необхідні параметри у зовнішній сервіс і одночасно фіксує операцію у таблиці wallet_history.

Паралельно з фінансовими операціями функціонує модуль обробки сервісних заявок, інформація про які зберігається в таблиці maintenance. При створенні нової заявки формується запис із базовим описом проблеми, станом

об’єкта та супутніми параметрами, а також ініціюється діалог між мешканцем і виконавцем, що реалізується через таблиці dialogue та message. Зміна статусів заявок, додавання фотографій, технічних коментарів і результатів виконання робіт фіксується в полях status, logs, pictures. На основі накопичених даних окремий модуль AI-аналітики формує записи у таблиці maintenance_ai_analysis, де зберігаються агреговані оцінки стану, прогнози щодо навантаження на сервісні служби та рекомендації щодо планових ремонтів. Додатково реалізовано модуль AI-чатів, який зберігає контекст взаємодії користувача з помічником у таблицях maintenance_chat_sessions_ai та maintenance_chat_mess_ai і забезпечує формування коротких резюме діалогу для менеджера.

Для підвищення надійності системи використовуються middleware безпеки, механізми логування та фонова обробка задач. Модуль безпеки налаштовує політики CORS, встановлює захисні HTTP-заголовки, обмежує розмір тіла запиту та, за необхідності, активує додаткові перевірки для маршрутів, пов’язаних із критичними операціями. Логер фіксує основні характеристики запитів і відповідей, включно з методом, URL, кодом статусу та часом обробки, а для важливих бізнес-подій може відправляти нотифікації до зовнішніх каналів моніторингу. У каталозі cron_task розміщено задачі, що періодично виконують архівацію логів, очищення застарілих заявок, надсилання нагадувань про неоплачені рахунки та ротацію криптографічних ключів.

Узагальнюючи наведене, серверна логіка Node.js + PostgreSQL реалізує повний цикл обробки запитів – від автентифікації користувача та перевірки його прав до виконання складних фінансових транзакцій, аналітичної обробки сервісних заявок і підтримки фонових процесів. Такий підхід забезпечує узгодженість даних, високу надійність і розширюваність системи, створюючи основу для подальшої інтеграції нових функціональних модулів.

3.3. Демонстрація ключових модулів

Для підтвердження працездатності розробленої платформи у цьому підрозділі узагальнено роботу трьох базових модулів: системи управління платежами, фінансового аналітичного дашборда та модуля комунікації між учасниками спільноти. Кожен з них реалізує окрему групу бізнес-процесів, але взаємодіє з єдиною серверною логікою на Node.js і спільною базою даних PostgreSQL. Загальна характеристика модулів наведена в табл. 3.2.

Таблиця 3.2

Ключові модулі програмної системи

Модуль	Основне призначення
Управління платежами	Створення, відображення й обробка інвойсів, інтеграція з онлайн-платежами
Фінансовий аналітичний дашборд	Агрегація й візуалізація доходів, витрат та показників ефективності
Комунікація користувачів	Соціальна взаємодія мешканців і службові повідомлення між учасниками платформи

Система управління платежами забезпечує повний життєвий цикл фінансових документів від моменту формування рахунку до його оплати та синхронізації з зовнішніми обліковими системами. На рис. 3.8 подано веб-інтерфейс списку інвойсів у кабінеті управителя. У таблиці відображаються ідентифікатор рахунку, пов’язаний об’єкт нерухомості, тип платежу, дата сплати, статус, загальна сума, застосована сума та розмір заборгованості. Панель фільтрів у верхній частині дозволяє звузити вибірку за статусом, категорією доходу, конкретною будівлею чи часовим інтервалом, а поле пошуку – швидко знаходити рахунки за користувачем або номером. Наявність індикатора синхронізації з зовнішньою системою (наприклад, позначка Sync) дає змогу контролювати, які інвойси вже передано до бухгалтерського обліку. При зверненні до цього екрану фронтенд надсилає до бекенду параметризований запит; серверна частина формує SQL-запит з урахуванням фільтрів і повертає сторінковий список рахунків.

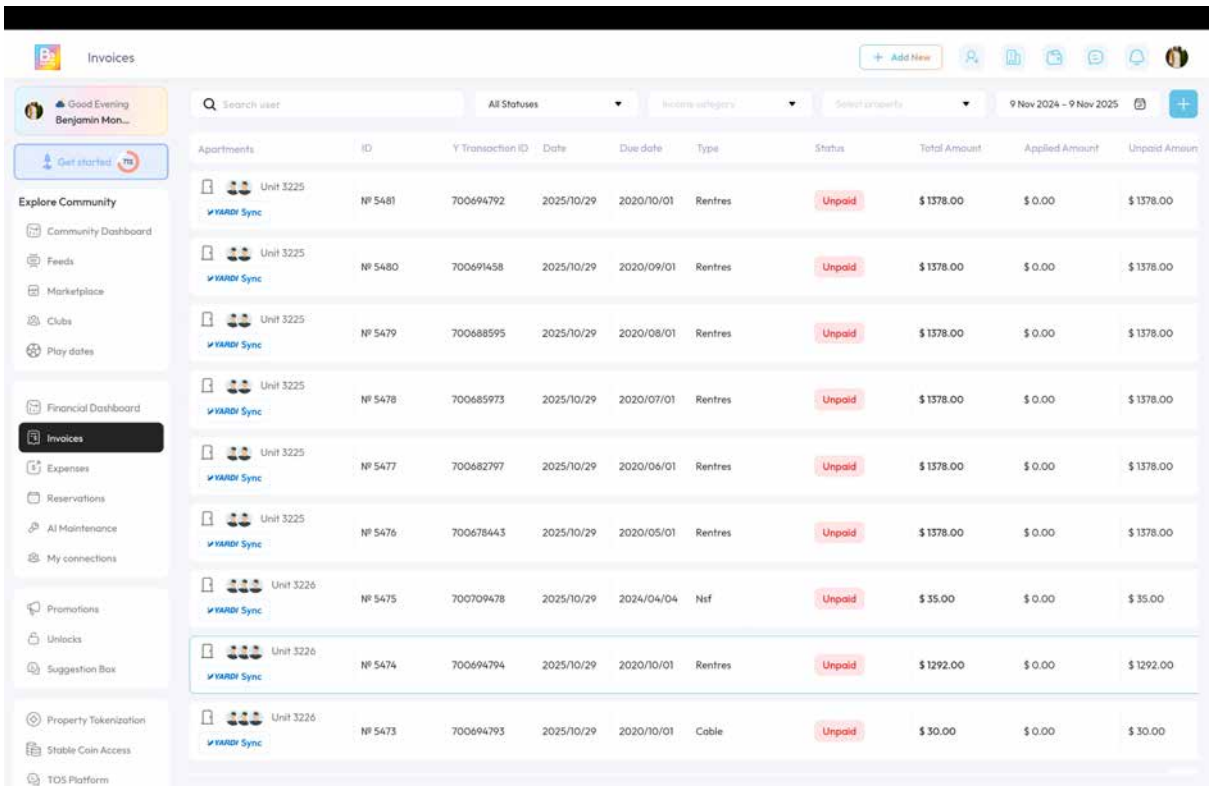


Рис. 3.8 Список інвойсів у веб-кабінеті управителя

Створення нового рахунку здійснюється у окремому інтерфейсі налаштувань інвойсу, зображеному на рис. 3.9.

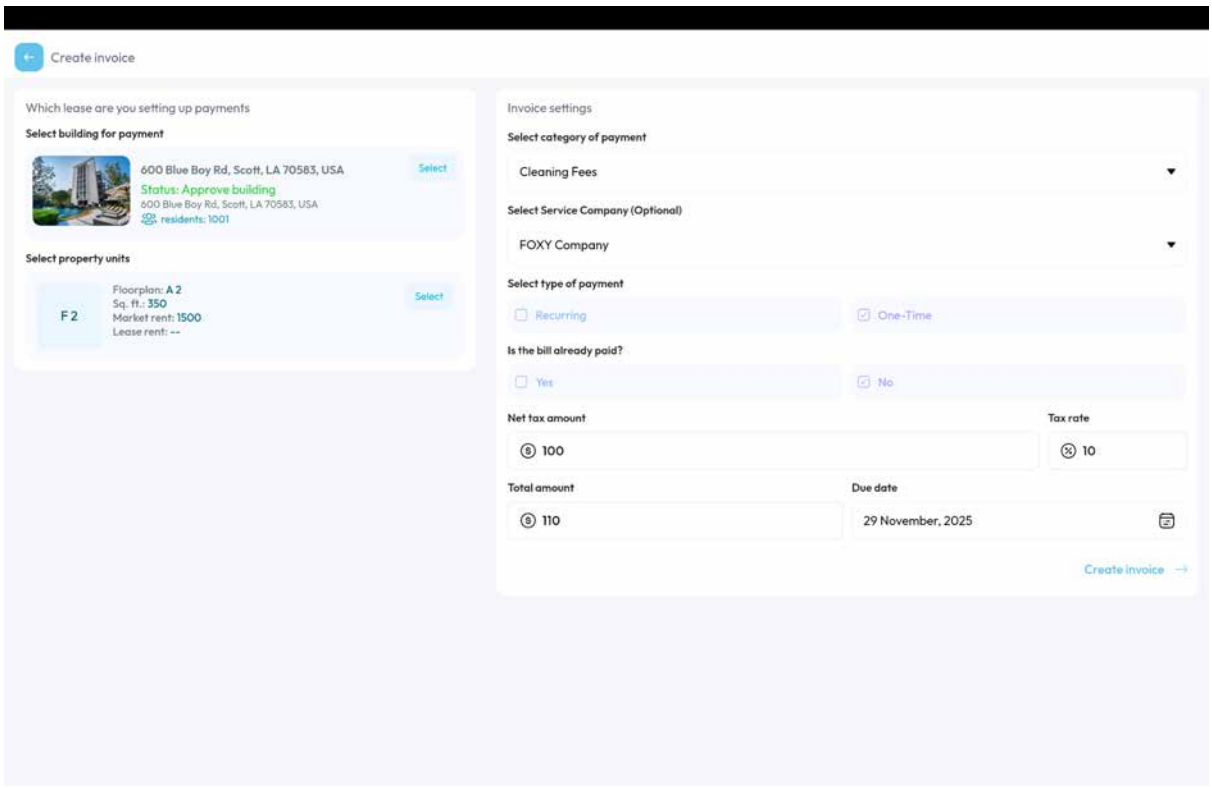


Рис. 3.9 Екран створення нового інвойсу

Ліва частина екрана призначена для вибору будівлі та конкретного юніту, що забезпечує правильне прив’язування платежу до об’єкта нерухомості. Права частина містить форму з параметрами рахунку: категорією платежу, за потреби – сервісною компанією, типом платежу (разовий чи рекурентний), сумою без податку, податковою ставкою, підсумковою сумою й датою сплати. Після підтвердження форми відповідний контролер на сервері створює запис у таблиці payment, ініціює формування початкового запису в wallet_history та, у разі налаштування онлайн-оплати, готує вихідні дані для інтеграції з платіжним провайдером.

