ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Інтелектуальна система моніторингу та бронювання паркувальних місць у міському середовищі»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Владислав НАКОНЕЧНИЙ
(підпис)

Виконав: Здобувач вищої освіти групи ТЦР-44
Владислав НАКОНЕЧНИЙ

Керівник: Ірина ГНІДЕНКО
ст. викладач

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
«____ » _____2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Наконечному Владиславу Володимировичу

1. Тема кваліфікаційної роботи: «Розробка інтелектуальної системи моніторингу та бронювання паркувальних місць у міському середовищі»
керівник кваліфікаційної роботи старший викладач кафедри Ірина ГНІДЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про архітектуру сучасних клієнт-серверних систем, опис методів моніторингу та управління об'єктами в реальному часі, технічна документація щодо використання стеку технологій та СУБД для зберігання даних про паркувальні місця.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд та аналіз існуючих методів та інтелектуальних технологій моніторингу та бронювання паркувальних місць у міському середовищі.
2. Проектування архітектури інтелектуальної системи та розробка вимог до програмного забезпечення для автоматизації процесів паркування.

3. Програмна реалізація, опис функціонування та взаємодії компонентів інтелектуальної системи моніторингу та бронювання паркувальних місць.
4. Тестування системи.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів та існуючих рішень на ринку.
2. Вимоги до програмного забезпечення.
3. Обґрунтування вибору програмних засобів реалізації.
4. Діаграма варіантів використання.
5. Алгоритм роботи інтелектуальної системи та процесу бронювання.
6. Діаграма класів.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання«20» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	23.02-27.02.2026	
2	Аналіз та дослідження існуючих аналогів	28.02-06.03.2026	
3	Обґрунтування вибору технологій (React, Flask, SQLite) та архітектури системи	07.03-13.03.2026	
4	Проектування бази даних та розробка REST API для взаємодії компонентів	14.03-27.03.2026	
5	Розробка користувацького інтерфейсу (фронтенд) та інтеграція інтерактивної карти	28.03-10.04.2026	
6	Реалізація інтелектуальної логіки бронювання та моніторингу зайнятості місць	11.04-24.04.2026	
7	Оформлення роботи: вступ, висновки, реферат	25.04-04.05.2026	
8	Розробка демонстраційних матеріалів	05.05-10.05.2026	
9	Попередній захист роботи	11.05-22.05.2026	

Здобувач вищої освіти _____ Владислав НАКОНЕЧНИЙ

(підпис)

Керівник

кваліфікаційної роботи _____ Ірина ГНІДЕНКО

(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 105 стор., 8 табл., 22 рис., 58 джерел.

Мета роботи – підвищення ефективності процесу моніторингу та бронювання паркувальних місць за допомогою розробленої інтелектуальної системи на основі клієнт-серверної архітектури та методів аналізу даних.

Об'єкт дослідження – процес моніторингу та управління паркувальними місцями в умовах міського середовища.

Предмет дослідження – методи та програмні засоби побудови інтелектуальної системи моніторингу, бронювання та прогнозування завантаженості паркувальних місць.

Короткий зміст роботи: У роботі проаналізовано існуючі архітектурні підходи та технології побудови систем розумного паркування. Проаналізовано інструментальні засоби та аналоги: SFpark, Parkopedia та муніципальні сервіси. Розроблено алгоритм роботи застосунку та програмно реалізовані ключові функціональні можливості, зокрема: інтерактивна карта з відображенням вільних місць, система особистих кабінетів, функціонал бронювання на конкретний час та інтелектуальний розрахунок вартості послуг. Проведено функціональне та модульне тестування додатку. В роботі використано бібліотеку Leaflet для роботи з картами, React для створення інтерфейсу користувача, Flask для серверної логіки та SQLite для зберігання даних про паркувальні зони та користувачів.

Сферою використання застосунку є організація паркувального простору в межах міських територій, бізнес-центрів та приватних паркінгів для покращення логістики та комфорту водіїв.

КЛЮЧОВІ СЛОВА: ІНТЕЛЕКТУАЛЬНА СИСТЕМА, МОНІТОРИНГ ПАРКУВАННЯ, БРОНЮВАННЯ МІСЦЬ, SMART CITY, ВЕБ-ЗАСТОСУНОК, REACT, FLASK, SQLITE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ANPR – Automatic Number Plate Recognition – автоматичне розпізнавання номерних знаків

API – Application Programming Interface – програмний інтерфейс додатку

BLE – Bluetooth Low Energy – технологія бездротового зв’язку малої потужності

CORS – Cross-Origin Resource Sharing – механізм спільного використання ресурсів між доменами

CRUD – Create, Read, Update, Delete – базові операції над даними

CSS – Cascading Style Sheets – каскадні таблиці стилів

DOM – Document Object Model – об’єктна модель документа

ER – Entity-Relationship – модель «сутність–зв’язок»

HTTP – Hypertext Transfer Protocol – протокол передачі гіпертексту

IoT – Internet of Things – Інтернет речей

JSON – JavaScript Object Notation – формат обміну даними

KPI – Key Performance Indicator – ключовий показник ефективності

LoRaWAN – Long Range Wide Area Network – мережа далекого радіусу дії

NB-IoT – Narrowband Internet of Things – вузькосмуговий IoT

REST – Representational State Transfer – архітектурний стиль вебсервісів

SHA – Secure Hash Algorithm – безпечний алгоритм хешування

SPA – Single Page Application – односторінковий вебзастосунок

SQL – Structured Query Language – мова структурованих запитів

СУБД – система управління базами даних

URI – Uniform Resource Identifier – уніфікований ідентифікатор ресурсу

WAL – Write-Ahead Logging – журналювання з випереджувальним записом

WSGI – Web Server Gateway Interface – інтерфейс шлюзу вебсервера

ВСТУП

Актуальність теми. Стрімке зростання кількості транспортних засобів у великих містах світу спричинило гострий дефіцит організованих паркувальних місць, що є однією з найбільш відчутних проблем сучасної урбаністики. За даними Головного сервісного центру МВС України, кількість зареєстрованих автомобілів у Києві за період 2015–2022 років зросла вдвічі – з 695 тисяч до 1,38 мільйона одиниць, тоді як кількість організованих муніципальних паркомісць складає лише близько 30 тисяч. Водії витрачають значний час на пошук вільного місця, що спричиняє зростання транспортних заторів, збільшення шкідливих викидів в атмосферу та загальне зниження ефективності міської мобільності.

Аналіз існуючих рішень засвідчує, що жодна з відомих систем не поєднує одночасно чотири ключові компоненти: моніторинг стану паркомісць у реальному часі, попереднє бронювання конкретного місця з розрахунком вартості, прогнозування завантаженості на основі аналізу історичних даних та рекомендаційну систему пошуку оптимального паркомісця. Заповнення цього функціонального розриву засобами сучасних веб-технологій та методів інтелектуального аналізу даних визначає актуальність даної роботи.

Мета і завдання дослідження. Метою кваліфікаційної роботи є розробка інтелектуальної системи моніторингу та бронювання паркувальних місць у міському середовищі, що забезпечує комплексне вирішення проблеми управління паркувальним простором на основі клієнт-серверної архітектури з використанням веб-технологій та статистичних методів аналізу даних.

Для досягнення поставленої мети визначено такі завдання:

1. Проаналізувати предметну область та існуючі системи моніторингу та бронювання паркувальних місць, виявити їхні переваги та недоліки;
2. Виконати порівняльний аналіз технологій та архітектурних підходів до побудови систем управління паркуванням та обґрунтувати вибір стеку технологій;

3. Визначити функціональні та нефункціональні вимоги до системи, спроєктувати архітектуру, базу даних та REST API;
4. Розробити алгоритм прогнозування завантаженості паркувальних зон на основі статистичного аналізу історичних даних;
5. Виконати програмну реалізацію серверної та клієнтської частин системи;
6. Провести тестування та верифікацію працездатності системи.

Об’єкт дослідження – процес моніторингу та управління паркувальними місцями в умовах міського середовища.

Предмет дослідження – методи та програмні засоби побудови інтелектуальної системи моніторингу, бронювання та прогнозування завантаженості паркувальних місць.

Методи дослідження: аналіз та синтез – для дослідження предметної області та формулювання вимог; порівняльний аналіз – для оцінки існуючих систем та технологій; статистичний аналіз – для розробки алгоритму прогнозування завантаженості; моделювання – для проєктування архітектури та бази даних; експериментальний метод – для тестування та верифікації розробленої системи.

Практичне значення одержаних результатів. Розроблена система може бути використана муніципальними органами управління транспортом, операторами паркувальних майданчиків та власниками торговельних центрів для оптимізації управління паркувальним простором. Модульна архітектура системи дозволяє поетапне впровадження – від програмного прототипу з імітацією сенсорних даних до повноцінної системи з підключенням реальних IoT-пристроїв.

Структура та обсяг роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку посилань та додатків. Перший розділ присвячено аналізу предметної області, огляду існуючих рішень та обґрунтуванню вибору технологій. У другому розділі виконано проєктування архітектури, бази даних, алгоритмів та REST API. Третій розділ містить опис програмної реалізації та результати тестування системи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ У СФЕРІ ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ ПАРКУВАННЯ

1.1 Аналіз проблеми паркування в умовах сучасного міського середовища

Стрімке зростання кількості транспортних засобів у великих містах світу стало одним із найбільш відчутних викликів для сучасної урбаністики. За останнє десятиліття щільність автомобілізації в містах-мільйонниках зросла настільки, що традиційна дорожньо-транспортна інфраструктура перестала справлятися з навантаженням. Проблема дефіциту організованих паркувальних місць безпосередньо впливає на якість життя мешканців, екологічну ситуацію та економічну ефективність міського транспорту. Водії витрачають значний час на пошук вільного паркомісця, що породжує низку негативних наслідків: зростання транспортних заторів, підвищення рівня шкідливих викидів в атмосферу, збільшення витрат пального та загальне зниження продуктивності міської мобільності.

Особливо гостро ця проблема проявляється у столиці України – місті Києві. Згідно з даними Головного сервісного центру Міністерства внутрішніх справ України, кількість зареєстрованих транспортних засобів у столиці за період з 2015 по 2022 рік зросла вдвічі: з 695 тисяч до 1,38 мільйона одиниць [1]. Максимальна добова кількість активних автомобілів на дорогах міста досягла позначки 1,13 мільйона. При цьому, за оцінками Київської міської державної адміністрації, на початку 2024 року в місті налічувалося лише близько 30 тисяч організованих муніципальних паркомісць [2], що свідчить про кратний розрив між попитом та пропозицією паркувальної інфраструктури.

Дослідження, проведені у межах Програми розвитку єдиного паркувального простору Києва, засвідчили, що забезпеченість столиці паркомісцями є

щонайменше вдвічі нижчою за необхідну [3]. Найскладніша ситуація спостерігається в центральній частині міста, де сконцентровано ключові адміністративні установи, бізнес-центри та об’єкти торгівлі. Водії вимушені залишати автомобілі вздовж тротуарів, створюючи перешкоди для пішохідного руху, або займати смуги проїзної частини, що додатково звужує пропускну здатність вулиць. За підсумками 2024 року інспекція з паркування міста Києва наклала штрафи на понад 230 тисяч водіїв за порушення правил паркування, а надходження до міського бюджету від цих штрафів склали 74,1 мільйона гривень [4].

Варто зазначити, що проблема паркування не є суто вітчизняною – вона має глобальний характер. За різними оцінками, у великих містах Європи та Північної Америки від 25 до 40 відсотків міського трафіку в години пік припадає на автомобілі, водії яких шукають вільне паркомісце. Це явище отримало назву «круїзинг» (від англійського cruising for parking) і вважається одним із найбільш нераціональних проявів міської мобільності [5]. Дослідження, опубліковане у журналі PeerJ Computer Science, підтверджує, що марний пошук паркомісць спричиняє значні витрати часу та пального, а також суттєво збільшує обсяг шкідливих викидів у міському середовищі [6].

У контексті розвитку концепції «розумного міста» (Smart City) питання ефективного управління паркувальним простором набуває особливого значення. Сучасні міста активно впроваджують цифрові платформи, що інтегрують дані з різноманітних джерел для оптимізації міських процесів. Паркування є одним із тих напрямків, де інформаційні технології здатні забезпечити найбільший практичний ефект за відносно невеликих інвестицій. Ключова ідея полягає в тому, щоб замінити хаотичний пошук вільного місця інформаційно підтриманим процесом, у якому водій заздалегідь знає, де розташоване вільне паркомісце, і може забронювати його до моменту прибуття [7].

Технологічною основою сучасних інтелектуальних систем паркування є Інтернет речей (Internet of Things, IoT). Сенсорні пристрої, встановлені на кожному паркомісці, у режимі реального часу визначають його статус (вільне або зайняте)

та передають дані на центральний сервер через бездротові мережі зв’язку. Серед поширених типів сенсорів – ультразвукові, інфрачервоні, магнітні та на основі комп’ютерного зору. Кожен тип має свої переваги та обмеження щодо точності, вартості, енергоспоживання та стійкості до погодних умов [8]. Останніми роками набули популярності рішення на базі протоколу LoRaWAN (Long Range Wide Area Network), який дозволяє передавати дані на значні відстані за мінімального енергоспоживання, що робить його придатним для масштабних міських впроваджень [9].