Для кінцевих користувачів, які працюють переважно з мобільним застосунком, реалізовано спрощений інтерфейс перегляду та оплати рахунків. Список інвойсів відображається у компактному вигляді з номером, категорією, сумою та статусом; перехід до деталізації відкриває екран з повною інформацією та кнопкою ініціації оплати. При натисканні цієї кнопки застосунок звертається до REST-ендпойнта оплати, описаного у підрозділі 3.2, який формує сесію онлайн-платежу. Після успішного завершення транзакції webhook-обробник на сервері оновлює статус рахунку та відповідний запис в історії транзакцій.

Другим ключовим компонентом є фінансовий аналітичний дашборд, який надає управителям агреговане уявлення про стан портфеля нерухомості. На рис. 3.10 показано базову конфігурацію дашборда. У верхньому рядку панелі відображаються інтегральні показники: кількість об’єктів, загальна кількість юнітів, число зайнятих та вакантних юнітів, кількість зареєстрованих і верифікованих мешканців за обраний період. Центральне місце займає блок Total income із круговою діаграмою, що демонструє структуру доходів за категоріями, а також графік Rent vs other income, який відображає динаміку надходжень у часовому розрізі та дозволяє оцінити сезонні коливання. Допоміжні блоки містять інформацію про рекламні кампанії, обсяг доходів від реклами та поточний баланс.

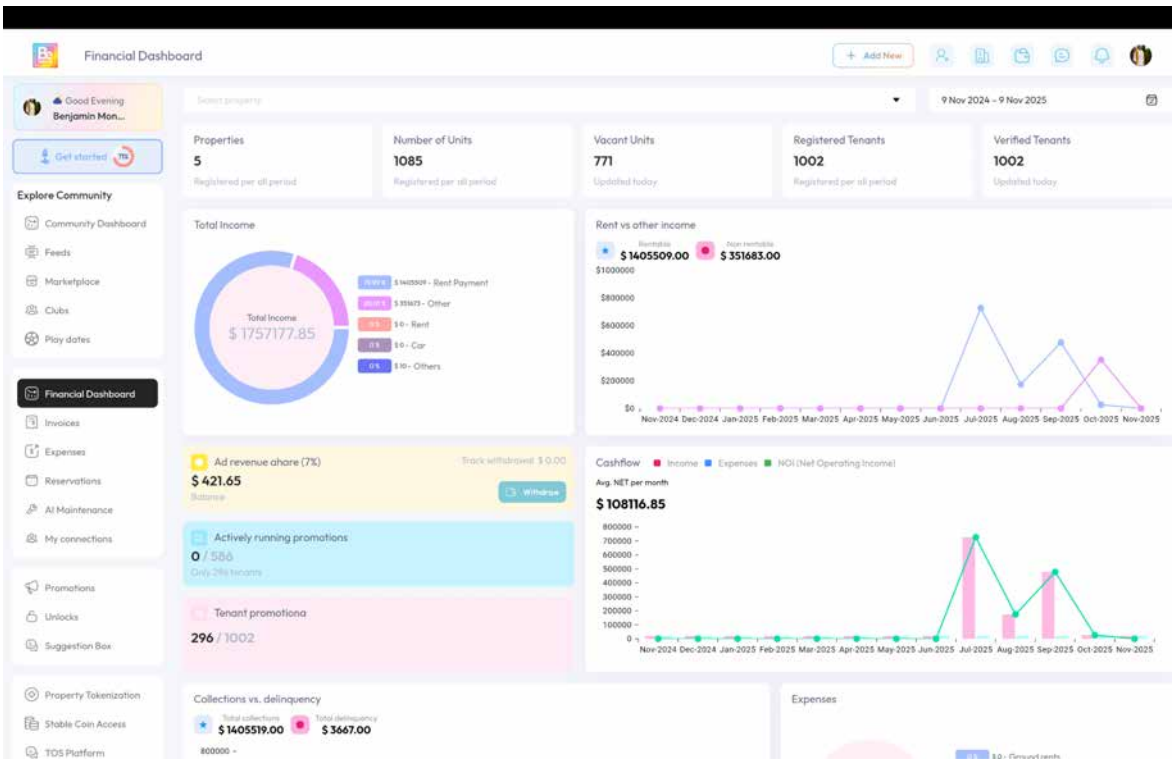


Рис. 3.10 Фінансовий дашборд: агреговані доходи та основні KPI

На серверному боці цей модуль реалізовано через набір REST-ендпойнтів, що виконують агрегувальні запити до PostgreSQL. Для розрахунку показників використовуються функції сумування, групування за категоріями платежів, об’єктами та періодами, а також розрахунок похідних метрик на кшталт середнього місячного кешфлоу або питомої заборгованості на юніт. На клієнтському боці отримані дані кешуються, перетворюються у структури, придатні для відображення діаграм, та оновлюються при зміні фільтрів дашборда. Узагальнені приклади таких метрик подані в табл. 3.3.

Таблиця 3.3

Приклади показників, що обчислюються для фінансового дашборда

Показник	Джерело даних	Опис
Загальний дохід за період	Таблиця payment	Сума всіх сплачених інвойсів за обраний інтервал
Частка прострочених платежів	payment, статуси оплати	Відношення суми прострочених рахунків до загальної
Середній місячний кешфлоу	payment, wallet_history	Різниця між надходженнями та витратами
Кількість активних промо-кампаній	Таблиця промо-активностей	Число кампаній зі статусом «активна»

На рис. 3.11 наведено альтернативний вигляд дашборда, який акцентує увагу на аналізі інкасацій, простроченої заборгованості та результативності промо-кампаній. Діаграма Collections vs. delinquency дає змогу порівняти сукупний обсяг отриманих коштів із сумою прострочених рахунків у розрізі місяців, що полегшує виявлення проблемних періодів. Поряд розміщено блок Expenses із круговою діаграмою розподілу витрат за категоріями, а також таблицю Top 5 promotions, яка відображає кількість переглядів, кліків, конверсію та дохід за кожною маркетинговою активністю.

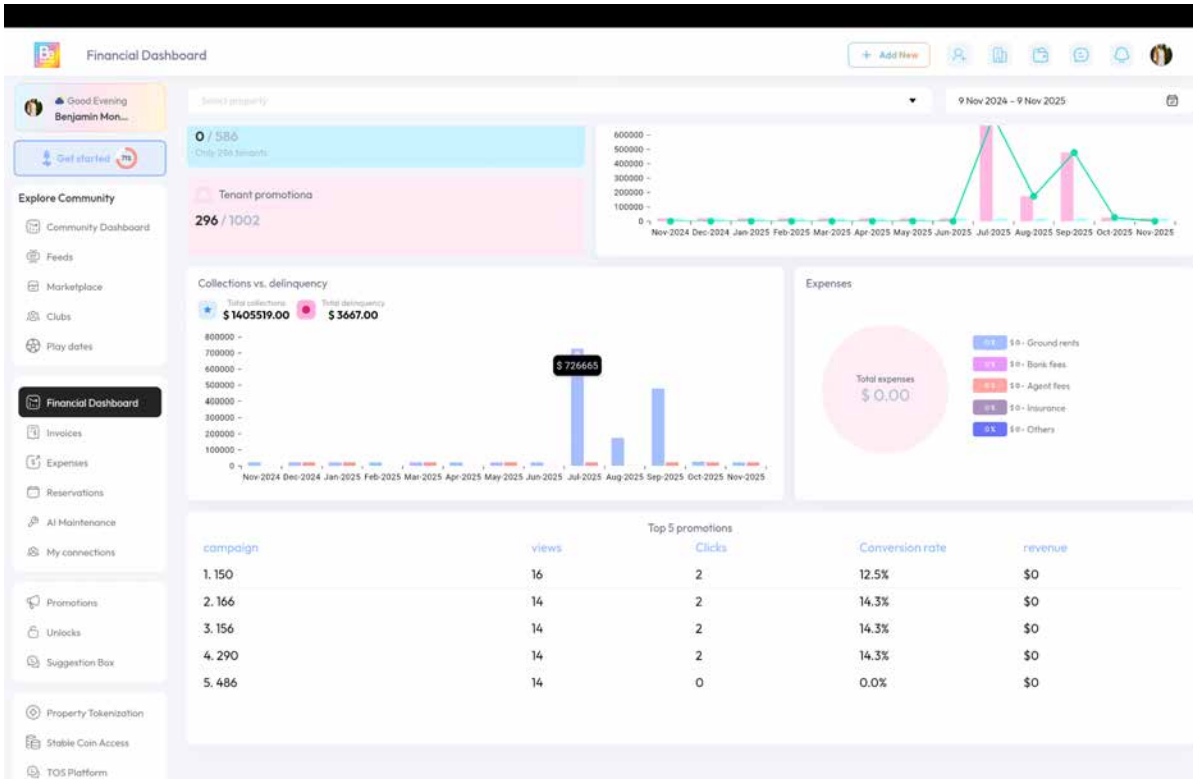


Рис. 3.11 Фінансовий дашборд: аналіз інкасацій, прострочень та промо-кампаній

Третій модуль – система комунікації – поєднує соціальну взаємодію мешканців із службовими комунікаціями між управителями, власниками та орендарями. На рівні бази даних він спирається на таблиці clubs, private_post, dialogue, message, club_member та account, що дозволяє реалізувати як масові повідомлення у клубах за інтересами, так і індивідуальні діалоги. У мобільному застосунку цей модуль проявляється у вигляді стрічки новин і розділу клубів, де користувачі можуть переглядати публікації, реагувати на них, коментувати та

долучатися до тематичних спільнот. Веб-кабінет управителя орієнтований на оперативний доступ до інформації про об’єкти й мешканців та на організацію службового спілкування.

На рис. 3.12 подано екран «My properties», що відображає портфель об’єктів нерухомості. Для кожної будівлі показано її назву, адресу, кількість юнітів і мешканців, а також індикатори зайнятості. З цього екрана управитель може перейти до управління конкретною нерухомістю, включно з переглядом заявок, фінансових показників і комунікацій.