Окрім збору даних у реальному часі, важливою складовою інтелектуальних систем паркування є аналітичний модуль, який на основі накопичених історичних даних здатен прогнозувати завантаженість паркувальних зон у певний день тижня та годину доби. Застосування методів машинного навчання дозволяє виявляти закономірності у паттернах використання паркомісць та надавати користувачам рекомендації щодо оптимального часу та місця для паркування [10]. У дослідженні Dahiya та ін. (2024), опублікованому у Journal of Advanced Transportation, здійснено порівняльний аналіз моделей машинного навчання для прогнозування доступності паркомісць у середовищі IoT, що підтвердило практичну цінність предиктивних алгоритмів для зниження часу пошуку місця та підвищення ефективності використання паркувальної інфраструктури [11].

Ситуація з паркуванням у Києві ускладнюється також нормативно-правовою нестабільністю. Протягом останніх років тарифи на платне паркування неодноразово скасовувалися через судові рішення. Зокрема, у лютому 2024 року Шостий апеляційний адміністративний суд визнав незаконним розпорядження КМДА про затвердження тарифів, внаслідок чого паркування тимчасово стало безоплатним [2]. Загалом за період з 2021 по 2024 рік плату за паркування скасовували чотири рази сумарною тривалістю близько 12 місяців. Така невизначеність негативно впливає на дисципліну водіїв та ефективність використання паркувального простору, оскільки за відсутності економічних стимулів водії схильні займати паркомісця на тривалий час, зменшуючи їхню оборотність.

Щомісячні надходження до бюджету столиці від оплати паркування продемонстрували значну динаміку зростання – з 1 мільйона гривень у 2021 році до 10 мільйонів гривень на місяць у 2024 році [12]. Це свідчить про поступове формування культури оплати паркування серед водіїв, однак проблема залишається далекою від вирішення. Кількість щоденних постанов інспекторів за порушення правил паркування зросла у п’ять–сім разів порівняно з 2021 роком, що вказує як на посилення контролю, так і на масовість порушень.

У 2024 році Київська міська державна адміністрація затвердила оновлені межі паркувальних зон міста на основі аналітичних даних платформи TomTom Move O/D – одного з провідних світових джерел статистики автомобільних поїздок [13]. Перша зона, яка охоплює центр міста з найвищим попитом на паркування, розширилася з 66 паркувальних майданчиків на 1451 машиномісце до 224 паркувальних майданчиків на 4320 автомобілів. Такий підхід, заснований на математично обґрунтованих даних, є прикладом переходу від суб’єктивного адміністрування до управління на основі аналітики, що цілком узгоджується з принципами інтелектуальних систем моніторингу.

Аналіз поточного стану справ дозволяє виділити кілька ключових проблем, що потребують розв’язання засобами інформаційних технологій. По-перше, відсутність єдиної цифрової платформи, яка б у реальному часі надавала водіям інформацію про наявність вільних паркомісць у межах міста. По-друге, неможливість попереднього бронювання конкретного місця, що змушує водіїв покладатися на випадковість при пошуку паркування. По-третє, відсутність аналітичних інструментів прогнозування завантаженості, які б дозволяли як водіям планувати свої поїздки, так і адміністраторам – приймати обґрунтовані рішення щодо розвитку паркувальної інфраструктури. По-четверте, необхідність автоматизації процесів моніторингу та контролю, що знизить навантаження на інспекційні органи та підвищить ефективність збору плати за паркування.

Розробка інтелектуальної системи моніторингу та бронювання паркувальних місць є актуальним завданням, що відповідає сучасним тенденціям цифрової трансформації міського середовища. Така система має поєднувати функції збору

даних у реальному часі, інтерактивної візуалізації паркувальних зон на мапі, онлайн-бронювання місць із розрахунком вартості, а також інтелектуального прогнозування та рекомендацій на основі аналізу накопичених даних. Впровадження подібних рішень здатне суттєво покращити ситуацію з паркуванням у великих містах, зменшити час пошуку вільного місця, знизити рівень транспортних заторів та шкідливих викидів, а також підвищити ефективність використання обмежених паркувальних ресурсів.

1.2 Огляд існуючих систем моніторингу та бронювання паркувальних місць

Сучасний ринок інтелектуальних систем паркування представлений значною кількістю рішень, які різняться за масштабом впровадження, технологічним підходом та функціональними можливостями. Для обґрунтування архітектурних та технологічних рішень, що закладаються в основу розроблюваної системи, доцільно провести детальний аналіз найбільш відомих та успішно реалізованих проєктів у цій галузі. Такий аналіз дозволить визначити їхні сильні та слабкі сторони, а також окреслити перспективні напрямки вдосконалення.

SFpark (Сан-Франциско, США). Одним із найбільш масштабних та науково обґрунтованих проєктів у галузі інтелектуального паркування є система SFpark, розроблена Муніципальною транспортною агенцією Сан-Франциско (SFMTA) у 2011 році за підтримки Федерального управління шляхів США. Проєкт передбачав встановлення бездротових електромагнітних сенсорів у асфальтне покриття під 8200 паркувальними місцями для визначення їхнього статусу в режимі реального часу [14]. Ключовою інновацією SFpark стало впровадження динамічного ціноутворення, що базується на попиті: вартість паркування автоматично змінюється залежно від рівня завантаженості конкретного кварталу з метою підтримки оптимального рівня зайнятості на рівні 60–80 відсотків [15]. За результатами пілотного дослідження, середній час пошуку паркомісця скоротився на 30 відсотків, а кількість штрафних квитанцій зменшилася із 45 до 20 відсотків загальних надходжень від паркування [16]. Середня погодинна вартість паркування

при цьому знизилася на 4 відсотки. У 2018 році систему було розширено на всі 28 000 паркоматів міста.

ParkMobile. Платформа ParkMobile є однією з найбільш поширених комерційних систем управління паркуванням у Сполучених Штатах та Європі. Сервіс інтегровано з тисячами паркоматів у понад 1000 містах та обслуговує понад 22 мільйони зареєстрованих користувачів [17]. Основна функціональність ParkMobile зосереджена на цифровій оплаті вуличного паркування через мобільний додаток, що дозволяє водіям сплачувати за місце та подовжувати час паркування дистанційно, без необхідності повертатися до паркомату. Система підтримує сповіщення про закінчення оплаченого часу, управління кількома транспортними засобами та інтеграцію з навігаційними сервісами. Проте ParkMobile не здійснює моніторинг фізичного стану паркомісць у реальному часі та не має власної мережі IoT-сенсорів, що обмежує можливості надання інформації про фактичну доступність місць.

SpotHero. Платформа SpotHero реалізує модель попереднього бронювання паркувальних місць у гаражах та на приватних паркувальних майданчиках. Сервіс працює у понад 300 містах Північної Америки та співпрацює з більш ніж 4500 паркувальними об’єктами [18]. Основна перевага SpotHero полягає у можливості порівняння цін та бронювання конкретного місця заздалегідь, що гарантує водію наявність паркомісця на момент прибуття. Платформа забезпечує знижки до 50 відсотків порівняно з цінами при безпосередньому заїзді, що досягається завдяки викупу нереалізованих місць у паркувальних операторів за зниженою вартістю. Водночас SpotHero орієнтований переважно на закриті парковки (гаражі, лоти) і не охоплює вуличне паркування, а також не використовує сенсорну інфраструктуру для моніторингу доступності місць у реальному часі.

Parkopedia. Міжнародна платформа Parkopedia позиціонується як найбільша у світі база даних паркувальних місць, що охоплює понад 70 мільйонів місць у 89 країнах [19]. Система агрегує інформацію з різноманітних джерел – муніципальних баз даних, приватних операторів, картографічних сервісів – та надає користувачам відомості про розташування, ціни, режим роботи та доступність парковок.

Parkopedia інтегрована з навігаційними системами багатьох автовиробників, зокрема BMW, Audi, Mercedes-Benz та Toyota, що дозволяє водіям отримувати інформацію про паркування безпосередньо з бортового комп'ютера автомобіля. Платформа також підтримує функцію оплати паркування через додаток у низці міст. Обмеженням Parkopedia є те, що дані про доступність місць часто мають статистичний, а не сенсорний характер, тобто базуються на прогнозних моделях, а не на показниках фізичних датчиків.

LA Express Park (Лос-Анджелес, США). Система LA Express Park, впроваджена Міністерством транспорту Лос-Анджелеса у 2012 році, є ще одним прикладом реалізації концепції динамічного ціноутворення на основі попиту. Система охоплює район Downtown Los Angeles і використовує мережу сенсорів для моніторингу завантаженості паркувальних місць у реальному часі. На підставі отриманих даних ціни на паркування автоматично коригуються з періодичністю раз на місяць. Водії мають змогу переглядати доступність та ціни через мобільний додаток та вебсайт проєкту [20]. Досвід LA Express Park підтвердив, що навіть помірні зміни цін можуть суттєво впливати на розподіл попиту між паркувальними зонами та знижувати рівень транспортних заторів у центральних районах.

Київ Цифровий (Україна). В Україні найбільш масштабною цифровою платформою, що частково реалізує функції управління паркуванням, є мобільний додаток «Київ Цифровий». Через цей сервіс водії мають можливість здійснювати оплату за паркування на муніципальних майданчиках, переглядати розташування паркувальних зон на інтерактивній карті та отримувати сповіщення. Станом на 2024 рік у додатку зареєстровано понад 100 тисяч водіїв, які регулярно сплачують за паркування [21]. Додаток також підтримує функцію безоплатного паркування для захисників та ветеранів. Однак «Київ Цифровий» не є повноцінною інтелектуальною системою паркування: він не здійснює моніторинг фактичної зайнятості місць за допомогою сенсорів, не надає функцію бронювання конкретного паркомісця та не містить модулів прогнозування завантаженості на основі аналітики даних.

Barcelona Smart Parking (Іспанія). Барселона є одним із визнаних лідерів серед європейських міст у впровадженні технологій розумного паркування. Місто розгорнуло мережу IoT-сенсорів, вбудованих безпосередньо у дорожнє покриття, які передають дані про зайнятість кожного паркомісця на центральну платформу управління. Водії отримують інформацію про вільні місця через мобільний додаток із навігацією до обраного паркомісця. Система інтегрована з міською платформою Smart City Barcelona, що дозволяє координувати управління паркуванням із іншими міськими сервісами – транспортом, освітленням, моніторингом довкілля [22]. Впровадження системи сприяло зменшенню середнього часу пошуку паркомісця та зниженню інтенсивності руху в центральних районах.

Для систематизації результатів проведеного аналізу доцільно подати порівняльну характеристику розглянутих систем у табличній формі (табл. 1.1).

Таблиця 1.1

Порівняльна характеристика існуючих систем паркування

Країна походження	Система	IoT-сенсори	Бронювання	Прогнозування	Динамічне ціноутворення
США	SFpark	Так	Ні	Ні	Так
США	ParkMobile	Ні	Частково	Ні	Ні
США	SpotHero	Ні	Так	Ні	Ні
Велика Британія	Parkopedia	Частково	Частково	Так	Ні
США	LA Express Park	Так	Ні	Ні	Так
Україна	Київ Цифровий	Ні	Ні	Ні	Ні
Іспанія	Barcelona Smart Parking	Так	Ні	Частково	Ні

Проведений аналіз засвідчує, що жодна з розглянутих систем не реалізує повний комплекс функцій, необхідних для ефективного управління паркувальним простором у міському середовищі. Системи, що базуються на IoT-сенсорах (SFpark, Barcelona Smart Parking, LA Express Park), забезпечують якісний моніторинг у реальному часі, проте не надають можливості попереднього бронювання конкретного місця. Комерційні платформи бронювання (SpotHero, ParkWhiz) вирішують задачу резервування, але не мають інфраструктури для моніторингу фактичного стану місць. Агрегатори даних (Parkopedia) охоплюють значну географію, однак оперують переважно статистичними, а не сенсорними даними. Українська платформа «Київ Цифровий» виконує функцію оплати, але не містить ані сенсорного моніторингу, ані бронювання, ані предиктивної аналітики.

Спільним недоліком більшості розглянутих рішень є відсутність інтегрованого модуля інтелектуального прогнозування завантаженості, який би на основі аналізу історичних даних надавав користувачам рекомендації щодо оптимального часу та місця паркування. Лише окремі платформи (Parkopedia) здійснюють статистичне прогнозування, і жодна з проаналізованих систем не поєднує одночасно чотири ключові компоненти: моніторинг у реальному часі, попереднє бронювання, прогнозування завантаженості та рекомендаційну систему. Саме заповнення цього функціонального розриву є метою розроблюваної в рамках даної кваліфікаційної роботи інтелектуальної системи моніторингу та бронювання паркувальних місць.

Окрім функціональних обмежень, варто відзначити економічний аспект: впровадження масштабних сенсорних мереж, подібних до SFpark чи Barcelona Smart Parking, потребує значних капітальних інвестицій. Вартість проєкту SFpark склала близько 25 мільйонів доларів США, з яких 80 відсотків було профінансовано з федерального бюджету [14]. Для міст із обмеженими бюджетними ресурсами, зокрема українських, більш доцільним є поетапний підхід, що передбачає створення програмної платформи з можливістю поступового підключення апаратних сенсорних модулів. Такий підхід дозволяє розпочати з програмної

імітації даних сенсорів та базової аналітики, а в подальшому інтегрувати реальні IoT-пристрої без суттєвої модифікації архітектури системи.

Аналіз існуючих систем моніторингу та бронювання паркувальних місць дозволяє сформулювати ключові вимоги до розроблюваного рішення: поєднання функцій реального моніторингу статусу місць, попереднього бронювання з розрахунком вартості, інтелектуального прогнозування завантаженості на основі історичних даних та рекомендаційної системи для визначення оптимального паркомісця. Система має бути побудована на відкритих технологіях, мати модульну клієнт-серверну архітектуру та підтримувати можливість масштабування як у програмному, так і в апаратному напрямках.