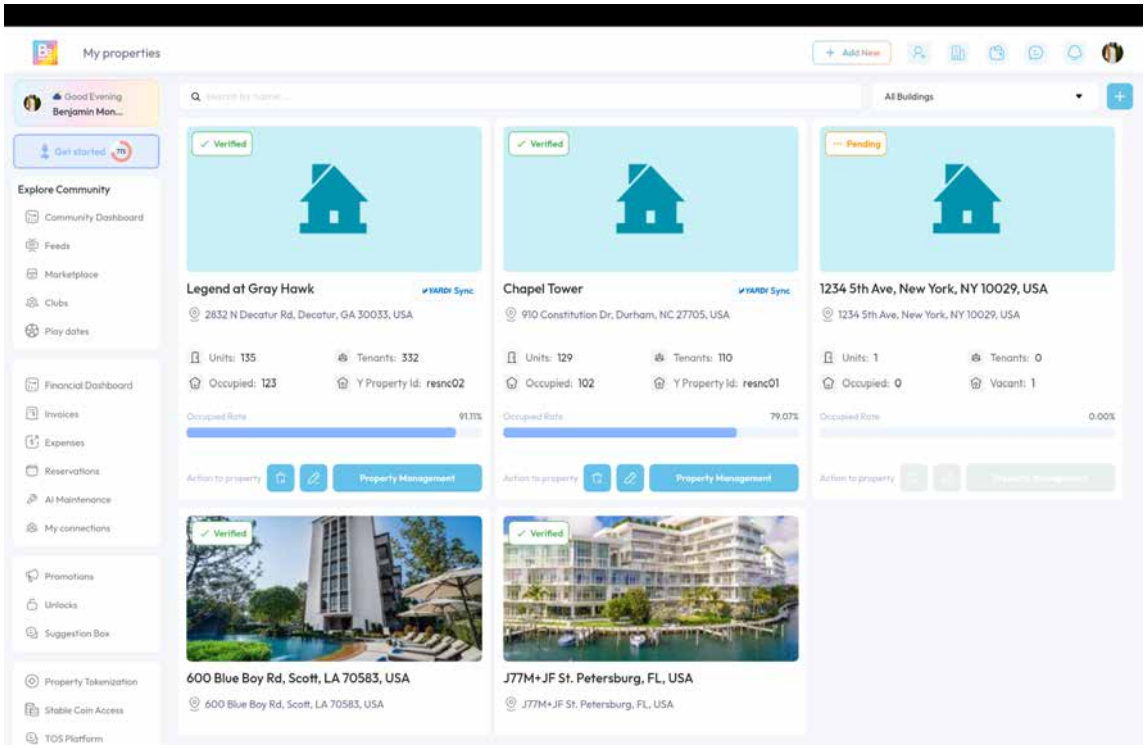


Рис. 3.12 Перелік об’єктів з панеллю переходу до засобів управління

Деталізований рівень взаємодії демонструє вкладка Tenants у модулі управління будівлею, зображена на рис. 3.13. Таблиця містить перелік усіх мешканців, їхні контактні дані, статус KYC-верифікації та службові поля, а також набір кнопок швидкого доступу до документів і деталей акаунта. З цього інтерфейсу можливий миттєвий перехід до діалогу з конкретним користувачем або надсилання йому службового повідомлення. На серверному рівні такі дії відповідають виклику REST-ендпойнтів модулів облікових записів і повідомлень.

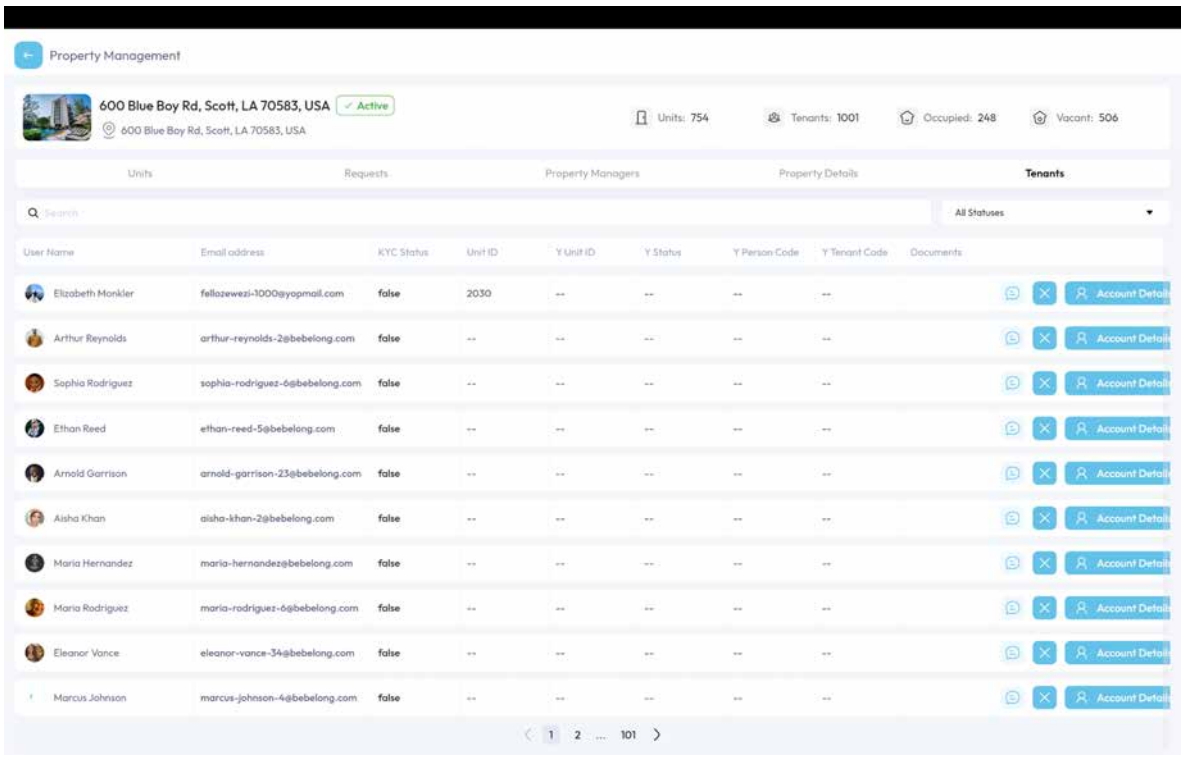


Рис. 3.13 Вкладка Tenants з інформацією про мешканців будівлі

Завершальною ланкою модуля є екран профілю користувача в адміністративній панелі (рис. 3.14).

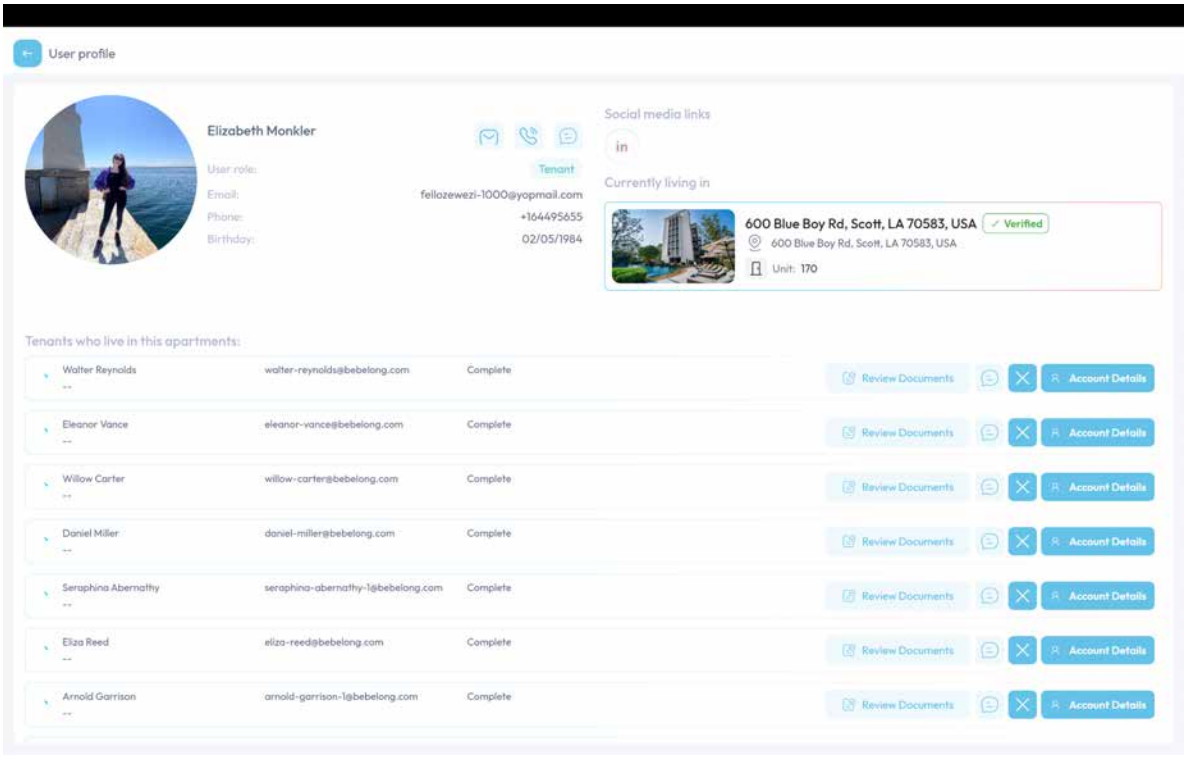


Рис. 3.14 Екран профілю користувача з інструментами взаємодії

Тут відображаються основні персональні дані, контактна інформація, посилання на профілі в соціальних мережах, поточна адреса проживання з указанням будівлі та юніту, а також перелік співмешканців. Наявність елементів керування дозволяє запускати сценарії зв’язку, змінювати статус верифікації, обмежувати доступ або, навпаки, відкривати додаткові можливості для певних категорій користувачів. Усі ці операції перевіряються middleware безпеки, що контролюють ролі та права доступу, а критичні дії додатково журналюються для подальшого аудиту.

Таким чином, модулі управління платежами, фінансового аналізу та комунікацій утворюють взаємопов’язаний функціональний комплекс. Вони використовують спільні механізми автентифікації, авторизації та логування, спираються на єдину модель даних і взаємодіють із REST-інтерфейсами, описаними в попередніх підрозділах. Така інтеграція дає змогу не лише демонструвати коректну роботу окремих компонентів, а й підтверджує цілісність і практичну придатність розробленої інформаційної системи для управління нерухомістю.

3.4. Особливості розгортання й експлуатації

Розгортання програмної системи здійснюється з урахуванням вимог до модульності, низького зв’язування компонентів, прозорості інтерфейсів взаємодії, надійного аудиту операцій та забезпечення подальшої масштабованості. Архітектура серверної частини побудована таким чином, щоб окремі функціональні блоки могли розгортатися й оновлюватися незалежно один від одного, зберігаючи цілісність системи та стабільність роботи користувацького інтерфейсу. В основі розгортання лежить DevOps-підхід, який включає автоматизоване складання, тестування й доставлення оновлень за допомогою інструментів CI/CD.

Базове оточення серверної частини формується на основі контейнеризації сервісів. Сервіс автентифікації та безпеки, підсистеми керування ролями, модуль управління нерухомістю, блоки обробки платежів, модуль WalletHistory, система

опрацювання сервісних заявок та комунікацій, аналітичний модуль і адміністративна панель можуть розміщуватися як у єдиному контейнерному середовищі, так і в окремих контейнерах залежно від вимог до продуктивності. Такий підхід мінімізує залежності між компонентами й полегшує оновлення, оскільки кожний модуль може бути перебудований та розгорнутий окремо, без зупинки всієї системи. Контейнери взаємодіють через стандартизовані HTTP-інтерфейси, що забезпечує прозору й керовану комунікацію між сервісами.

У процесі розгортання бази даних PostgreSQL застосовується стратегія централізованого зберігання даних із суворим дотриманням транзакційної узгодженості. Первинне інсталювання включає створення основних схем, таблиць та індексів відповідно до моделей доменної області. З метою підтримання постійної актуальності структури БД застосовуються міграції, які виконуються автоматично на кожному середовищі (development, staging, production) під час доставки оновлень. Модуль Auth & Security додатково використовує окремі таблиці для керування ключами підпису, історією refresh-токенів та журналами безпеки, що потребує регулярної ротації ключового матеріалу.

Експлуатація системи передбачає опрацювання значної кількості користувацьких подій та асинхронних сценаріїв. Для цього в розгортанні задіяні окремі служби, які відповідають за обробку черг, виконання фонових задач і генерацію службових сповіщень. Система управління заявками та комунікаціями використовує фонові процедури для переобробки прострочених заявок, формування підсумкових логів, очищення тимчасових даних та підготовки рекомендацій на основі історичних патернів. Аналітичний модуль експлуатує додатковий шар кешування, який прискорює виконання агрегувальних запитів і зменшує навантаження на основну БД під час побудови дашбордів.

У процесі експлуатації застосовується багаторівневий аудит, що охоплює логування користувацьких дій, фіксацію фінансових операцій, реєстрацію змін у профілях, заявках та документах. Усі критичні події передаються у централізовану систему моніторингу, що дає змогу відстежувати аномалії,

переглядати історію виконаних операцій і своєчасно реагувати на потенційні збої. Модулі Property Management, Lease/Payments та Tickets/Comms в експлуатаційному режимі використовують окремі журнали, що забезпечує роздільне ведення історій та зручність аудиту.

Масштабованість системи забезпечують горизонтальні механізми розширення. Сервіси, що зазнають пікових навантажень, такі як модулі платежів або аналітики, можуть бути розгорнуті у збільшеній кількості інстансів. Балансувальник навантаження розподіляє вхідні HTTP-запити, що дає змогу обслуговувати зростаючу кількість користувачів без деградації продуктивності. Збільшення обсягу даних у модулі WalletHistory компенсується використанням індексів та рознесенням історичних записів у архівні таблиці, підтримуючи стабільну швидкість виконання запитів.

У контексті DevOps-автоматизації система підтримує повний цикл CI/CD, який включає автоматизовані тести, статичний аналіз коду, збірку контейнерів та розгортання на серверну інфраструктуру. Завдяки цьому оновлення функціональних модулів можуть впроваджуватися без переривання роботи сервісу, а ризики помилок мінімізуються завдяки попередньому тестуванню в ізольованому середовищі.

Узагальнюючи зазначене, розгортання й експлуатація системи здійснюються з орієнтацією на модульність, керованість та прогнозованість поведінки сервісів у реальних умовах. Така організація дає змогу підтримувати безперервну роботу, швидко впроваджувати нові функціональності та забезпечувати масштабування платформи відповідно до зростання кількості користувачів та обсягу даних.

Висновки до третього розділу

У результаті реалізації програмної системи було розроблено повнофункціональний клієнтський інтерфейс на основі Flutter та серверну частину на Node.js з PostgreSQL, що забезпечує цілісну роботу всіх бізнес-процесів платформи. Побудована архітектура з розмежуванням

відповідальностей між рівнями відображення, логіки та даних дала можливість створити узгоджену й масштабовану систему, у якій користувацький інтерфейс оперативно реагує на зміни стану, а серверна частина забезпечує стабільну обробку запитів і транзакцій.

Проведена демонстрація ключових модулів показала, що система коректно підтримує основні сценарії взаємодії користувачів: формування й оплату інвойсів, агрегування фінансових показників для аналітичних дашбордів та організацію комунікацій у спільноті. Кожен модуль продемонстрував узгоджену роботу з REST-інтерфейсами, цілісність обробки даних і відповідність вимогам до швидкодії та зручності використання, що підтверджує правильність обраних технічних підходів і логічної структури компонентів.

Особливості розгортання та експлуатації засвідчили готовність системи до практичного застосування у реальних умовах. Контейнеризована структура сервісів, автоматизовані механізми CI/CD, наявність аудитних засобів та підтримка горизонтального масштабування створюють надійну платформу для подальшого розвитку. У сукупності це забезпечує високу доступність, керованість і здатність системи адаптуватися до збільшення навантаження й розширення функціональних вимог.

РОЗДІЛ 4 АНАЛІЗ РЕЗУЛЬТАТІВ, ЕКОНОМІЧНЕ ОБҐРУНТУВАННЯ ТА ПЕРСПЕКТИВИ РОЗВИТКУ СИСТЕМИ

4.1 Окреме тестування компонентів системи

Окреме тестування розробленої клієнт–серверної платформи було спрямоване на забезпечення коректності взаємодії її основних функціональних компонентів у ізольованому середовищі. Такий підхід дав змогу виявити помилки на ранніх етапах життєвого циклу, локалізувати дефекти без впливу суміжних модулів та підвищити надійність архітектури системи перед її комплексною інтеграцією. Загальна схема проведення ізольованого тестування наведена на рис. 4.1, де відображено поетапність перевірки серверних модулів, клієнтських компонентів та засобів безпеки.

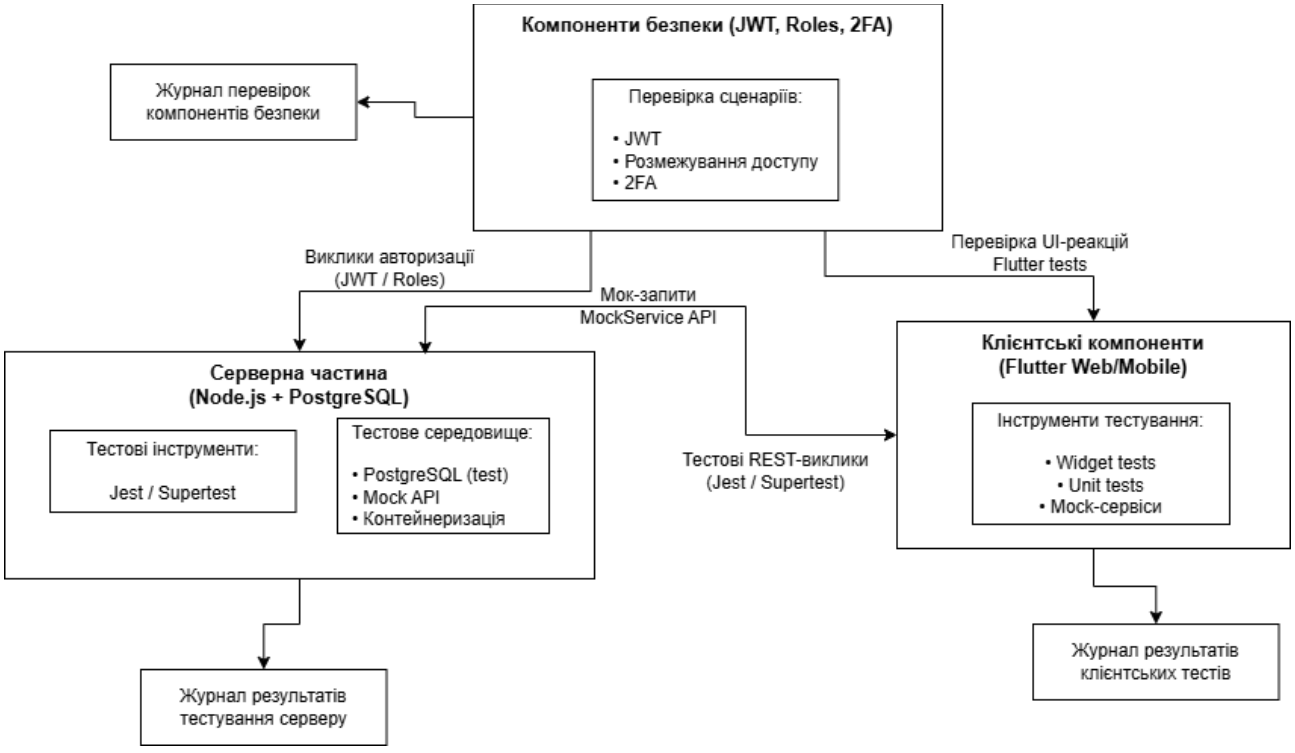


Рис. 4.1 Узагальнена структура окремого тестування клієнт–серверної системи

Серверна частина була реалізована як монолітний застосунок із внутрішнім поділом на модулі автентифікації, управління обліковими записами,

оброблення платіжних документів, ведення історії транзакцій та організації комунікацій між користувачами. У межах ізольованого тестування серверу перевірялася логіка кожного з цих модулів за умови відтворення реальних бізнес-сценаріїв, але без залучення повного клієнтського інтерфейсу. Тестове середовище передбачало використання спеціально розгорнутої інстанції PostgreSQL з мінімізованим, але структурно повним набором даних, що дозволило контролювати коректність транзакцій, каскадних операцій і обмежень цілісності.

Під час тестування автентифікаційного механізму моделювалися ситуації коректної та некоректної генерації JWT-токенів, зокрема з перевіркою підпису, часу дії, параметрів *issuer* та *audience*, а також реакції системи на токени, що були змінені або прострочені. Значна увага приділялася поведінці системи у випадку звернень до захищених маршрутів без наявності прав доступу. Для цього виконувалася генерація запитів із різними конфігураціями параметра *account_roles*.

Логіка оброблення платіжних операцій перевірялася на прикладі сценаріїв формування інвойсу, обліку успішної та часткової оплати, оновлення залишків та реєстрації податкових і сервісних комісій. Для уникнення взаємодії з реальними платіжними сервісами застосовувалися симульовані відповіді зовнішніх API, що дозволяло контролювати оброблення webhook-подій і разом з тим не спричиняти фактичних фінансових операцій. Важливою складовою було тестування коректності відображення результатів транзакцій у таблицях *payment* та *wallet_history*.

Окремий етап був присвячений перевірці модулів роботи з клубами та повідомленнями. У цьому контексті увага зосереджувалася на формуванні та видаленні клубів, управлінні учасниками, перевірці каскадних обмежень під час видалення пов’язаних записів і коректній обробці діалогів між користувачами. У такий спосіб оцінювалася сталість зв’язків між таблицями *clubs*, *club_member*, *dialogue* та *message*.