1.3 Порівняльний аналіз технологій та архітектурних підходів до побудови систем управління паркуванням

Проєктування інтелектуальної системи паркування вимагає обґрунтованого вибору технологій на кожному рівні архітектури – від сенсорного забезпечення до серверної частини та інтерфейсу користувача. Сучасні системи управління паркуванням будуються, як правило, на основі багаторівневої архітектури, що включає рівень збору даних (сенсори та IoT-пристрої), рівень передачі даних (комунікаційні протоколи), рівень обробки та зберігання (серверна інфраструктура, бази даних) та рівень представлення (клієнтські додатки). Кожен з цих рівнів має альтернативні технологічні рішення, вибір між якими визначається вимогами до продуктивності, масштабованості, вартості впровадження та простоти підтримки.

Сенсорні технології. На рівні збору даних про зайнятість паркомісць застосовуються кілька основних типів сенсорів. Ультразвукові датчики визначають наявність транспортного засобу за принципом відбиття звукових хвиль від поверхні об’єкта. Вони характеризуються високою точністю (до 99 відсотків), проте чутливі до погодних умов – дощу, снігу, забруднення поверхні датчика [23]. Магнітні (магнітометричні) сенсори реєструють зміну магнітного поля Землі,

спричинену присутністю металевого корпусу автомобіля. Цей тип сенсорів стійкий до погодних впливів та має тривалий термін автономної роботи, що зумовило його використання у проєктах SFpark та Barcelona Smart Parking [24]. Інфрачервоні датчики працюють на основі детектування теплового випромінювання транспортного засобу, однак їхня точність знижується за високих температур зовнішнього середовища. Системи на основі комп'ютерного зору використовують камери та алгоритми розпізнавання образів для визначення статусу паркомісць, а також дозволяють додатково ідентифікувати номерні знаки транспортних засобів [25].

Для систематизації порівняльних характеристик сенсорних технологій доцільно подати їх у табличній формі (табл. 1.2).

Таблиця 1.2

Порівняльна характеристика сенсорних технологій для систем паркування

Тип сенсора	Точність	Вартість	Стійкість до погоди	Автономність	Додаткові функції
Ультразвуковий	Висока	Середня	Низька	Середня	Ні
Магнітний	Висока	Низька	Висока	Висока	Ні
Інфрачервоний	Середня	Низька	Середня	Висока	Ні
Комп'ютерний зір	Висока	Висока	Середня	Низька	ANPR

Комунікаційні протоколи. Передача даних від сенсорів до серверної інфраструктури здійснюється через бездротові мережі. Серед основних протоколів, що застосовуються в системах паркування, варто виділити Wi-Fi, Bluetooth Low Energy (BLE), ZigBee, LoRaWAN та NB-IoT. Протокол Wi-Fi забезпечує високу швидкість передачі даних, проте має обмежений радіус дії та значне енергоспоживання, що робить його малопридатним для автономних сенсорних

вузлів. Bluetooth Low Energy ефективний для малих відстаней (до 100 метрів) і характеризується низьким енергоспоживанням. ZigBee підтримує mesh-топологію мережі, що забезпечує самоорганізацію та відмовостійкість. Проте найбільш перспективними для масштабних міських впроваджень вважаються технології LoRaWAN та NB-IoT, які поєднують значний радіус дії (до 15 кілометрів для LoRaWAN) з мінімальним енергоспоживанням, що дозволяє сенсорним вузлам працювати від батареї протягом кількох років [26]. Дослідження Jabbar та ін. (2024) підтвердило ефективність IoT-системи паркування на базі LoRaWAN за критеріями надійності передачі даних, точності детектування та економічності розгортання [27].

Серверні технології та бази даних. На рівні обробки та зберігання даних у сучасних системах паркування застосовується широкий спектр серверних технологій. Мікрофреймворки на базі Python (Flask, FastAPI, Django) та Node.js (Express) використовуються для побудови REST API, що забезпечують взаємодію між клієнтською та серверною частинами системи. Python-фреймворки є особливо популярними у дослідницьких та прототипних проєктах завдяки простоті розробки, великій екосистемі бібліотек для аналізу даних та машинного навчання (NumPy, Pandas, scikit-learn) та невисокому порогу входу [28]. Для зберігання даних застосовуються як реляційні СУБД (PostgreSQL, MySQL, SQLite), так і нереляційні рішення (MongoDB, InfluxDB). SQLite є оптимальним вибором для систем невеликого та середнього масштабу, оскільки не потребує окремого серверного процесу, зберігає всю базу даних у єдиному файлі та підтримує повноцінний діалект SQL із транзакційною безпекою [29]. Для високонавантажених систем із десятками тисяч одночасних запитів доцільним є використання PostgreSQL або хмарних сервісів баз даних.

Клієнтські технології та візуалізація. На рівні представлення даних користувачу основним трендом є розробка вебзастосунків типу Single Page Application (SPA) із використанням JavaScript-фреймворків React, Vue.js або Angular. React, розроблений компанією Meta, є найбільш поширеним рішенням для побудови інтерактивних інтерфейсів завдяки компонентному підходу,

віртуальному DOM та широкій екосистемі сторонніх бібліотек [30]. Для відображення геопросторових даних – розташування паркувальних зон на карті міста – застосовуються картографічні бібліотеки. Leaflet.js є легковісною відкритою бібліотекою для інтерактивних карт, що базується на тайлах OpenStreetMap і не потребує платних API-ключів, на відміну від Google Maps API [31]. Альтернативою є Mapbox GL JS, який забезпечує більш гнучку стилізацію карт, проте має обмеження безкоштовного тарифу за кількістю запитів.

Архітектурні підходи. З точки зору загальної архітектури, системи паркування можуть будуватися за моделями хмарних обчислень (Cloud Computing), туманних обчислень (Fog Computing) або гібридною моделлю. Хмарна архітектура передбачає передачу всіх даних на віддалений сервер для централізованої обробки, що забезпечує необмежену масштабованість, але створює залежність від якості інтернет-з'єднання та може спричиняти затримки при обробці критичних даних у реальному часі. Туманна архітектура розміщує частину обчислювальних ресурсів ближче до джерел даних – на периферійних шлюзах, що знижує навантаження на мережу та забезпечує мінімальну затримку обробки [32]. Дослідження, опубліковане в журналі Scientific Reports (2025), продемонструвало переваги туманної архітектури для систем паркування у реальному часі, зокрема скорочення затримки відповіді та оптимізацію використання мережевих ресурсів [33]. Для систем середнього масштабу, що розгортаються в рамках одного міста, оптимальним є класичний клієнт-серверний підхід із REST API, який забезпечує достатню продуктивність при значно простішому розгортанні та супроводі.

Методи прогнозування та машинного навчання. Інтелектуальна складова систем паркування реалізується через алгоритми прогнозування завантаженості, що базуються на аналізі історичних даних. Серед підходів, що застосовуються у дослідженнях, виділяються статистичні методи (ковзне середнє, експоненційне згладжування, ARIMA), класичні алгоритми машинного навчання (лінійна регресія, Random Forest, Gradient Boosting) та глибокі нейронні мережі (LSTM, CNN) [34]. Статистичні методи є найпростішими у реалізації та інтерпретації результатів, не потребують значних обчислювальних ресурсів і добре працюють

при наявності стабільних часових патернів – наприклад, типових циклів «робочий день – вихідний» або «ранковий пік – обід – вечірній пік». Для систем, що знаходяться на етапі початкового впровадження та ще не накопичили достатнього обсягу даних для навчання складних моделей, статистичний підхід на основі середніх значень завантаженості за день тижня та годину доби є обґрунтованим та практично ефективним рішенням [35].

1.4 Обґрунтування вибору методів та інструментів розробки

На підставі проведеного аналізу предметної області, існуючих рішень та технологічних підходів, для реалізації інтелектуальної системи моніторингу та бронювання паркувальних місць обрано наступний стек технологій та архітектурних рішень.

Серверна частина: Python 3.12, Flask 3.0. Мова програмування Python обрана з огляду на її домінуючі позиції у сфері аналізу даних, машинного навчання та швидкого прототипування вебзастосунків. Flask є мікрофреймворком, що реалізує патерн WSGI та забезпечує створення REST API з мінімальним обсягом конфігурації. На відміну від повнофункціональних фреймворків (Django), Flask надає розробнику повний контроль над архітектурою додатку, що є важливим для дослідницьких проєктів із нестандартною бізнес-логікою [36]. Flask підтримує модульну організацію коду через механізм Blueprints, інтеграцію з будь-якими ORM та СУБД, а також має розвинену екосистему розширень (Flask-CORS, Flask-JWT-Extended, Flask-SQLAlchemy). Вибір Flask обґрунтовується також тим, що серверна частина системи реалізує порівняно невелику кількість ендпоінтів (близько 15), і використання більш масивних фреймворків було б надлишковим.

База даних: SQLite 3. Для зберігання даних обрано реляційну СУБД SQLite, яка є вбудованою бібліотекою і не потребує окремого серверного процесу. SQLite зберігає всю базу даних у єдиному файлі, підтримує стандартний синтаксис SQL, забезпечує атомарність транзакцій через механізм Write-Ahead Logging (WAL) та підтримує зовнішні ключі з каскадними операціями [29]. Для системи паркування

міського масштабу, що обслуговує до кількох сотень одночасних запитів, продуктивності SQLite є цілком достатньо. Додатковою перевагою є портативність: для розгортання системи на новому сервері достатньо скопіювати єдиний файл бази даних без необхідності встановлення та конфігурації СУБД. У разі зростання навантаження архітектура системи дозволяє здійснити міграцію на PostgreSQL із мінімальними змінами коду завдяки стандартизованому SQL-інтерфейсу.

Клієнтська частина: React 18, Vite. Для побудови інтерфейсу користувача обрано бібліотеку React, яка забезпечує створення інтерактивних вебзастосунків типу Single Page Application. React реалізує компонентний підхід до побудови інтерфейсу, використовує віртуальний DOM для оптимізації оновлень та підтримує функціональні компоненти з хуками стану (useState, useEffect, useCallback), що спрощує управління станом додатку без необхідності застосування зовнішніх бібліотек (Redux, MobX) [30]. Як інструмент збірки обрано Vite, який забезпечує миттєвий запуск сервера розробки та прозорий проксі запитів до бекенду, що спрощує локальну розробку. Стилізація компонентів здійснюється через inline CSS-об’єкти, що забезпечує повну ізоляцію стилів без необхідності додаткових CSS-процесорів.

Картографічний сервіс: Leaflet.js, OpenStreetMap. Для інтерактивної візуалізації паркувальних зон на карті міста обрано бібліотеку Leaflet.js у поєднанні з тайловим сервісом OpenStreetMap. Leaflet є найбільш поширеною відкритою картографічною бібліотекою для вебзастосунків, що характеризується невеликим розміром (близько 42 кілобайти стисненого коду), високою продуктивністю та підтримкою кастомних маркерів, спливаючих підказок та інтерактивних шарів [31]. Принциповою перевагою є відсутність необхідності в платних API-ключах: OpenStreetMap надає тайли безкоштовно, тоді як Google Maps Platform потребує активації білінгу з вартістю від 7 доларів США за 1000 запитів до Maps JavaScript API.

Авторизація: токен-базована автентифікація (Bearer Token). Для забезпечення контролю доступу до системи реалізовано механізм автентифікації на основі Bearer-токенів. При успішному вході користувача сервер генерує

криптографічно безпечний токен (256 біт), який зберігається на клієнтській стороні та передається з кожним запитом у заголовку Authorization. Паролі зберігаються у вигляді SHA-256 хешу із додаванням унікальної випадкової «солі» для кожного користувача, що унеможливлює використання веселкових таблиць (rainbow tables) для відновлення паролів. Система підтримує рольову модель доступу з двома ролями: «користувач» та «адміністратор», що дозволяє розмежувати доступ до адміністративних функцій (управління зонами, перегляд статистики, управління користувачами).

Алгоритм прогнозування: статистичний метод на основі історичних даних. Для реалізації інтелектуального модуля прогнозування завантаженості обрано статистичний підхід, що базується на обчисленні середнього, мінімального та максимального значень завантаженості за кожну годину кожного дня тижня на основі накопичених за 30 діб спостережень. Такий метод є обґрунтованим для систем на етапі початкового впровадження, коли обсяг даних ще недостатній для навчання складних моделей машинного навчання. Перевагою обраного підходу є простота реалізації, інтерпретованість результатів та мінімальні вимоги до обчислювальних ресурсів. У подальшому архітектура системи дозволяє замінити статистичний модуль на модель машинного навчання (Random Forest, LSTM) без зміни API-інтерфейсу.

Рекомендаційна система: алгоритм пошуку найближчого об’єкта. Для реалізації рекомендаційного модуля, який підказує водію оптимальне паркомісце, використовується алгоритм пошуку найближчих паркувальних зон із вільними місцями на основі евклідової відстані між поточними координатами користувача та координатами паркувальних зон. Алгоритм виконує фільтрацію за наявністю вільних місць, обчислює відстань, ранжує результати та повертає п’ять найближчих варіантів із зазначенням кількості вільних місць, вартості та відстані. У перспективі алгоритм може бути розширений урахуванням прогнозованої завантаженості, часу в дорозі та індивідуальних вподобань користувача.