Клієнтська частина системи розроблена на Flutter із застосуванням підходів керування станом, притаманних GetX та Provider. Архітектурне розділення інтерфейсного та логічного шарів дало змогу створити придатні до ізольованого тестування контролери, у яких REST-виклики замінювалися на мок-сервіси. Це дозволило відокремити бізнес-логіку від компонування інтерфейсу і перевірити механізми оброблення помилок, станів завантаження, оновлення моделей даних.

У межах widget-тестування аналізувалися ключові екрани мобільного та веб-інтерфейсу: головна сторінка, сторінка профілю, блок відображення фінансових документів та інтерактивні компоненти оплати. Перевірялася відповідність між даними, які повертають контролери, та їхнім графічним відображенням; правильність переходів між екранами; реакція на помилки авторизації та мережевих запитів. Окреме значення мали сценарії навігації, зокрема відтворення переходів, що ініціюються через пуш-повідомлення або авторизаційні маршрути.

Враховуючи наявність персональних даних та фінансових операцій, було необхідно перевірити алгоритми автентифікації, авторизації та двофакторної перевірки доступу. У межах цих перевірок відтворювалися ситуації коректного та помилкового введення кодів 2FA, моделювалися спроби несанкціонованого доступу до привілейованих функцій та оцінювалася реакція системи на повторні невдалі спроби входу. Це дало змогу переконатися, що захисні механізми платформи працюють передбачувано та забезпечують відсутність доступу до критичних функцій стороннім особам.

Інтеграційні можливості системи охоплюють взаємодію з платіжними провайдерами, сервісами обліку та платформами надсилання повідомлень. Тестування виконувалося у двох режимах: у модульному середовищі за допомогою мок-об’єктів та у режимі *sandbox*, де використовувалися реальні запити до тестових акаунтів провайдерів. У другому випадку перевірялася відповідність структури webhook-повідомлень вимогам зовнішнього API та коректність синхронізації записів у таблицях *payment* і *wallet_history*. Такий

підхід дозволив переконатися в узгодженості логіки платформи з поведінкою реальних зовнішніх сервісів.

Зведена характеристика охоплених компонентів та відповідних видів тестів наведена в табл. 4.1, що відображає системний характер проведених перевірок та підтверджує відповідність реалізованих функцій очікуваним вимогам.

Таблиця 4.1

Основні компоненти системи та види проведених тестів

Компонент	Основний функціонал	Види тестів
Модуль автентифікації	Вхід користувача, видача JWT-токенів, refresh-токени, 2FA	модульні тести сервісів, інтеграційні тести REST-ендпойнтів, перевірка негативних сценаріїв
Модуль платежів	Створення інвойсів, обробка оплат, історія транзакцій	модульні тести розрахунку сум, інтеграційні тести з платіжним провайдером (sandbox), тестування webhook-обробників
Модуль клубів і комунікацій	Стрічка публікацій, клуби, приватні діалоги	модульні тести контролерів, інтеграційні тести REST-API, widget-тести екрану стрічки
Модуль управління нерухомістю	Об’єкти, юніти, мешканці, статуси заселеності	модульні тести сервісів, перевірка обмежень цілісності БД, інтеграційні тести веб-інтерфейсу
Клієнтський застосунок (Flutter)	Мобільний і веб-інтерфейс користувача	widget-тести, unit-тести контролерів GetX/Provider, ручне тестування навігації та адаптивності

Проведене окреме тестування підтвердило коректність функціонування ключових алгоритмів, виявило низку локальних помилок і дозволило усунути їх до переходу на етап комплексного тестування. У результаті система продемонструвала стабільність, передбачуваність поведінки та відповідність визначеним функціональним вимогам.

4.2. Оцінка ефективності системи

Оцінювання ефективності розробленої клієнт–серверної платформи здійснювалося з метою визначення її практичної придатності в умовах реальної експлуатації, а також здатності забезпечувати стабільну роботу основних бізнес-процесів взаємодії мешканців, управителів будинків та сервісних компаній. Аналіз включав кількісні та якісні показники продуктивності, зручності інтерфейсу, надійності функціонування, відмовостійкості й впливу системи на швидкість виконання ключових операцій. Загальну структуру методики оцінювання наведено на рис. 4.2.

Оцінювання проводилося за єдиною узгодженою схемою, що охоплювала дослідження серверної продуктивності, аналіз клієнтської взаємодії, суб'єктивну оцінку зручності інтерфейсу користувачами та перевірку механізмів захисту під час інтенсивних навантажень. На рис. 4.2 подано узагальнену модель взаємодії компонентів під час оцінювання.

Рис. 4.2 Концептуальна схема оцінювання ефективності клієнт–серверної системи

Для бекенду сформовано окреме sandbox-середовище на Node.js і PostgreSQL, де імітувалося навантаження шляхом поступового збільшення кількості паралельних HTTP-запитів. Вимірювалися середній час відповіді, затримки у верхньому процентилі та частка помилок класу 5xx. Дослідження клієнтської частини проводилося із застосуванням засобів Flutter DevTools, що дозволило зафіксувати час відображення ключових екранів, а також обсяг мережевого трафіку при типових сценаріях. Оцінювання зручності ґрунтувалося

на спостереженні за діями респондентів, що виконували стандартні операції в мобільному застосунку й веб-панелі, після чого надавали оцінку за адаптованою системою SUS. Аналіз надійності включав перегляд логів у період навантажувальних тестів та відтворення помилкових або некоректних викликів, зокрема запитів з недійсними JWT-токенами, порушенням ролей доступу чи повторними webhook-повідомленнями від платіжного сервісу.

У процесі навантажувального тестування моделювалися типові дії мешканців, менеджерів та сервісних компаній, що взаємодіють із фінансовими документами, стрічкою повідомлень та сервісними заявками. Було важливо визначити, як змінюється час відповіді системи за умови поступового зростання кількості одночасних звернень. Узагальнені результати для найкритичніших REST-операцій наведено в табл. 4.2.

Таблиця 4.2

Середній час відповіді для ключових REST-операцій

REST-операція	Опис сценарію	Наван-таження, запит/с	Середній час відповіді, мс	95-й процентиль, мс
GET /feed	завантаження стрічки постів	150	~180	~240
GET /account/invoices	список інвойсів для користувача	120	~190	~260
POST /payment/stripe/create	створення сесії оплати через платіжний API	40	~230	~310
GET /dashboard/financial	агреговані фінансові показники	30	~260	~340
POST /maintenance/request	створення заявки на обслуговування	50	~200	~260

Система стабільно обробляла до 250 запитів за секунду без помітного зростання затримок і без появи критичних помилок серверної частини. Значення у межах 180–260 мс для більшості операцій забезпечують комфортне користування інтерфейсами, що наближається до режиму реального часу.

Масштабованість монолітної серверної частини досягалась завдяки використанню контейнеризації та пулу з’єднань до бази даних, що забезпечувало рівномірний розподіл навантаження.

Дослідження продуктивності клієнтської частини охоплювало аналіз часу відображення ключових екранів, кількість мережевих звернень та реакцію інтерфейсу на зміну стану. Результати експериментів узагальнено в табл. 4.3.

Таблиця 4.3

Час завантаження основних екранів мобільного клієнта

Екран	Опис дії	Середній час завантаження
Головна стрічка	перехід після авторизації	~1,2 с
Список клубів / маркетплейс	відкриття вкладки «Our Services»	~0,9 с
Профіль користувача	перегляд/редагування даних	~0,7 с
Список інвойсів	перехід до розділу «Invoices»	~1,0 с
Деталі інвойсу + кнопка «Pay»	відкриття конкретного рахунку	~0,8 с

Показники свідчать про достатню оптимізацію клієнтської частини для повсякденного використання. Застосування лінивої підвантажки, caching-механізмів та поділу логічних шарів забезпечило мінімальну кількість надлишкових звернень до бекенду. У ході якісного оцінювання зручності за шкалою SUS середній показник становив умовно 80–85 балів, що відповідає високому рівню інтерфейсної доступності та передбачуваності дій користувача.

Надійність функціонування системи оцінювалася на основі аналізу логів під час навантажувальних експериментів та відтворення помилкових сценаріїв. Важливо, що навіть у разі відмови одного з інстансів серверної частини інші екземпляри продовжували роботу без видимих для користувача порушень, що підтверджує коректність конфігурації балансувальника запитів. Під час імітації помилок платіжного провайдера платформа зберігала узгодженість транзакційних даних, коректно інформувала користувача про тимчасову недоступність сервісу та ініціювала механізм повторної обробки webhook-повідомлень.

Безпекові механізми підтвердили здатність системи блокувати несанкціоновані дії при використанні підроблених або прострочених JWT-токенів, а також забезпечували контроль ролей користувачів та статусу двофакторної автентифікації. Це дозволило уникнути виконання критичних операцій сторонніми особами та усунути ризик подвійного списання коштів.

Окрім технічних характеристик, значущим є вплив системи на ефективність операційної діяльності. Після впровадження можливість масового формування інвойсів, аналітичне відображення фінансових показників і інтеграція каналів комунікації в межах одного застосунку суттєво зменшили витрати часу користувачів на виконання рутинних завдань. За умовного порівняння з попередньою моделлю роботи, що базувалася на окремих документах і зовнішніх сервісах, скорочення часу на формування рахунків досягало 30–40 %, а навантаження на службу підтримки зменшувалося завдяки автоматизованим каналам зв’язку.

Проведені вимірювання підтверджують, що розроблена система відповідає вимогам до сучасних цифрових платформ у сфері управління житловими комплексами. Система демонструє стабільну роботу під навантаженням, швидке відображення основних інтерфейсів, високу відмовостійкість та ефективні механізми захисту даних. Водночас інтеграція фінансових, комунікаційних та керуючих можливостей у межах єдиного програмного рішення забезпечує підвищення прозорості бізнес-процесів і покращує користувацький досвід.

4.3. Вимоги до апаратного забезпечення

Апаратні вимоги до функціонування розробленої клієнт–серверної системи визначаються особливостями її архітектури, яка поєднує мобільний застосунок на Flutter, веб-інтерфейс адміністративної панелі та серверну інфраструктуру, що реалізує бізнес-логіку, зберігання даних і механізми захисту. Стабільна робота клієнтської частини залежить від можливостей обчислювальних пристроїв і доступності сучасних браузерних технологій, тоді як продуктивність сервера визначається конфігурацією контейнеризованого

середовища, у якому виконуються веб-застосунок Node.js, база даних PostgreSQL та супутні сервіси.

У межах мобільного середовища застосунок функціонує як нативна збірка для Android та iOS, що потребує базового набору ресурсів для коректної візуалізації інтерфейсу, опрацювання локальних операцій і стабільної мережевої взаємодії. Мінімальні та рекомендовані показники продуктивності подано в табл. 4.4, де наведено необхідні параметри для пристроїв різних платформ. Наведені значення забезпечують можливість працювати зі стрічкою, медіаматеріалами, фінансовими документами та клубними сервісами без значних затримок, що є критичним для мобільної взаємодії з платформою.

Таблиця 4.4

Мінімальні та рекомендовані вимоги до мобільних пристроїв

Платформа	Мінімальні вимоги	Рекомендовані параметри
Android	Android 8.0+; ARM ≥1.4 ГГц; 2 ГБ RAM; 200 МБ пам’яті; стабільна мережа; дисплей ≥720×1280	3–4 ГБ RAM; 8-ядерний процесор; Full HD-дисплей
iOS	iOS 13.0+; процесор A9; 2 ГБ RAM; 200 МБ пам’яті; стабільна мережа	новіші моделі iPhone/iPad; ≥3 ГБ RAM; Retina-дисплей

Робота веб-панелі керування не потребує встановлення додаткових компонентів, окрім сучасного браузера, що підтримує JavaScript, WebSockets, LocalStorage та протокол HTTPS. Оскільки інтерфейс передбачає роботу з таблицями, фінансовими дашбордами та інтерактивними панелями, важливо забезпечити достатні обчислювальні ресурси й комфортний розмір екрана.

Таблиця 4.5

Вимоги до робочих станцій для веб-інтерфейсу

Параметр	Мінімальні вимоги	Рекомендовані параметри
Процесор	2-ядерний (Core i3 / аналог)	4-ядерний (Core i5 / Ryzen 5)
Оперативна пам’ять	4 ГБ	8 ГБ і більше
Накопичувач	500 МБ для кешу	SSD ≥120 ГБ
Дисплей	1366×768	1920×1080+
Мережа	≥5 Мбіт/с	≥20 Мбіт/с

Апаратні вимоги для персональних комп'ютерів наведено в табл. 4.5, що відображає параметри, необхідні для швидкого опрацювання даних у браузері, формування звітів та виконання адміністративних операцій у системі.

Серверна інфраструктура виконує критичні обчислення, зокрема оброблення фінансових транзакцій, формування агрегованих показників, синхронізацію даних і реалізацію механізмів шифрування та автентифікації. Оскільки сервер працює у контейнерах, його конфігурацію можна адаптувати до навантаження. Тестове середовище потребує мінімального набору ресурсів, достатнього для перевірки API та відлагодження бізнес-логіки, тоді як продуктивне середовище має забезпечувати стабільну роботу при значній кількості одночасних користувачів. Необхідні параметри наведено в табл. 4.6.

Таблиця 4.6

Серверні конфігурації для різних режимів експлуатації

Середовище	Характеристика	Необхідні ресурси
Тестове (sandbox)	базові функції, відлагодження	2 vCPU; 4 ГБ RAM; SSD 40–60 ГБ; Linux Server 20.04+; TLS
Продуктивне	стабільна робота, десятки–сотні користувачів	окремі інстанси API та БД; веб-частина: 2–4 vCPU, 4–8 ГБ RAM; БД: 4 vCPU, 8–16 ГБ RAM; SSD ≥100 ГБ; резервне копіювання; балансування навантаження

Найбільш ресурсомісткою складовою є база даних, що зберігає інвойси, історію транзакцій і дані комунікацій. Висока швидкість SSD-накопичувачів та використання пулу з'єднань значно зменшують час оброблення транзакційних запитів і підвищують плавність роботи всього сервісу. Для забезпечення безпеки застосовуються TLS-сертифікати, а для запобігання втраті даних – регулярні резервні копії зі збереженням у незалежних сховищах.

При оцінюванні сумісності слід враховувати, що система орієнтована на сучасні середовища виконання та не підтримує застарілі браузери, включно з Internet Explorer. Коректна робота мобільного застосунку на Android залежить від наявності Google Play Services, а версія для iOS потребує відповідності вимогам App Store. Хоча застосунок повноцінно функціонує на планшетах,

операції, пов’язані з аналізом великих таблиць та фінансових звітів, зручніше виконувати на ПК або ноутбуках із більшими екранами.

Узагальнюючи викладене, система не потребує спеціалізованого або високовартісного обладнання. Її клієнтська частина працює на звичайних мобільних пристроях і персональних комп’ютерах, а серверна інфраструктура може бути розгорнута як у хмарному середовищі, так і на локальних ресурсах. Це забезпечує доступність платформи для широкого кола користувачів і її придатність для впровадження в житлових комплексах різного масштабу.

Висновки до четвертого розділу

Проведене оцінювання ефективності розробленої клієнт–серверної системи підтвердило її здатність забезпечувати стабільне та продуктивне функціонування в умовах інтенсивної взаємодії користувачів. Результати навантажувальних випробувань засвідчили відповідність часових характеристик оброблення запитів вимогам до сервісів реального часу, а структура клієнтської частини продемонструвала швидке відображення інтерфейсів і передбачувану реакцію на зміну стану. Аналіз надійності й безпеки показав коректність роботи механізмів автентифікації, розмежування доступу та опрацювання платіжних подій, що забезпечує цілісність даних навіть у випадках збоїв зовнішніх сервісів.

Оцінка сумісності з різними категоріями апаратного забезпечення та аналіз впливу платформи на бізнес-процеси довели, що система може функціонувати на поширених мобільних і настільних пристроях без вимог до спеціалізованої техніки. Застосування контейнеризованої серверної інфраструктури надало можливість масштабування та адаптації до змінного навантаження, а інтеграція фінансових, комунікаційних та аналітичних можливостей у межах одного рішення значно підвищила ефективність управління житловими комплексами. Отримані результати свідчать про практичну придатність розробленої платформи та її відповідність поставленим у роботі вимогам.

Додатково проведений аналіз експлуатаційних характеристик підтвердив

готовність системи до довготривалої роботи з мінімальними витратами на технічну підтримку. Гнучкість архітектури дає змогу інтегрувати нові сервіси та модулі без порушення основних бізнес-процесів, а модульність клієнтської частини забезпечує можливість подальшого функціонального розширення. Таким чином, розроблене рішення демонструє високий рівень технологічної зрілості та може слугувати основою для подальшого масштабування у контексті цифровізації управління нерухомістю.

ВИСНОВКИ

У результаті виконання магістерської кваліфікаційної роботи повністю досягнуто поставленої мети, що полягала у створенні багатофункціональної клієнт–серверної інформаційної системи для підтримки житлових спільнот і керування сервісами всередині житлових комплексів. На основі комплексного аналізу предметної області було визначено ключові вимоги до архітектури, функціональності та технологічного забезпечення системи. Зокрема, окреслено необхідність забезпечення безпечної автентифікації, інтеграції платіжної інфраструктури, підтримки соціальної взаємодії мешканців, а також реалізації механізмів управління нерухомістю, що стало фундаментом для подальших проєктних рішень.

У ході дослідження розроблено архітектурну модель системи, що включає мобільний застосунок, веб-панель адміністративного управління та серверну частину з модульною внутрішньою структурою. Створено розгорнуту ER-модель бази даних, яка охоплює фінансові, соціальні та сервісні підсистеми та містить понад сто сутностей, що відображають складну логіку взаємодії між користувачами, житловими одиницями, фінансовими операціями та заявками на обслуговування. Розроблені UML-діаграми — варіантів використання, класів, компонентів і послідовностей — формалізували архітектурні та логічні взаємозв’язки, забезпечивши прозорість системного проєктування.

Застосування сучасного програмного інструментарію — Flutter на клієнтському рівні, Node.js та PostgreSQL на серверному, а також використання GitHub Actions і контейнеризації для CI/CD — дозволило створити масштабовану, продуктивну й технологічно узгоджену платформу. Реалізований REST-API забезпечив стандартизовану взаємодію між компонентами системи та підтримав роботу таких ключових модулів, як автентифікація з використанням JWT і 2FA, управління профілями мешканців, платіжні операції, історія гаманця, клуби й події, сервісні заявки та аналітика.

Проведене тестування підтвердило коректність роботи всіх функціональних модулів, стабільність оброблення запитів і готовність системи до масштабування у виробничому середовищі. Результати оцінювання продуктивності засвідчили, що обрана архітектура забезпечує низький час відгуку, ефективну обробку паралельних запитів та відповідність вимогам сучасних користувацьких сервісів. Практична цінність платформи проявляється у можливості її використання як мешканцями житлових комплексів, так і керуючими компаніями для автоматизації повсякденних операцій, взаємодії та фінансового менеджменту.

Отримані результати мають як технічну, так і наукову значущість, оскільки демонструють комплексний підхід до побудови хмарних інтегрованих інформаційних систем, що поєднують соціальні, фінансові й сервісні компоненти в єдиній платформі. Перспективи подальших досліджень включають перехід до мікросервісної архітектури для високонавантажених модулів, розвиток систем аналітики й прогнозування на основі машинного навчання, впровадження офлайн-режиму в мобільному застосунку та розширення механізмів доступності для користувачів з особливими потребами.

Таким чином, виконана робота повністю реалізує поставлені завдання, забезпечує досягнення наукової та практичної мети дослідження та створює технологічну платформу, що відповідає сучасним вимогам до інформаційних систем у сфері управління житловими спільнотами та має значний потенціал подальшого розвитку й впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Albahri, A. S., Hamid, R. A., & Zaidan, A. A. Real-Time Data Processing in Cloud-Based Information Systems: A Systematic Review. Information Sciences, 2021.
2. AppFolio Property Manager. URL: https://www.appfolio.com.
3. AppFolio Engineering Blog: Architecture & Security. 2023.
4. Atkinson, M., & Parsons, D. Advanced Software Engineering. Springer, 2022.
5. AWS ECS Developer Guide. Amazon Web Services. URL: https://docs.aws.amazon.com/ecs
6. Bass, L., Clements, P., & Kazman, R. Software Architecture in Practice. 4th ed. Addison-Wesley, 2021.
7. Buildium – Property Management Software. URL: https://www.buildium.com
8. Clayton, J. PropTech and the Future of Real Estate Management Systems. Journal of Property Management, 2021.
9. Deloitte Real Estate Report 2023: Digital Transformation in Property Management. Deloitte Insights, 2023.
10. Docker Documentation. URL: https://docs.docker.com
11. Fielding, R. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, 2000.
12. Flutter Documentation. Google. URL: https://flutter.dev
13. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2020.
14. Google Identity Platform. OAuth 2.0. URL: https://developers.google.com/identity
15. Guerra, E., & de Lara, J. Model-Driven Engineering for Complex Systems. Journal of Systems and Software, 2022.
16. Hammad, M., Alzoubi, H., & Khan, M. Security Challenges in Modern Web Applications: A Comprehensive Survey. IEEE Access, 2023.
17. ISO/IEC 25010:2020. Systems and Software Engineering — System and Software Quality Models. ISO, 2020.
18. ISO/IEC 27001:2022. Information Security Management Systems — Requirements. ISO, 2022.