Таким чином, обраний стек технологій забезпечує баланс між функціональністю, продуктивністю та простотою розгортання. Всі компоненти є

відкритими (open-source) або безкоштовними, що мінімізує вартість впровадження та усуває залежність від комерційних ліцензій. Модульна архітектура системи дозволяє замінювати окремі компоненти (СУБД, алгоритм прогнозування, картографічний сервіс) без суттєвої переробки інших частин, що забезпечує гнучкість подальшого розвитку.

Висновки до розділу 1

У першому розділі кваліфікаційної роботи проведено комплексний аналіз предметної області інтелектуальних систем моніторингу та бронювання паркувальних місць у міському середовищі. За результатами дослідження сформульовано такі висновки:

1. Встановлено, що проблема дефіциту організованих паркувальних місць є однією з найгостріших для великих міст України та світу. Кількість зареєстрованих транспортних засобів у Києві зросла вдвічі за період 2015–2022 років і досягла 1,38 мільйона одиниць, тоді як кількість муніципальних паркомісць складає лише близько 30 тисяч. Водії витрачають значний час на пошук вільного місця, що спричиняє зростання заторів, шкідливих викидів та загальне зниження ефективності міської мобільності.
2. Проведено порівняльний аналіз семи існуючих систем паркування: SFpark, ParkMobile, SpotHero, Parkopedia, LA Express Park, Barcelona Smart Parking та «Київ Цифровий». Встановлено, що жодна з розглянутих систем не поєднує одночасно чотири ключові компоненти: моніторинг у реальному часі на базі IoT-сенсорів, попереднє бронювання конкретного місця, прогнозування завантаженості на основі аналітики даних та рекомендаційну систему. Цей функціональний розрив визначає актуальність та практичну цінність розроблюваної системи.
3. Виконано порівняльний аналіз технологій на кожному рівні архітектури системи паркування: сенсорних технологій (ультразвукових, магнітних, інфрачервоних, комп’ютерного зору), комунікаційних протоколів (Wi-Fi,

BLE, ZigBee, LoRaWAN, NB-IoT), серверних фреймворків (Flask, Django, Express), СУБД (SQLite, PostgreSQL, MongoDB), клієнтських фреймворків (React, Vue.js, Angular) та картографічних бібліотек (Leaflet.js, Google Maps, Mapbox).

4. Обґрунтовано вибір стеку технологій для реалізації системи: Python Flask як серверний фреймворк, SQLite як СУБД, React як клієнтська бібліотека, Leaflet.js з OpenStreetMap як картографічний сервіс, токен-базована автентифікація з хешуванням паролів, статистичний метод прогнозування завантаженості та евклідовий алгоритм рекомендацій. Обрані технології є відкритими, безкоштовними та забезпечують модульну архітектуру з можливістю подальшого масштабування.

2 ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ МОНІТОРИНГУ ТА БРОНЮВАННЯ ПАРКУВАЛЬНИХ МІСЦЬ

2.1 Визначення функціональних та нефункціональних вимог до системи

Проєктування інтелектуальної системи моніторингу та бронювання паркувальних місць розпочинається з формалізації вимог, які визначають межі функціональності, очікувану поведінку та якісні характеристики програмного продукту. Вимоги до системи поділяються на функціональні, що описують конкретні можливості та сценарії використання, та нефункціональні, що визначають обмеження щодо продуктивності, безпеки, надійності та інших якісних параметрів. Формулювання вимог базується на результатах аналізу предметної області, проведеного в першому розділі, та враховує виявлені недоліки існуючих систем паркування.

Розроблювана система призначена для двох категорій користувачів із різними потребами та рівнями доступу. Першою категорією є звичайні користувачі (водії), які використовують систему для пошуку, перегляду та бронювання паркувальних місць, а також для отримання рекомендацій та прогнозів завантаженості. Другою категорією є адміністратори системи, які здійснюють управління паркувальними зонами, моніторинг загального стану системи, перегляд аналітики та управління обліковими записами користувачів. Така рольова модель є типовою для інформаційних систем у сфері міських сервісів та забезпечує розмежування відповідальності між кінцевими споживачами послуги та персоналом, що її адмініструє.

Функціональні вимоги. На підставі проведеного аналізу предметної області та з урахуванням необхідності заповнення функціонального розриву, виявленого при огляді існуючих рішень, сформульовано 15 функціональних вимог до системи. Вимоги згруповано за підсистемами: автентифікація та управління доступом (FR-

01 – FR-03), моніторинг та візуалізація (FR-04 – FR-05), бронювання (FR-06 – FR-09), інтелектуальні функції (FR-10 – FR-11), аналітика та адміністрування (FR-12 – FR-15). Кожній вимозі присвоєно ідентифікатор та визначено пріоритет реалізації (табл. 2.1).

Таблиця 2.1

Функціональні вимоги до системи

ID	Назва	Опис	Пріоритет
FR-01	Реєстрація користувача	Система має дозволяти реєстрацію нових користувачів із зазначенням логіну, email, пароля, повного імені та телефону	Висока
FR-02	Автентифікація та авторизація	Система має забезпечувати вхід користувача за логіном або email та паролем із видачею токена доступу	Висока
FR-03	Рольова модель доступу	Система має підтримувати ролі «користувач» та «адміністратор» із розмежуванням прав доступу до функцій	Висока
FR-04	Перегляд паркувальних зон на карті	Система має відображати всі активні паркувальні зони на інтерактивній карті міста з маркерами, що вказують кількість вільних місць	Висока
FR-05	Перегляд детальної інформації про зону	Система має надавати інформацію про назву, адресу, тип, тариф, кількість вільних/зайнятих/заброньованих місць та візуалізацію кожного місця	Висока
FR-06	Бронювання паркомісця	Система має дозволяти користувачу обрати вільне місце, вказати час початку, тривалість та номерний знак, з автоматичним розрахунком вартості	Висока
FR-07	Перевірка перетину бронювань	Система має автоматично перевіряти наявність часових перетинів з існуючими бронюваннями для обраного місця	Висока
FR-08	Перегляд бронювань	Система має надавати користувачу перелік його бронювань з можливістю фільтрації за статусом (активні, завершені, скасовані)	Середня

Продовження таблиці 2.1

FR-09	Скасування бронювання	Система має дозволяти скасувати активне бронювання з автоматичним звільненням зарезервованого місця	Висока
FR-10	Прогнозування завантаженості	Система має надавати прогноз завантаженості обраної зони за кожну годину обраного дня тижня на основі історичних даних	Середня
FR-11	Рекомендація паркувальних зон	Система має рекомендувати найближчі паркувальні зони з вільними місцями на основі координат користувача	Середня
FR-12	Панель моніторингу (дашборд)	Система має надавати дашборд з KPI: кількість місць, завантаженість, бронювання, дохід, графіки по зонах та годинах	Середня
FR-13	Управління паркувальними зонами	Адміністратор має мати змогу створювати, редагувати та видаляти паркувальні зони з автоматичним генеруванням місць	Висока
FR-14	Перегляд списку користувачів	Адміністратор має мати змогу переглядати перелік зареєстрованих користувачів з їхніми даними та ролями	Низька
FR-15	Імітація оновлення сенсорів	Система має підтримувати ендпоінт для імітації зміни стану паркомісць з журналюванням завантаженості	Низька

Розглянемо детальніше ключові функціональні вимоги та їхню роль у забезпеченні цільових сценаріїв використання системи.

Вимоги FR-01 – FR-03 визначають підсистему автентифікації та авторизації. Реєстрація нового користувача (FR-01) передбачає введення п’яти полів: унікальний логін, email-адреса, пароль, повне ім’я та номер телефону. Система має перевіряти унікальність логіну та email, а також зберігати пароль виключно у хешованому вигляді із використанням криптографічної солі. Автентифікація (FR-02) здійснюється через REST-ендпоінт, що приймає логін або email разом із

паролем, виконує верифікацію та у разі успіху повертає токен доступу. Рольова модель (FR-03) передбачає існування двох ролей – «user» та «admin» – із перевіркою ролі на рівні кожного захищеного ендпоінта серверної частини.

Вимоги FR-04 – FR-05 визначають підсистему моніторингу та візуалізації. Інтерактивна карта (FR-04) має відображати всі активні паркувальні зони у вигляді маркерів, колір яких динамічно змінюється залежно від рівня завантаженості: зелений при завантаженості до 60 відсотків, жовтий – від 60 до 85, червоний – понад 85 відсотків. При натисканні на маркер або обранні зони зі списку, система має відображати детальну інформацію (FR-05): назву, адресу, тип зони (вулична, підземна, багаторівнева, відкрита), тариф за годину, а також візуальну сітку всіх паркомісць із кольоровою індикацією їхнього поточного статусу – вільне, зайняте, заброньоване або на обслуговуванні.

Вимоги FR-06 – FR-09 визначають підсистему бронювання – центральний компонент системи. Процес бронювання (FR-06) складається з послідовності кроків: обрання вільного місця на візуальній сітці, введення часу початку паркування, вибір тривалості (1, 2, 3, 4, 6 або 8 годин), зазначення номерного знаку транспортного засобу [41]. Система автоматично обчислює вартість бронювання як добуток тривалості та погодинного тарифу обраної зони та відображає суму до підтвердження. Перед створенням бронювання виконується перевірка перетину з існуючими активними бронюваннями для обраного місця (FR-07), що виключає подвійне резервування. Після успішного бронювання статус місця автоматично змінюється з «free» на «reserved». Користувач має змогу переглядати перелік своїх бронювань (FR-08) із фільтрацією за статусом та скасовувати активні бронювання (FR-09) з автоматичним поверненням місця у статус «free».

Вимоги FR-10 – FR-11 визначають інтелектуальну підсистему. Модуль прогнозування (FR-10) має приймати на вхід ідентифікатор зони та день тижня (0 – понеділок, 6 – неділя) і повертати масив прогнозних значень завантаженості за кожну годину (з 7:00 до 22:00), що включає середнє, мінімальне та максимальне значення та кількість точок даних, на яких базується розрахунок. Рекомендаційний модуль (FR-11) має приймати географічні координати (широту та довготу) та

повертати перелік із п’яти найближчих паркувальних зон, що мають вільні місця, із зазначенням відстані у кілометрах, кількості вільних місць та тарифу.

Вимоги FR-12 – FR-15 визначають підсистему аналітики та адміністрування. Дашборд моніторингу (FR-12) має відображати агреговані показники: загальна кількість місць, кількість вільних, зайнятих та заброньованих, загальний дохід від бронювань, кількість активних та сьогоднішніх бронювань, а також візуалізації у вигляді кільцевих діаграм (завантаженість), стовпчастих графіків (завантаженість по годинах та дохід по днях) та таблиці зі статусом кожної зони. Адміністративні функції включають CRUD-операції для паркувальних зон (FR-13) із автоматичним генеруванням паркомісць при створенні нової зони, перегляд списку зареєстрованих користувачів (FR-14) та ендпоінт імітації сенсорних даних (FR-15), що змінює стан випадкових паркомісць та записує поточну завантаженість у журнал [42].

Нефункціональні вимоги. Окрім функціональних можливостей, система має відповідати низці якісних характеристик, що визначають її придатність до практичного використання. Нефункціональні вимоги охоплюють аспекти продуктивності, безпеки, масштабованості, портативності, зручності використання, надійності, сумісності та відкритості програмного забезпечення (табл. 2.2).

Таблиця 2.2

Нефункціональні вимоги до системи

ID	Категорія	Опис вимоги
NFR-01	Продуктивність	Час відповіді API не має перевищувати 500 мс для типових запитів при одночасній роботі до 100 користувачів
NFR-02	Безпека	Паролі мають зберігатися у вигляді хешу SHA-256 із унікальною сіллю; доступ до захищених ендпоінтів лише за Bearer-токеном
NFR-03	Масштабованість	Архітектура має дозволяти міграцію з SQLite на PostgreSQL та заміну модулів прогнозування без переробки API

Продовження таблиці 2.2

NFR-04	Портативність	Система має розгортатися на будь-якому сервері з Python 3.10+ та Node.js 18+ без додаткових залежностей від ОС
NFR-05	Зручність використання	Інтерфейс має бути інтуїтивно зрозумілим, адаптованим до темної теми, з часом навчання не більше 5 хвилин
NFR-06	Надійність	Система має забезпечувати цілісність даних через транзакції, зовнішні ключі та перевірочні обмеження (CHECK)
NFR-07	Сумісність	Клієнтська частина має працювати в сучасних браузерах (Chrome, Firefox, Safari, Edge) без додаткових плагінів
NFR-08	Відкритість	Всі компоненти мають використовувати відкриті ліцензії (MIT, BSD, Apache) без залежності від комерційних API

Вимога NFR-01 визначає очікувану продуктивність системи. Час відповіді у 500 мілісекунд є загальноприйнятим порогом для інтерактивних вебзастосунків, що забезпечує комфортну взаємодію користувача без відчутних затримок. Для REST API на базі Flask із SQLite [44] цей показник є реалістичним при оптимізації запитів через індекси бази даних. У розробленій системі створено чотири індекси для таблиць bookings та occupancy_log, що прискорюють найбільш частотні запити: пошук бронювань за користувачем, за місцем, за статусом та пошук журналу завантаженості за зоною, днем тижня та годиною.

Вимога NFR-02 визначає мінімальні стандарти безпеки. Зберігання паролів у відкритому вигляді є критичною вразливістю, тому система використовує одностороннє хешування SHA-256 із додаванням 128-бітної випадкової солі, унікальної для кожного облікового запису. Токен доступу генерується за допомогою криптографічно безпечного генератора secrets.token_hex(32), що забезпечує 256 біт ентропії. Захищені ендпоінти перевіряють наявність та валідність токена через декоратори auth_required та admin_required, що реалізують патерн middleware.

Вимога NFR-03 визначає здатність системи до масштабування. Архітектура побудована таким чином, що серверна частина взаємодіє з базою даних через

стандартний SQL-інтерфейс модуля sqlite3 бібліотеки Python. У разі необхідності переходу на PostgreSQL достатньо замінити рядок підключення та адаптувати окремі SQL-конструкції, специфічні для SQLite (наприклад, AUTOINCREMENT). Модуль прогнозування реалізовано як окремий ендпоінт із чітко визначеним форматом вхідних та вихідних даних, що дозволяє замінити статистичний алгоритм на модель машинного навчання без зміни клієнтського коду [45].

Вимоги NFR-04 – NFR-08 визначають додаткові якісні характеристики. Портативність (NFR-04) забезпечується тим, що SQLite є вбудованою бібліотекою Python і не потребує встановлення окремого серверного процесу СУБД – вся база зберігається у єдиному файлі. Зручність використання (NFR-05) досягається через продуману систему навігації з бічною панеллю, кольорову індикацію статусів, інтуїтивні форми бронювання та контекстні підказки на карті. Надійність (NFR-06) забезпечується механізмами транзакційної безпеки SQLite (режим WAL), зовнішніми ключами з каскадним видаленням та обмеженнями CHECK на рівні схеми бази даних, що унеможливлюють внесення некоректних значень статусів, ролей та типів. Сумісність (NFR-07) досягається використанням стандартних веб-технологій (HTML5, CSS3, JavaScript ES6+) без залежності від браузерних плагінів. Відкритість (NFR-08) забезпечується вибором виключно відкритих компонентів: Flask (BSD), React (MIT), Leaflet (BSD), SQLite (Public Domain), OpenStreetMap (ODbL) [46].

Основні сценарії використання системи. На підставі сформульованих функціональних вимог визначено ключові сценарії використання (Use Cases), що описують типову взаємодію акторів із системою:

UC-01 «Пошук та бронювання паркомісця»: водій відкриває карту, обирає зону, переглядає вільні місця на візуальній сітці, натискає на вільне місце, заповнює форму бронювання (час, тривалість, номерний знак), підтверджує бронювання, отримує підтвердження з розрахованою вартістю.

UC-02 «Отримання рекомендації»: водій натискає кнопку «Рекомендації AI», система визначає п’ять найближчих зон із вільними місцями та відображає їх із

зазначенням відстані, кількості місць та тарифу, водій обирає зону та переходить до бронювання.

UC-03 «Прогнозування завантаженості»: водій обирає зону та день тижня на сторінці прогнозування, система відображає стовпчасту діаграму середньої завантаженості за кожну годину та таблицю з детальними показниками та рекомендаціями («Рекомендовано», «Помірно», «Уникайте»).

UC-04 «Моніторинг стану (адміністратор)»: адміністратор переглядає дашборд з KPI, аналізує графіки завантаженості по годинах та доходу по днях, переглядає таблицю стану зон із індикаторами завантаженості.

UC-05 «Управління паркувальними зонами (адміністратор)»: адміністратор створює нову зону (назва, адреса, координати, кількість місць, тариф, тип), система автоматично генерує паркомісця з унікальними номерами та ідентифікаторами сенсорів; адміністратор також може редагувати параметри існуючих зон або видаляти зони з каскадним видаленням пов’язаних місць [47].

Сформульований набір із 15 функціональних та 8 нефункціональних вимог повністю визначає межі та якісні характеристики розроблюваної системи. Функціональні вимоги охоплюють п’ять підсистем: автентифікацію, моніторинг, бронювання, інтелектуальні функції та адміністрування, що забезпечує комплексне вирішення проблеми управління паркувальним простором. Нефункціональні вимоги гарантують продуктивність, безпеку, масштабованість та портативність рішення. Визначені сценарії використання слугуватимуть основою для проєктування архітектури, бази даних та API системи у наступних підрозділах.

2.2 Проєктування архітектури системи та реляційної бази даних

Проєктування архітектури є ключовим етапом розробки програмної системи, що визначає способи організації компонентів, взаємодію між ними та потоки даних. Архітектура розроблюваної інтелектуальної системи моніторингу та бронювання паркувальних місць базується на класичній трирівневій клієнт-серверній моделі, доповненій рівнем IoT-пристроїв для збору сенсорних даних.

Загальна архітектура системи. Система складається з чотирьох логічних рівнів, кожен з яких відповідає за окрему групу функцій (рис. 2.1). Клієнтський рівень реалізований як React SPA-додаток, що виконується у веббраузері та взаємодіє із серверною частиною через HTTP-запити у форматі JSON. Серверний рівень побудований на базі мікрофреймворку Flask і включає шість функціональних модулів: автентифікації, бронювання, прогнозування, рекомендацій, аналітики та управління зонами. Рівень даних представлений реляційною базою SQLite [48], що зберігає п’ять основних таблиць. IoT-рівень забезпечує надходження даних про стан паркомісць від сенсорних вузлів; на етапі прототипування він реалізований через модуль програмної імітації.

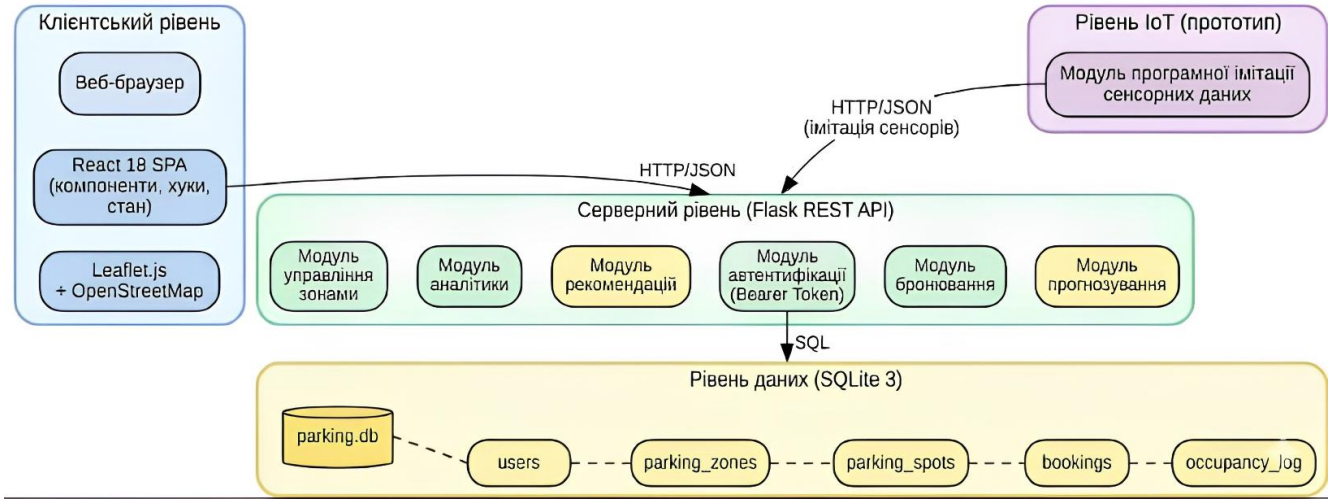


Рис. 2.1. Загальна архітектура системи SmartParking

Взаємодія між рівнями здійснюється через чітко визначені інтерфейси. Клієнтський додаток надсилає HTTP-запити до REST API серверної частини, передаючи токен авторизації у заголовку Authorization кожного запиту. Серверна частина обробляє запити, виконує бізнес-логіку (валідацію, обчислення вартості, перевірку перетинів, прогнозування) та формує SQL-запити до бази даних. Відповіді повертаються у форматі JSON із відповідними HTTP-кодами стану: 200 для успішних запитів, 201 для створення ресурсу, 400 для помилок валідації, 401 для помилок автентифікації, 403 для недостатніх прав, 404 для відсутніх ресурсів та 409 для конфліктів (наприклад, подвійне бронювання).

Діаграма варіантів використання. Для формалізації взаємодії акторів із системою побудовано діаграму варіантів використання (рис. 2.2). Система передбачає двох акторів: водій та адміністратор. Водій має доступ до дев’яти варіантів використання: реєстрація, автентифікація, перегляд карти паркувальних зон, перегляд деталей зони та місць, бронювання паркомісця, перегляд бронювань, скасування бронювання, прогнозування завантаженості та отримання рекомендацій. Адміністратор, окрім автентифікації та перегляду бронювань, має доступ до панелі моніторингу (дашборду), управління зонами (CRUD-операції) та перегляду списку користувачів.

Рис. 2.2. Діаграма варіантів використання системи SmartParking

Модульна структура серверної частини. Серверна частина системи організована за принципом функціональної модульності, де кожен модуль

відповідає за окрему підсистему та реалізує відповідний набір REST-ендпоінтів. Модуль автентифікації обробляє реєстрацію, вхід, верифікацію токена та перевірку ролей через декоратори auth_required та admin_required. Модуль бронювання реалізує створення, перегляд та скасування бронювань із перевіркою доступності місця та перетинів часу. Модуль прогнозування виконує агрегаційні SQL-запити до таблиці occupancy_log [49] та повертає прогнозні значення завантаженості. Модуль рекомендацій здійснює пошук найближчих зон із вільними місцями на основі евклідової відстані. Модуль аналітики формує агреговані показники для дашборду. Модуль управління зонами реалізує CRUD-операції з автоматичним генеруванням місць при створенні нової зони.

Загалом серверна частина реалізує 15 REST-ендпоінтів, розподілених за модулями (табл. 2.3).

Таблиця 2.3

REST API ендпоінти системи

Метод	Шлях	Опис	Доступ
POST	/api/auth/register	Реєстрація	Публічний
POST	/api/auth/login	Автентифікація	Публічний
GET	/api/auth/me	Поточний користувач	Авторизований
GET	/api/zones	Список зон	Публічний
GET	/api/zones/:id	Деталі зони + місця	Публічний
POST	/api/zones	Створити зону	Адмін
PUT	/api/zones/:id	Оновити зону	Адмін
DELETE	/api/zones/:id	Видалити зону	Адмін
POST	/api/bookings	Створити бронювання	Авторизований
GET	/api/bookings	Список бронювань	Авторизований
PUT	/api/bookings/:id/cancel	Скасувати	Авторизований
GET	/api/predict/:zone_id	Прогнозування	Публічний
GET	/api/recommend	Рекомендації	Публічний
GET	/api/analytics/dashboard	Дашборд KPI	Авторизований
POST	/api/simulate/update	Імітація сенсорів	Публічний

Проєктування реляційної бази даних. Структура бази даних розроблена відповідно до принципів реляційної моделі та нормалізована до третьої нормальної форми (3НФ). База даних складається з п’яти таблиць, що описують основні

сутності предметної області: користувачів, паркувальні зони, паркомісця, бронювання та журнал завантаженості. Діаграму «сутність–зв’язок» (ER-діаграму) бази даних представлено на рис. 2.3.

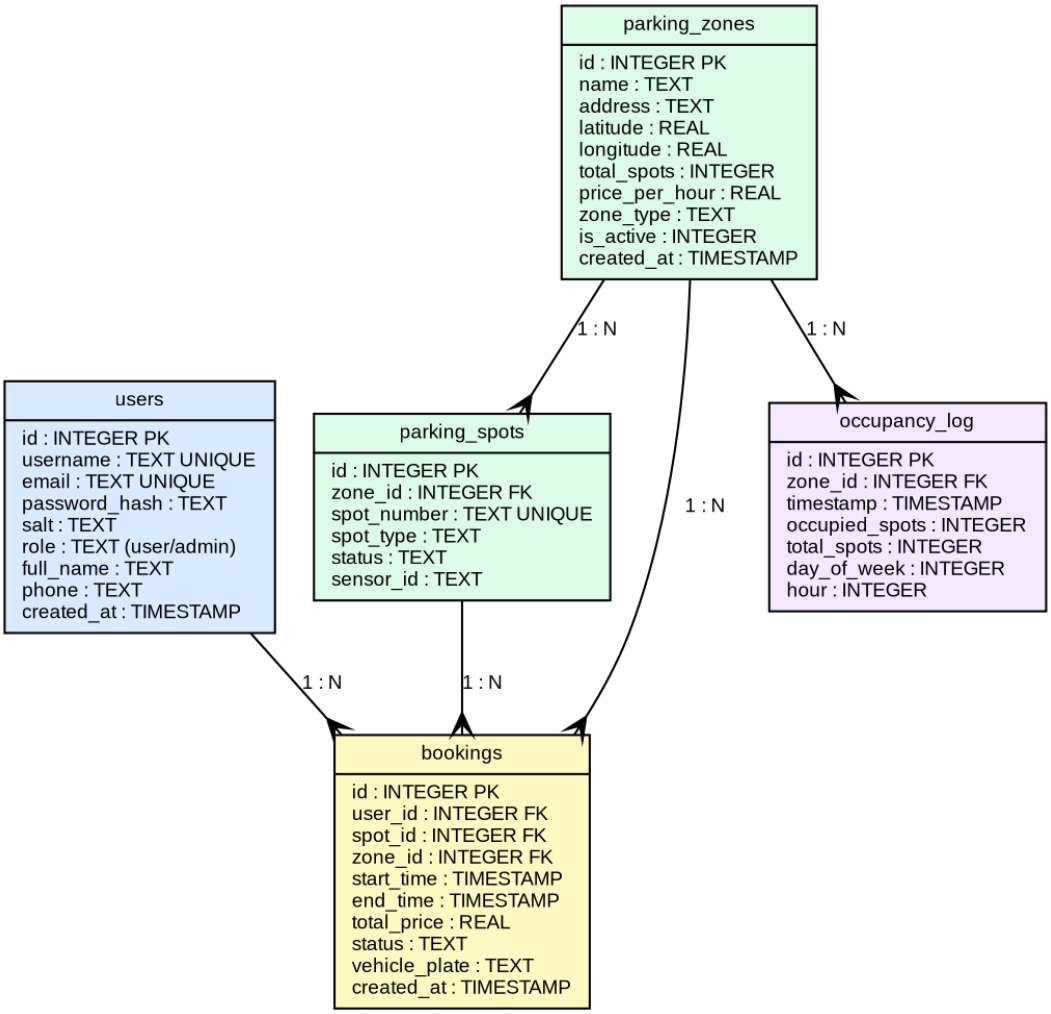


Рис. 2.3. ER-діаграма бази даних системи SmartParking

Таблиця users зберігає облікові записи користувачів. Поля username та email мають обмеження унікальності (UNIQUE), що виключає дублювання акаунтів. Пароль зберігається у вигляді SHA-256 хешу (поле password_hash) разом з унікальною сіллю (поле salt). Поле role визначає роль користувача із обмеженням CHECK, що дозволяє лише значення «user» або «admin».