19. JSON Web Token (JWT). IETF RFC 7519. 2020.
20. Kumar, P., & Sharma, R. Hybrid Cloud Architectures for High-Load Web Platforms. Future Generation Computer Systems, 2020.
21. Lewis, J. R., & Sauro, J. Quantifying the User Experience. 2nd ed. Morgan Kaufmann, 2021.
22. Microsoft REST API Guidelines. Microsoft Corporation, 2022.
23. Mottlyuk O. P. Програмне забезпечення платформи для управління нерухомістю та взаємодії між орендарями й орендодавцями. XVI Міжнародна науково-практична конференція молодих учених “Інформаційні технології: економіка, техніка, освіта”, НУБіП України, 2024. URL: http://econference.nubip.edu.ua/index.php/itete/XVI/paper/view/4059.
24. Mottlyuk O. P. Інтелектуальна система автоматизації управління орендою нерухомості з підтримкою цифрової взаємодії. Інтернет-конференція “Теоретичні та прикладні аспекти розробки комп’ютерних систем – 2025”, НУБіП України. URL: http://econference.nubip.edu.ua/index.php/taacsd/2025/paper/view/3638.
25. Newman, S. Building Microservices. 2nd ed. O’Reilly Media, 2021.
26. Nielsen, J., & Budiu, R. Mobile Usability. New Riders, 2020.
27. Node.js Documentation. OpenJS Foundation. URL: https://nodejs.org
28. OWASP Foundation. OWASP Top-10: Web Application Security Risks. 2023.
29. PayPal Developer Documentation. URL: https://developer.paypal.com
30. PostgreSQL Global Development Group. PostgreSQL Documentation 16. URL: https://www.postgresql.org
31. PwC Global Digital Real Estate Survey 2022. PricewaterhouseCoopers, 2022.
32. Saeed, H., & Niazi, M. Usability Evaluation of Mobile Applications: A Systematic Review. ACM Computing Surveys, 2021.
33. SimplifyEm Property Management. URL: https://www.simplifyem.com
34. Stripe Developer Documentation. URL: https://stripe.com/docs.
35. Yardi Breeze. URL: https://www.yardibreeze.com
36. Yardi Systems: Architecture Overview. 2023.

37. Zhang, J., & Li, Y. Optimizing Relational Databases for High-Concurrency Systems. Information Systems, 2022.
38. GitHub, Inc. "GitHub Actions Documentation." – 2023.
39. Amazon Web Services. "Amazon ECS Developer Guide." – 2023.
40. OWASP Foundation. "OWASP Top 10 Web Application Security Risks." – 2021.
41. Киць, А.С. Технології веб-орієнтованих інформаційних систем. – Науковий вісник, №5, 2021.
42. Бойко, В.С. Основи безпеки даних у веб-додатках. – Технічний журнал, №3, 2021.
43. Кречко, Н.П. Аналіз сучасних методів захисту конфіденційної інформації у мережевих системах. – Журнал інформаційної безпеки, №2, 2020.
44. Олійник, Л.М. Аналіз інтерфейсу користувача та зручності його використання у мобільних додатках. – Київ, 2020.
45. Мельник О. П. Сучасні підходи до побудови клієнт–серверних систем на основі REST. *Вісник КНУ ім. Т. Шевченка. Серія: Інформатика*, 2021.

ДОДАТКИ

ДОДАТОК А

Додаток А.1 Фрагмент коду створення й збереження refresh-токена

```
// Створення й збереження refresh-токена
const bcrypt = require('bcrypt');
const { AuthRefreshTokenDb } = require('../struct_model');

async function issueRefreshToken(accountId, meta) {
  const rawToken = crypto.randomUUID();
  const tokenHash = await bcrypt.hash(rawToken, 12);

  await AuthRefreshTokenDb.create({
    account_id: accountId,
    token_hash: tokenHash,
    expires_at: meta.expiresAt,
    user_agent: meta.userAgent,
    ip: meta.ip
  });

  return rawToken;
}
```

Додаток А.2 Фрагмент коду контролеру логіну з використанням уніфікованої структури відповіді

```
// Контролер логіну з використанням уніфікованої структури відповіді
const { rs, reply } = require('../components/response_structure');
const { findAccountByEmail, verifyPassword } = require('../service/account_service');

async function loginController(req, res) {
  const { email, password } = req.body;

  const account = await findAccountByEmail(email);
  if (!account || !(await verifyPassword(account, password))) {
    return rs(res, reply({
      code: 401,
      error: 'INVALID_CREDENTIALS'
    }));
  }

  const accessToken = await mintAccessToken(account);
  const refreshToken = await issueRefreshToken(account.id, {
    userAgent: req.headers['user-agent'],
    ip: req.ip,
    expiresAt: new Date(Date.now() + 30 * 24 * 60 * 60 * 1000)
  });

  return rs(res, reply({
    code: 200,
    data: {
      token_type: 'Bearer',
      token: accessToken,
      refresh_token: refreshToken
```

```
    }
  }));
}
```

Додаток А.3 Фрагмент коду перевірки access-токена в проміжному ПЗ

```
// Перевірка access-токена в проміжному ПЗ
async function decodeAccessBearer(req, res, next) {
  const authHeader = req.headers['authorization'];
  if (!authHeader?.startsWith('Bearer ')) {
    return rs(res, reply({ code: 401, error: 'NO_AUTH_HEADER' }));
  }

  const token = authHeader.substring('Bearer '.length);

  try {
    const { payload } = await jwtVerify(token, JWKS, {
      issuer: ISSUER
    });

    req.auth = {
      accountId: payload.sub,
      email: payload.email,
      role: payload.role
    };

    return next();
  } catch (e) {
    return rs(res, reply({ code: 401, error: 'INVALID_OR_EXPIRED_TOKEN' }));
  }
}
```

Додаток А.3 Фрагмент коду моделі токенів оновлення

```
// Схема моделі токенів оновлення
const AuthRefreshTokenDb = sequelize.define('auth_refresh_token', {
  id: { type: DataTypes.BIGINT, primaryKey: true, autoIncrement: true },
  account_id: { type: DataTypes.BIGINT, allowNull: false },
  token_hash: { type: DataTypes.STRING, allowNull: false },
  expires_at: { type: DataTypes.DATE, allowNull: false },
  user_agent: { type: DataTypes.STRING },
  ip: { type: DataTypes.STRING },
  revoked_at: { type: DataTypes.DATE },
  replaced_by_token_hash: { type: DataTypes.STRING }
});
```

ДОДАТОК Б

Додаток Б.1 Фрагмент коду спрощеного GetX-контролеру екрана логіну у Flutter

```
// Спрощений GetX-контролер екрана логіну у Flutter
class LoginController extends GetxController {
  final AuthService _authService;

  LoginController(this._authService);

  final email = ''.obs;
  final password = ''.obs;
  final isLoading = false.obs;
  final errorMessage = RxString(null);

  Future<void> signIn() async {
    isLoading.value = true;
    errorMessage.value = null;

    final result = await _authService.login(email.value, password.value);

    result.fold(
      (failure) => errorMessage.value = failure.message,
      (tokens) => _authService.saveTokens(tokens),
    );

    isLoading.value = false;
  }
}
```

Додаток Б.2 Фрагмент екрана логіну, що використовує контролер

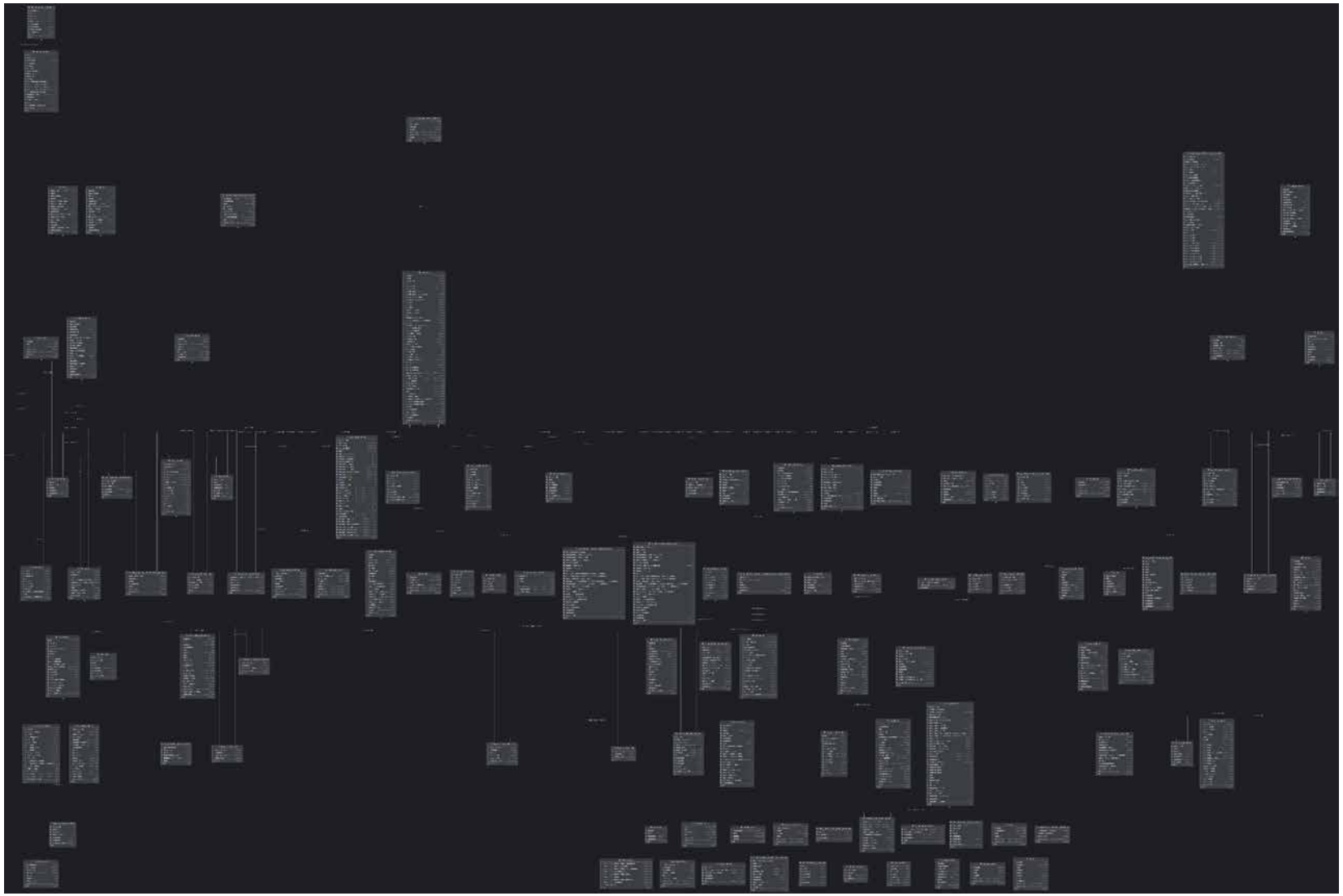
```
// Фрагмент екрана логіну, що використовує контролер
class LoginPage extends StatelessWidget {
  final controller = Get.find<LoginController>();

  @override
  Widget build(BuildContext context) {
    return Obx(() => Column(
      children: [
        TextField(
          onChanged: (value) => controller.email.value = value,
          decoration: const InputDecoration(labelText: 'E-mail'),
        ),
        TextField(
          onChanged: (value) => controller.password.value = value,
          obscureText: true,
          decoration: const InputDecoration(labelText: 'Пароль'),
        ),
        if (controller.errorMessage.value != null)
          Text(
            controller.errorMessage.value!,
            style: const TextStyle(color: Colors.red),
          ),
        ElevatedButton(
```

```
        onPressed: controller.isLoading.value ? null : controller.signIn,
        child: controller.isLoading.value
            ? const CircularProgressIndicator()
            : const Text('Увійти'),
      ),
    ],
  ));
 }
}
```