Таблиця parking_zones описує паркувальні зони міста. Координати (latitude, longitude) використовуються для відображення зон на карті та обчислення відстаней у рекомендаційному модулі. Поле zone_type класифікує тип зони із

CHECK-обмеженням на чотири дозволені значення: street (вулична), underground (підземна), multi_level (багаторівнева) та open (відкрита). Поле is_active дозволяє деактивувати зону без видалення.

Таблиця parking_spots зберігає інформацію про кожне окреме паркомісце. Зв'язок із таблицею parking_zones реалізовано через зовнішній ключ zone_id із каскадним видаленням (ON DELETE CASCADE), що забезпечує автоматичне видалення місць при видаленні зони. Поле status визначає поточний стан місця: free, occupied, reserved або maintenance. Композитне обмеження UNIQUE(zone_id, spot_number) гарантує унікальність номера місця в межах зони. Поле sensor_id зберігає ідентифікатор IoT-сенсора, що контролює дане місце.

Таблиця bookings є центральною транзакційною таблицею системи. Вона зв'язана трьома зовнішніми ключами: user_id → users (хто бронює), spot_id → parking_spots (яке місце), zone_id → parking_zones (в якій зоні). Поля start_time та end_time визначають часовий інтервал бронювання, total_price зберігає автоматично розраховану вартість. Статус бронювання (active, completed, cancelled) контролюється CHECK-обмеженням.

Таблиця occupancy_log є ключовою для роботи модуля прогнозування. Вона зберігає знімки завантаженості кожної зони з фіксацією часу, кількості зайнятих та загальних місць, а також дня тижня (0–6) та години (0–23). Поля day_of_week та hour винесено в окремі стовпці для оптимізації агрегаційних запитів прогнозування, що дозволяє уникнути витратних операцій парсингу дати на рівні SQL. Для прискорення запитів до цієї таблиці створено складений індекс idx_occupancy_zone(zone_id, day_of_week, hour).

Загалом у базі даних визначено чотири індекси для оптимізації найбільш частотних запитів: idx_bookings_user (пошук бронювань за користувачем), idx_bookings_spot (перевірка перетинів за місцем), idx_bookings_status (фільтрація за статусом) та idx_occupancy_zone (агрегація даних прогнозування). Цілісність даних забезпечується п'ятьма зовнішніми ключами з підтримкою каскадних операцій та шістьма CHECK-обмеженнями на рівні схеми.

2.3 Розробка алгоритму прогнозування завантаженості паркувальних зон на основі статистичного аналізу

Інтелектуальна складова системи реалізується через модуль прогнозування завантаженості, який на основі аналізу накопичених історичних даних визначає очікуваний рівень зайнятості паркувальної зони для кожної години обраного дня тижня. Цей модуль є ключовим елементом, що відрізняє розроблювану систему від більшості існуючих рішень і обґрунтовує використання терміну «інтелектуальна» в назві роботи.

Постановка задачі. Задача прогнозування завантаженості формулюється таким чином: для заданої паркувальної зони z та дня тижня d (де 0 відповідає понеділку, 6 – неділі) необхідно визначити очікуваний рівень завантаженості O(z, d, h) для кожної години h робочого періоду (з 7:00 до 22:00). Рівень завантаженості визначається як відношення кількості зайнятих місць до загальної кількості місць у зоні, виражене у відсотках. Результатом роботи алгоритму є масив прогнозних значень, кожен елемент якого містить середнє, мінімальне та максимальне значення завантаженості, а також кількість точок даних, на яких базується розрахунок.

Джерело даних. Вхідними даними для алгоритму є записи таблиці occupancy_log бази даних. Кожен запис фіксує кількість зайнятих та загальних місць у конкретній зоні на певний момент часу. Дані накопичуються двома шляхами: через регулярне журналювання при кожному оновленні стану сенсорів (або їхній імітації) та через первинне заповнення бази демонстраційними даними за 30 діб при ініціалізації системи. Демонстраційні дані моделюють реалістичні патерни завантаженості: підвищений попит у робочі дні з 8:00 до 18:00 із піками о 11:00–14:00, та помірний попит у вихідні з 10:00 до 16:00.

Опис алгоритму. Для прогнозування обрано статистичний метод на основі обчислення агрегованих показників. Цей підхід обґрунтований тим, що паркувальний попит має яскраво виражену циклічну структуру: робочі та вихідні дні суттєво відрізняються за патернами завантаженості, а протягом дня

спостерігаються стабільні ранкові, денні та вечірні піки. Блок-схему алгоритму представлено на рис. 2.4.

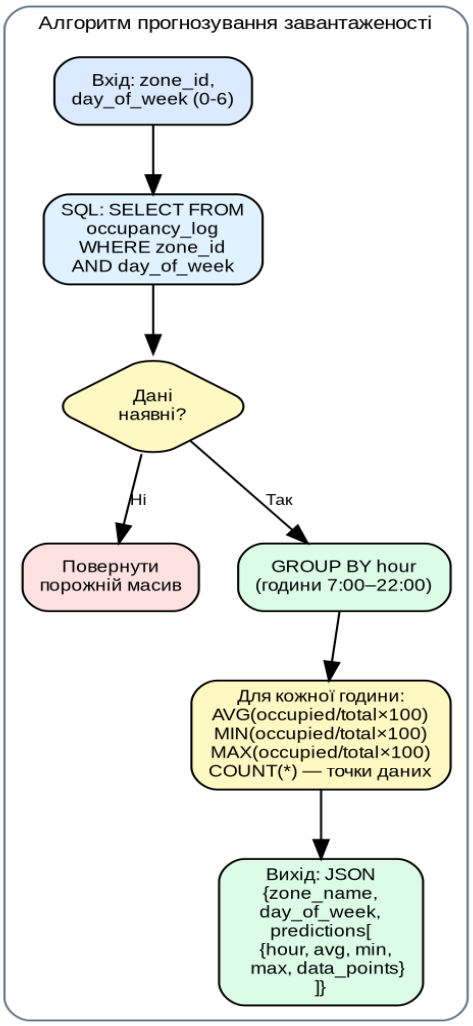


Рис. 2.4. Блок-схема алгоритму прогнозування завантаженості

Алгоритм виконується у такій послідовності. На першому кроці приймаються вхідні параметри: ідентифікатор зони zone_id та день тижня day_of_week. На другому кроці виконується вибірка з таблиці occupancy_log усіх записів, що відповідають заданій зоні та дню тижня. На третьому кроці виконується перевірка наявності даних; у разі відсутності записів повертається порожній масив із відповідним повідомленням. На четвертому кроці дані групуються за полем hour (година доби). На п’ятому кроці для кожної групи обчислюються чотири агреговані показники: середнє значення завантаженості AVG(occupied_spots / total_spots × 100), мінімальне значення MIN, максимальне значення MAX та кількість точок

даних COUNT [51]. На шостому кроці формується JSON-відповідь із масивом прогнозних значень.

Математично середня завантаженість для години h обчислюється за формулою:

$$\bar{O}(z, d, h) = (1/N) \times \Sigma (occupied\_i / total\_i) \times 100\%, \quad (2.1)$$

де N – кількість спостережень для заданої комбінації (зона, день тижня, година); occupied_i та total_i – кількість зайнятих та загальних місць відповідно для і-го спостереження.

Реалізація на рівні SQL [54]. Обчислення виконується одним SQL-запитом із використанням агрегатних функцій, що забезпечує максимальну ефективність за рахунок обробки даних безпосередньо на рівні СУБД без необхідності передачі великих обсягів сирих даних у прикладний код. Запит використовує складений індекс idx_occupancy_zone(zone_id, day_of_week, hour), що забезпечує час виконання O(log N) для кожної групи:

```
SELECT hour, ROUND(AVG(occupied_spots * 1.0 / total_spots) * 100, 1) AS avg_occupancy, MIN(...) AS min_occupancy, MAX(...) AS max_occupancy, COUNT(*) AS data_points FROM occupancy_log WHERE zone_id=? AND day_of_week=? GROUP BY hour ORDER BY hour
```

Послідовність бронювання паркомісця. Окрім алгоритму прогнозування, важливим аспектом проєктування є формалізація послідовності взаємодії компонентів системи при виконанні ключового сценарію – бронювання паркомісця.

Діаграму послідовності цього процесу представлено на рис. 2.5.

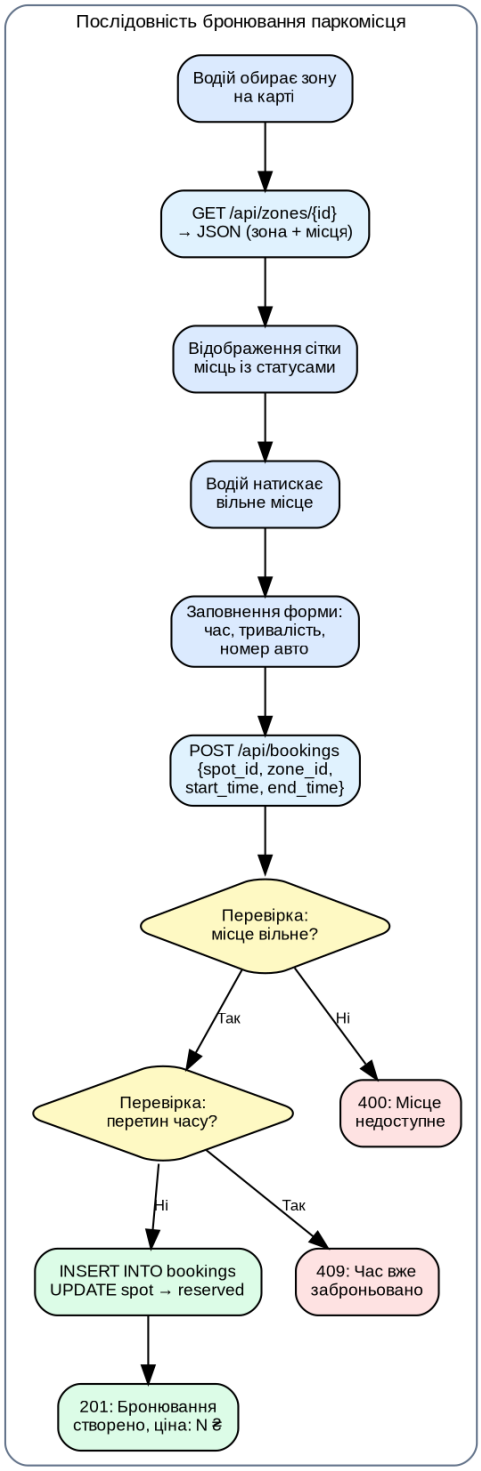


Рис. 2.5. Діаграма послідовності процесу бронювання паркомісця

Процес бронювання включає двоетапну валідацію: перевірку фізичної доступності місця (статус «free») та перевірку відсутності часових перетинів із існуючими бронюваннями. Лише після успішного проходження обох перевірок створюється запис у таблиці bookings та оновлюється статус місця. Такий підхід

запобігає конфліктним ситуаціям, коли два користувачі намагаються забронювати одне місце одночасно.

Алгоритм рекомендацій. Рекомендаційний модуль реалізує пошук п’яти найближчих паркувальних зон, що мають вільні місця, відносно заданих координат користувача. Відстань обчислюється за спрощеною формулою евклідової відстані з коефіцієнтом перетворення градусів у кілометри:

$$D(z) = \sqrt{((lat\_z - lat\_u)^2 + (lng\_z - lng\_u)^2)} \times 111 \text{ км}, \quad (2.2)$$

де lat_z, lng_z – координати центру паркувальної зони; lat_u, lng_u – координати користувача; коефіцієнт 111 – наближена кількість кілометрів в одному градусі широти. Запит виконується з фільтрацією за умовою наявності вільних місць (підзапит COUNT із status=«free» > 0), сортуванням за зростанням відстані та обмеженням результату п’ятьма записами (LIMIT 5). Результат включає назву зони, адресу, кількість вільних місць, тариф та обчислену відстань у кілометрах.