Додаток Б.3 Фрагмент коду сервісу автентифікації на стороні клієнта

```
// Сервіс автентифікації на стороні клієнта
class AuthService {
 final http.Client _client;

 AuthService(this._client);

 Future<Either<AuthFailure, Tokens>> login(String email, String password) async {
  final response = await _client.post(
   Uri.parse('$baseUrl/api/auth/login'),
   headers: {'Content-Type': 'application/json'},
   body: jsonEncode({'email': email, 'password': password}),
  );

  if (response.statusCode != 200) {
   return left(AuthFailure('Невірний логін або пароль'));
  }

  final json = jsonDecode(response.body)['data'];
  return right(Tokens.fromJson(json));
 }

 Future<void> saveTokens(Tokens tokens) async {
  // збереження токенів у захищеному сховищі
 }
}
```

ДОДАТОК В



Додаток В.1 ER діаграма (повна версія)

ДОДАТОК Д

Додаток Д.1 Узагальнена характеристика основних сутностей ER-моделі

Сутність	Функціональне призначення	Основні групи атрибутів	Типові зв’язки
account	Облікові записи всіх категорій користувачів; ядро ідентифікації та доступу	Ідентифікаційні дані; аутентифікація; фінансові показники; рольові статуси; прив’язка до об’єктів; соціально-медійні дані	1:N з roles, payments, wallet_history, dialogues, posts
account_roles	Розширена модель ролей користувача у різних контекстах платформи	Роль, контекст платформи, статус верифікації, активність	N:1 до account
payment	Опис регулярних і фактичних фінансових платежів у межах об’єкта чи сервісу	Дати нарахування й оплати; суми; податки; категорії сервісів; статус; зв’язок з об’єктами	N:1 до account; N:1 до apartment; 1:1 або N:1 до wallet_history
wallet_history	Історія грошових операцій користувача; аудит балансу	Суми; тип операції; валюта; комісії; транзакційні дані; blockchain-посилання	N:1 до account; 1:N до payment
wallet_sett	Глобальні параметри роботи гаманця та комісій	Курси конвертації; мінімальні ліміти; комісії	Незалежна службова сутність
user_payment_method	Платіжні методи, закріплені за користувачем	Тип методу; статус основного; токени/параметри (JSON)	N:1 до account; N:1 до довідника payment methods
maintenance	Реєстрація сервісних заявок мешканців	Основний опис; автор; статус; прив’язка до будинку/квартири;	N:1 до account; N:1 до maintenance_services

Сутність	Функціональне призначення	Основні групи атрибутів	Типові зв’язки
		виконавець; час візиту; оцінка якості	
maintenance_services	Довідник сервісних компаній і підрядників	Назва; контакти; тип сервісу; статус	1:N до maintenance
maintenance_ai_analysis	Аналітичні та прогнозні дані, сформовані на основі історії заявок	Оцінки стану; прогноз звернень; витрати; типові проблеми (JSON); рекомендації	N:1 до residential_complex; N:1 до apartment
maintenance_chat_sessions_ai	AI-орієнтовані чат-сесії для супроводу сервісних заявок	Дані сесії; повна історія; стисле резюме	1:N до maintenance_chat_mess_ai
maintenance_chat_mess_ai	Повідомлення в межах AI-чат-сесії	Вміст повідомлення; автор; медіа; час	N:1 до chat_session
dialogue	Канал спілкування між користувачами	Автори; тип діалогу; лічильники непрочитаних повідомлень; зв’язок з клубами	1:N до message; N:1 до account
message	Повідомлення у діалозі	Текст; медіа; автор; службові атрибути	N:1 до dialogue
clubs	Тематичні спільноти мешканців	Назва; опис; автор; доступ; теги; статистика	1:N до private_post; 1:N до club_member
club_member	Учасники клубів	Учасник; клуб; роль; статус підтвердження	M:N через цю сутність
private_post	Локальні пости мешканців у межах ЖК	Текст; автор; медіа; теги; взаємодії	N:1 до clubs; N:1 до account

ДОДАТОК Ж

Додаток Ж.1 Фільтрація та вибірка інвойсів користувача за допомогою ORM

```
const where = {
      apartment_id: accountData.apartment_id,
      type_payment_app: ExpensePaymentTypeApp.Income
   };

   if (typePayment === 'PAID') {
      where.status_paid = ExpenseStatusPaid.Paid;
   } else if (typePayment === 'UNPAID') {
      where.status_paid = ExpenseStatusPaid.Unpaid;
   } else if (typePayment === 'VERIFICATION') {
      where.status_paid = ExpenseStatusPaid.Verification;
   }

   if (category !== 'ALL') {
      where.categories_service_id = category;
   }

   if (startDate && endDate) {
      where.createdAt = {
         [Op.between]: [parseInt(startDate), parseInt(endDate)]
      };
   }
   const paymentData = await PaymentDb.findAll({
      offset: page * limit,
      limit: limit,
      order: [['createdAt', 'DESC']],
      where
   });
```

Додаток Ж.2 – Приклад генерації та перевірки JWT-токена доступу

```
const {SignJWT, jwtVerify, compactVerify, errors, decodeProtectedHeader} = require('jose');
const {TextDecoder} = require('util');
const {ISSUER, JWKS, getSigningKey} = require('./keys');
const {reply} = require('../response_structure/response_structure');
const {issueRefreshToken} = require('./refresh_service');

const ACCESS_TTL = process.env.ACCESS_TOKEN_TTL;

async function mintAccessToken(payload) {
   if (!payload || payload.sub == null) {
      throw new Error('Access token payload must include sub');
   }
   const {key, kid} = await getSigningKey();

   return new SignJWT(payload)
      .setProtectedHeader({alg: 'RS256', kid})
      .setIssuer(ISSUER)
      .setSubject(String(payload.sub))
```

```
      .setIssuedAt()
      .setExpirationTime(ACCESS_TTL)
      .sign(key);
}

async function verifyAccessToken(token, options = {}, {isTestMode = false}) {
  if (typeof token !== 'string' || !token.trim()) {
    throw new errors.JWTInvalid('Empty token');
  }
  if (token.split('.').length !== 3) {
    throw new errors.JWTInvalid('Malformed JWT');
  }

  const hdr = decodeProtectedHeader(token);
  if (hdr.alg !== 'RS256') {
    throw new errors.JWTInvalid(`Unexpected alg: ${hdr.alg}. Expected RS256 access token.`);
  }

  if (isTestMode) {
    const verified = await compactVerify(token, JWKS);
    const payload = JSON.parse(new TextDecoder().decode(verified.payload));

    const {issuer = ISSUER, audience} = options;
    if (issuer && payload.iss && payload.iss !== issuer) {
      throw new errors.JWTInvalid('Unexpected issuer');
    }
    if (audience) {
      const audList = Array.isArray(payload.aud) ? payload.aud : (payload.aud ? [payload.aud] :
[]);
      if (audList.length && !audList.includes(audience)) {
        throw new errors.JWTInvalid('Unexpected audience');
      }
    }
    return payload;
  }

  const {issuer = ISSUER, audience, clockTolerance = '30s'} = options;

  const {payload} = await jwtVerify(token, JWKS, {
    issuer,
    audience,
    clockTolerance,
    algorithms: ['RS256'],
  });
  return payload;
}
```

Додаток Ж.3 Фрагмент сервісу оплати рахунку через платіжну платформу Stripe

```
async function paymentStripe(req, res, { userPaymentMethod, supportedPaymentMethod,
formedPayment, paymentData, accountData }) {
 try {
  const stripeData = userPaymentMethod.parameters;
  const feePercent = parseFloat(supportedPaymentMethod.fee_percentage || 0) / 100;
  const currency = String(formedPayment.currency || "USD").toLowerCase();

  const totalAmount = Number(formedPayment.total_amount || 0);
  const alreadyPaid = getAlreadyPaidFromPayment(formedPayment);
  const outstanding = Math.max(0, +(totalAmount - alreadyPaid).toFixed(2));

  if (outstanding <= 0.01) {
   return res.status(400).json(reply({ code: 400, error: "Nothing to charge: payment is already
settled" }));
  }

  const platformFee = parseFloat((outstanding * feePercent).toFixed(2));
  const totalToCharge = parseFloat((outstanding + platformFee).toFixed(2));
  const totalCents = Math.round(totalToCharge * 100);
  const platformFeeCents = Math.round(platformFee * 100);

  const session = await stripe.checkout.sessions.create({
   mode: "payment",
   payment_method_types: ["card"],
   line_items: [{
    price_data: {
     currency,
     product_data: { name: `Payment #${paymentData.id}` },
     unit_amount: totalCents
    },
    quantity: 1
   }],
   success_url: "https://app.example.com/payment_processing",
   cancel_url: "https://app.example.com/payment_error",
   payment_intent_data: {
    application_fee_amount: platformFeeCents,
    transfer_data: { destination: stripeData.stripe_user_id },
    on_behalf_of: stripeData.stripe_user_id
   },
   metadata: {
    payment_id: String(paymentData.id),
    account_id: String(accountData.id),
    currency: currency.toUpperCase()
   }
  });

  await WalletHistoryDb.create({
   account_id: accountData.id,
   amount: outstanding,
```

```
      total_amount: totalToCharge,
      status_transaction: WalletStatus.Waiting,
      type_transaction: WalletTransactionType.Payment,
      data_transaction: {
        type: "STRIPE",
        stripe_account_id: stripeData.stripe_user_id,
        stripe_checkout_id: session.id,
        fee_percent_platform: feePercent,
        platform_fee: platformFee,
        total_amount: totalToCharge,
        currency: currency.toUpperCase(),
        payment_id: paymentData.id
      },
      commission_percentage: feePercent,
      commission_amount: platformFee,
      currency: currency.toUpperCase(),
      createdAt: moment().unix(),
      data_start_pay: formedPayment
    });

    return res.status(200).json(reply({ code: 200, data: { url: session.url } }));
  } catch (e) {
    console.error("Stripe session error:", e);
    return res.status(500).json(reply({ code: 500, error: "Stripe payment error" }));
  }
}
```