Класифікація рекомендацій прогнозу. Для зручності інтерпретації результатів прогнозування на клієнтській стороні реалізовано систему класифікації, що перетворює числове значення середньої завантаженості в якісну рекомендацію. Якщо прогнозована завантаженість не перевищує 60 відсотків, відображається рекомендація «Рекомендовано» зеленим кольором. При завантаженості від 60 до 85 відсотків – «Помірно» жовтим кольором. При завантаженості понад 85 відсотків – «Уникайте» червоним кольором. Така тристадійна класифікація дозволяє водію швидко оцінити доцільність паркування у обраній зоні в обраний час без необхідності аналізу числових значень.

Спроєктовано архітектуру системи на базі чотирирівневої клієнт-серверної моделі з модульною організацією серверної частини, реляційну базу даних із п’яти таблиць у третій нормальній формі, REST API із 15 ендпоінтами, алгоритм прогнозування завантаженості на основі статистичного аналізу та рекомендаційний алгоритм пошуку найближчих вільних паркувальних зон. Усі компоненти проєктування безпосередньо пов’язані з функціональними вимогами, визначеними у підрозділі 2.1, та створюють основу для програмної реалізації, що буде описана у третьому розділі.

2.4 Проєктування REST API для взаємодії компонентів системи

REST (Representational State Transfer) – це архітектурний стиль побудови розподілених систем, запропонований Роєм Філдінгом у 2000 році [37].

Ключовими принципами REST є відсутність стану між запитами (stateless), уніфіковані ідентифікатори ресурсів (URI), стандартні HTTP-методи для операцій над ресурсами та представлення даних у форматі JSON. У розроблюваній системі REST API виконує функцію єдиної точки взаємодії між клієнтською частиною (React SPA) та серверною бізнес-логікою (Flask), забезпечуючи повне розділення відповідальностей між рівнями архітектури [52].

Використання архітектури REST у поєднанні з форматом обміну даними JSON створює оптимальні умови для швидкої та надійної взаємодії між компонентами системи. Формат JSON є нативним для мови JavaScript, що використовується у клієнтському React-додатку, і водночас легко серіалізується та десеріалізується стандартними бібліотеками Python на стороні Flask-сервера. Крім того, принцип відсутності стану (stateless) означає, що сервер не зберігає інформацію про сесію клієнта у своїй пам'яті між запитами. Це значно спрощує масштабування серверної частини в майбутньому та знижує навантаження на обчислювальні ресурси, оскільки кожен запит самодостатній і містить усю необхідну контекстну інформацію для його повноцінної обробки.

Для забезпечення безпечної взаємодії у такому розподіленому середовищі особлива увага приділяється механізмам авторизації та валідації вхідних даних. Оскільки клієнт і сервер функціонують незалежно (часто на різних доменах або портах), процеси перевірки прав доступу реалізуються за допомогою токенів (наприклад, JWT), які автоматично додаються клієнтом у заголовки HTTP-запитів.

Це дозволяє серверній системі маршрутизації ефективно перевіряти повноваження користувача та відхиляти нелегітимні звернення ще до етапу виконання ресурсомісткої бізнес-логіки чи звернення до бази даних. Саме тому механізм проходження запиту від клієнта до сервера є багатоетапним і вимагає чіткої регламентації.

Життєвий цикл HTTP-запиту. Кожен запит від клієнтської частини проходить послідовність етапів обробки на серверній стороні, що забезпечують безпеку, маршрутизацію, валідацію та формування відповіді. Діаграму життєвого циклу запиту представлено на рис. 2.6.

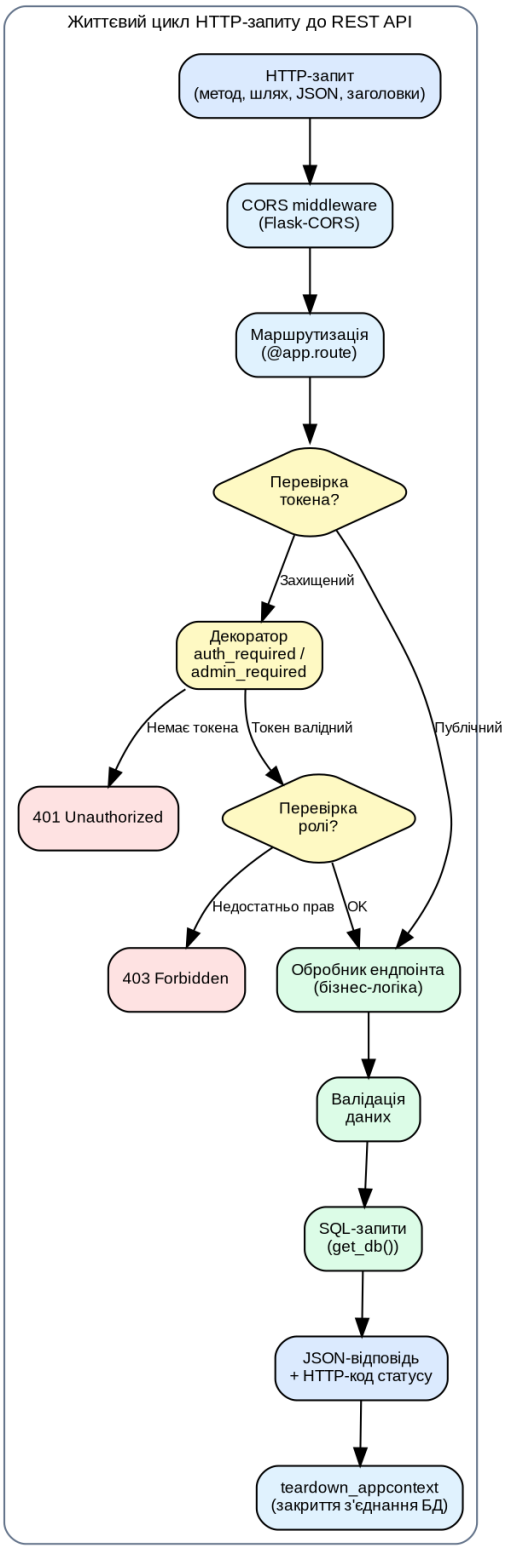


Рис. 2.6. Життєвий цикл HTTP-запиту до REST API системи

На першому етапі запит оброблюється middleware Flask-CORS, що додає заголовки Access-Control-Allow-Origin для забезпечення крос-доменних запитів між клієнтом (порт 3000) та сервером (порт 5000). На другому етапі виконується маршрутизація – Flask зіставляє URL та HTTP-метод запиту з відповідним обробником через декоратори @app.route. На третьому етапі, якщо ендпоінт є захищеним, перевіряється наявність та валідність Bearer-токена через декоратори auth_required або admin_required. При відсутності токена повертається HTTP 401, при недостатніх правах – HTTP 403. На четвертому етапі виконується обробник ендпоінта, що містить бізнес-логіку: валідацію вхідних даних, SQL-запити до бази та формування JSON-відповіді. На п’ятому етапі, після завершення обробки, Flask викликає teardown_appcontext, що автоматично закриває з’єднання з базою даних.

Проєктування API автентифікації. Підсистема автентифікації реалізує три ендпоінти: реєстрацію, вхід та отримання даних поточного користувача. Діаграму процесів реєстрації та автентифікації представлено на рис. 2.7.

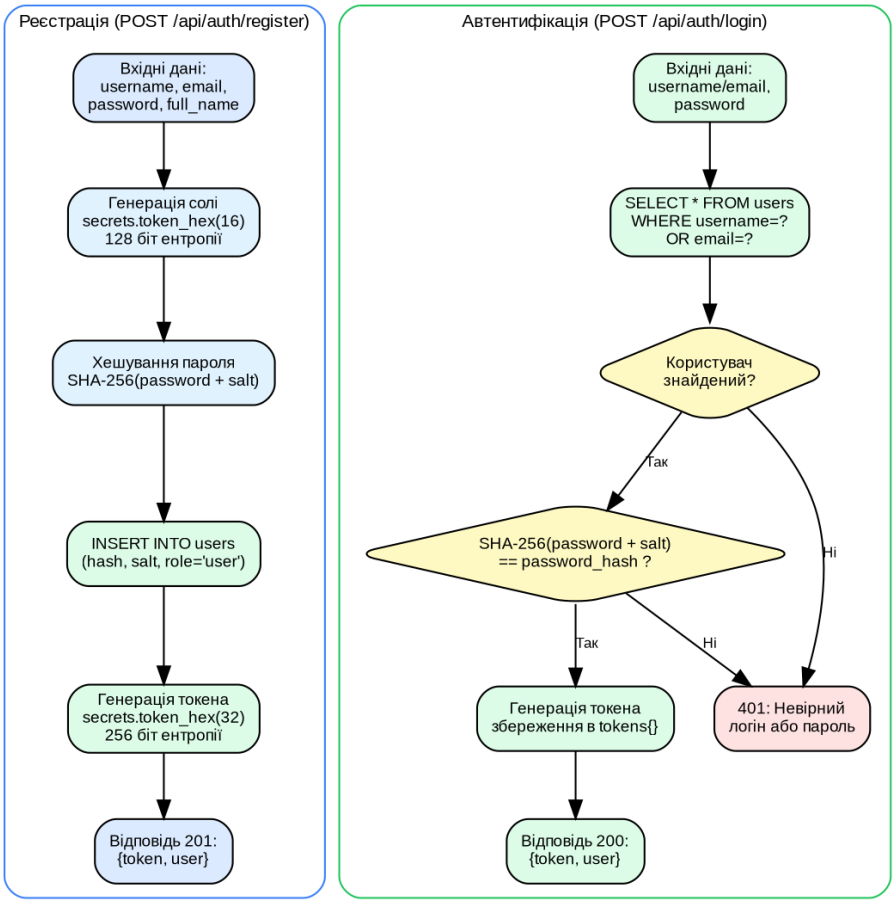


Рис. 2.7. Діаграма процесів реєстрації та автентифікації

При реєстрації (POST /api/auth/register) система приймає JSON-об’єкт із полями username, email, password та full_name. Пароль не зберігається у відкритому вигляді – для кожного користувача генерується унікальна 128-бітна сіль через secrets.token_hex(16), після чого обчислюється хеш SHA-256 від конкатенації пароля та солі. У базі зберігаються лише хеш та сіль. Після успішного створення запису генерується 256-бітний токен доступу, що повертається клієнту разом з даними користувача.

Формат запиту реєстрації:

```
POST /api/auth/register
Content-Type: application/json
{
  "username": "ivan",
  "email": "ivan@mail.com",
  "password": "pass123",
  "full_name": "Іван Петренко"
}
```

Формат відповіді (HTTP 201):

```
{
  "token": "a4f8c2...64 символи",
  "user": {"id": 3, "username": "ivan",
    "role": "user",
    "full_name": "Іван Петренко"}
}
```

При автентифікації (POST /api/auth/login) система приймає логін або email та пароль, знаходить користувача в базі, обчислює хеш від введеного пароля із збереженою сіллю та порівнює з хешем у базі. У разі збігу генерується новий токен. У разі невідповідності повертається HTTP 401 з узагальненим повідомленням «Невірний логін або пароль» без уточнення, що саме не збігається, – це є стандартною практикою безпеки для запобігання перебору облікових записів.

Ендпоінт GET /api/auth/me повертає дані поточного авторизованого користувача на основі токена із заголовка Authorization. Цей ендпоінт

використовується клієнтською частиною для перевірки валідності збереженого токена при завантаженні додатку.

Проєктування API управління паркувальними зонами. Підсистема управління зонами реалізує п’ять ендпоінтів за патерном CRUD (Create, Read, Update, Delete). Публічні ендпоінти GET /api/zones та GET /api/zones/:id доступні без авторизації, що дозволяє переглядати інформацію про паркування навіть незареєстрованим користувачам. Адміністративні ендпоінти POST, PUT та DELETE захищені декоратором admin_required [51].

Ендпоінт GET /api/zones повертає масив усіх активних паркувальних зон з агрегованою статистикою. Для кожної зони виконуються три корельовані підзапити, що обчислюють кількість вільних, зайнятих та заброньованих місць безпосередньо на рівні SQL. Такий підхід дозволяє уникнути проблеми N+1 запитів, коли для кожної із N зон довелося б виконувати окремий запит до таблиці parking_spots.

Ендпоінт GET /api/zones/:id повертає детальну інформацію про конкретну зону, включаючи повний перелік паркомісць із їхніми статусами, типами та ідентифікаторами сенсорів. Ці дані використовуються клієнтською частиною для побудови візуальної сітки місць із кольоровою індикацією.

При створенні нової зони (POST /api/zones) система автоматично генерує паркомісця у кількості, зазначеній у полі total_spots. Кожному місцю присвоюється унікальний номер за форматом «{zone_id}-{порядковий_номер}» та ідентифікатор сенсора за форматом «SNS-{zone_id}-{порядковий_номер}». Це забезпечує готовність зони до використання одразу після створення без додаткових кроків конфігурації.

Проєктування API бронювання. Підсистема бронювання є найбільш критичною з точки зору цілісності даних, оскільки оперує фінансовими транзакціями та має запобігати конфліктним ситуаціям. Підсистема реалізує три ендпоінти: створення, перегляд та скасування бронювань.

Ендпоінт POST /api/bookings реалізує двоетапну валідацію. На першому етапі перевіряється фізична доступність місця – виконується запит SELECT з умовою

status=’free’. Якщо місце вже зайняте або заброньоване, повертається HTTP 400. На другому етапі перевіряється відсутність часових перетинів з існуючими активними бронюваннями для цього місця за допомогою SQL-умови:

```
WHERE spot_id=? AND status='active'
  AND start_time < ? AND end_time > ?
```

Ця умова реалізує класичний алгоритм перевірки перетину часових інтервалів: два інтервали [A, B] та [C, D] перетинаються тоді і тільки тоді, коли A < D і C < B. У разі виявлення перетину повертається HTTP 409 (Conflict). Після успішної валідації вартість обчислюється як добуток тривалості у годинах та тарифу зони, створюється запис у таблиці bookings та статус місця оновлюється на ’reserved’.

Формат запиту бронювання:

```
POST /api/bookings
Authorization: Bearer <token>
{
  "spot_id": 42, "zone_id": 1,
  "start_time": "2026-04-02T10:00:00",
  "end_time": "2026-04-02T12:00:00",
  "vehicle_plate": "AA1234BB"
}
```

Формат відповіді (HTTP 201):

```
{
  "id": 51,
  "total_price": 120.0,
  "message": "Бронювання створено успішно"
}
```

Ендпоінт GET /api/bookings реалізує рольову логіку відображення: адміністратор отримує всі бронювання системи з інформацією про користувачів (JOIN users), а звичайний користувач – лише свої. Запит використовує JOIN з

трьома таблицями (parking_zones, users, parking_spots) для збагачення даних назвами зон, номерами місць та іменами користувачів.

Ендпоінт PUT /api/bookings/:id/cancel виконує перевірку прав: користувач може скасувати лише свої бронювання, адміністратор – будь-які. При скасуванні статус бронювання змінюється на ’cancelled’, а статус місця повертається на ’free’. Обидві операції виконуються в межах однієї транзакції SQLite.

Діаграми станів. Для формалізації поведінки ключових сутностей системи побудовано діаграми переходів станів паркомісця та бронювання (рис. 2.8).

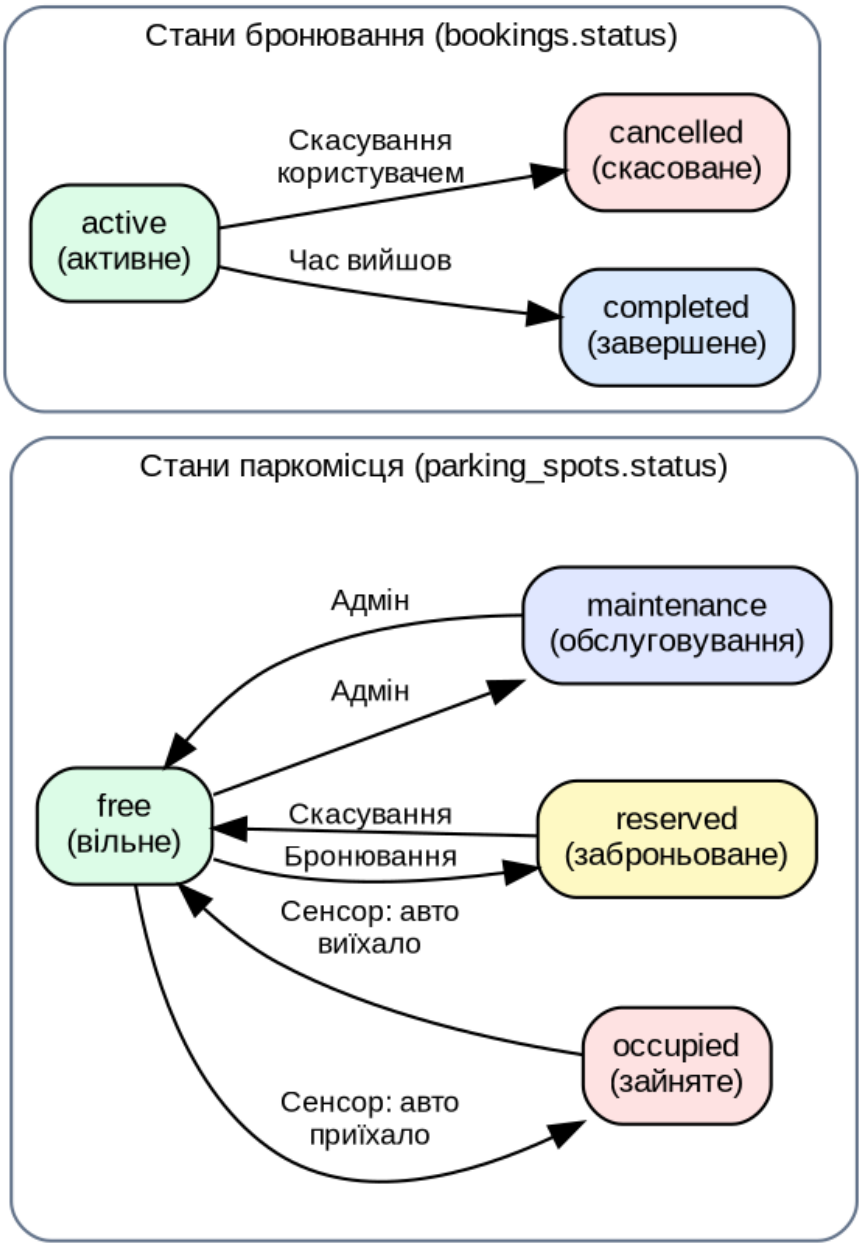


Рис. 2.8. Діаграми переходів станів паркомісця та бронювання

Паркомісце може перебувати в чотирьох станах: free (вільне), occupied (зайняте), reserved (заброньоване) та maintenance (на обслуговуванні). Перехід free → reserved відбувається при створенні бронювання, reserved → free – при скасуванні. Переходи free ↔ occupied моделюють реакцію на дані сенсорів (або їхню імітацію). Стан maintenance встановлюється адміністратором вручну. Бронювання має три стани: active (активне), completed (завершене) та cancelled (скасоване). Перехід active → completed відбувається після закінчення часу бронювання, active → cancelled – при скасуванні користувачем або адміністратором. Стани completed та cancelled є термінальними і не допускають подальших переходів.

Проєктування API прогнозування та рекомендацій. Ендпоінт GET /api/predict/:zone_id приймає ідентифікатор зони як параметр шляху та необов'язковий параметр day (день тижня, 0–6) як query-параметр. За замовчуванням використовується поточний день. Відповідь включає назву зони, день тижня та масив прогнозних значень за кожну годину з полями: hour (година), avg_occupancy (середня завантаженість, %), min_occupancy (мінімальна), max_occupancy (максимальна) та data_points (кількість спостережень).

Формат відповіді прогнозування:

```
{
  "zone_id": 1,
  "zone_name": "Парковка Хрещатик",
  "day_of_week": 1,
  "predictions": [
    {"hour": 7, "avg_occupancy": 32.5,
     "min_occupancy": 18.0,
     "max_occupancy": 45.0,
     "data_points": 4},
    {"hour": 8, "avg_occupancy": 78.3, ...},
    ...
  ]
}
```

Ендпоінт GET /api/recommend приймає координати користувача (lat, lng) як query-параметри та повертає масив із п’яти найближчих паркувальних зон, що мають вільні місця. Відповідь для кожної зони включає повну інформацію (назва, адреса, тариф, тип), кількість вільних місць та обчислену відстань у кілометрах. Запит виконується одним SQL-виразом із підзапитом для підрахунку вільних місць, обчисленням квадрата відстані, фільтрацією та сортуванням.

Проєктування API аналітики. Ендпоінт GET /api/analytics/dashboard є найбільш складним з точки зору кількості SQL-запитів – він виконує 10 незалежних запитів для формування комплексної аналітичної зведення. Відповідь включає чотири групи показників:

Перша група – загальна статистика: кількість активних зон, загальна кількість місць, розподіл за статусами (вільні, зайняті, заброньовані). Друга група – статистика бронювань: загальна кількість, кількість активних, кількість за сьогодні, загальний дохід. Третя група – динаміка доходу: масив з доходом та кількістю бронювань за кожен день останніх 14 днів. Четверта група – завантаженість: статистика за кожною зоною (вільні, зайняті, заброньовані) та середня завантаженість по годинах для робочих днів (понеділок – п’ятниця).

Стандартизація обробки помилок. Усі ендпоінти системи дотримуються єдиного формату повідомлень про помилки. У разі виникнення помилки відповідь містить JSON-об’єкт із полем error, що містить текстовий опис проблеми українською мовою. HTTP-код статусу відповідає семантиці помилки відповідно до стандарту RFC 7231. У табл. 2.4 наведено перелік HTTP-кодів, що використовуються у системі.

Таблиця 2.4

HTTP-коди статусу відповідей REST API

Код	Назва	Контекст використання
200	OK	Успішне виконання GET, PUT-запитів
201	Created	Успішне створення ресурсу (реєстрація, бронювання, створення зони)

Продовження таблиці 2.4

400	Bad Request	Некоректні вхідні дані, відсутні обов'язкові поля, місце недоступне
401	Unauthorized	Відсутній або невалідний токен, невірний логін або пароль
403	Forbidden	Недостатньо прав (користувач намагається виконати адмін-операцію)
404	Not Found	Зону або бронювання не знайдено за вказаним ідентифікатором
409	Conflict	Дублювання username/email при реєстрації, часовий перетин бронювань

Забезпечення крос-доменної взаємодії. Оскільки клієнтська та серверна частини працюють на різних портах (3000 та 5000 відповідно), браузер за замовчуванням блокує HTTP-запити між ними через політику Same-Origin Policy. Для вирішення цієї проблеми на серверній стороні застосовується розширення Flask-CORS, що автоматично додає заголовки Access-Control-Allow-Origin, Access-Control-Allow-Methods та Access-Control-Allow-Headers до кожної відповіді. У продакшн-середовищі проксіювання запитів через Vite (конфігурація proxy у vite.config.js) дозволяє уникнути CORS-проблем повністю, оскільки клієнт та API обслуговуються з одного домену.

Проєктування API імітації сенсорних даних. Ендпоінт POST /api/simulate/update моделює надходження даних від IoT-сенсорів для демонстрації та тестування системи. При виклику алгоритм перебирає всі місця зі статусами free та occupied і з імовірністю 10 відсотків змінює їхній стан на протилежний, імітуючи приїзд або виїзд автомобіля. Після оновлення статусів для кожної активної зони створюється запис у таблиці occupancy_log з поточною кількістю зайнятих місць, днем тижня та годиною. Ці записи використовуються модулем прогнозування для побудови статистичних моделей. Відповідь містить кількість змінених місць, що дозволяє верифікувати коректність роботи імітації.

Аспекти безпеки API. При проєктуванні REST API враховано типові вектори атак на вебзастосунки. Захист від SQL-ін’єкцій забезпечується використанням параметризованих запитів (placeholder ‘?’) у всіх SQL-виразах – значення ніколи не конкатенуються безпосередньо в рядок запиту. Захист від перебору паролів реалізовано через узагальнені повідомлення про помилки, що не розкривають, чи існує обліковий запис із вказаним логіном. Захист від несанкціонованого доступу забезпечується перевіркою токена та ролі на рівні кожного захищеного ендпоінта через декоратори-middleware. Перевірочні обмеження CHECK на рівні схеми бази даних гарантують, що навіть у разі обходу серверної валідації некоректні значення статусів, ролей та типів не будуть збережені.

Спроєктовано REST API із 15 ендпоінтами, що охоплює п’ять функціональних підсистем: автентифікацію (3 ендпоінти), управління зонами (5 ендпоінтів), бронювання (3 ендпоінти), інтелектуальні функції (2 ендпоінти) та аналітику й імітацію (2 ендпоінти). API побудований відповідно до принципів REST із використанням стандартних HTTP-методів, уніфікованого формату помилок та Bearer-токенів для авторизації. Кожен ендпоінт задокументовано з визначенням формату запиту, формату відповіді, можливих кодів помилок та рівня доступу.

Висновки до розділу 2

У другому розділі кваліфікаційної роботи виконано проєктування інтелектуальної системи моніторингу та бронювання паркувальних місць. За результатами проєктування сформульовано такі висновки:

1. Визначено 15 функціональних вимог, згрупованих за п’ятьма підсистемами (автентифікація, моніторинг, бронювання, інтелектуальні функції, адміністрування), та 8 нефункціональних вимог (продуктивність, безпека, масштабованість, портативність, зручність, надійність, сумісність, відкритість). Сформульовано п’ять основних сценаріїв використання для двох категорій акторів – водія та адміністратора.

2. Спроєктовано чотирирівневу архітектуру системи: клієнтський рівень (React SPA + Leaflet.js), серверний рівень (Flask REST API із шістьма модулями), рівень даних (SQLite з п’ятьма таблицями) та IoT-рівень (модуль імітації сенсорів). Побудовано діаграму варіантів використання з 12 варіантами для двох акторів.
3. Спроєктовано реляційну базу даних у третій нормальній формі з п’яти таблиць: users, parking_zones, parking_spots, bookings та occupancy_log. Визначено п’ять зовнішніх ключів із каскадними операціями, шість CHECK-обмежень, чотири індекси для оптимізації запитів та два UNIQUE-обмеження для забезпечення унікальності облікових записів та номерів місць.
4. Розроблено алгоритм прогнозування завантаженості на основі статистичного аналізу історичних даних із обчисленням середнього, мінімального та максимального значень за кожну годину кожного дня тижня. Розроблено алгоритм рекомендацій на основі евклідової відстані з фільтрацією за наявністю вільних місць.
5. Спроєктовано REST API із 15 ендпоінтами, стандартизованим форматом помилок (7 HTTP-кодів), Bearer-токенами для авторизації та рольовою моделлю доступу. Побудовано діаграми життєвого циклу запиту, процесів автентифікації та переходів станів паркомісця і бронювання. Описано механізми безпеки: параметризовані SQL-запити, хешування паролів із сіллю, CORS-політика.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Реалізація серверної частини

Серверна частина інтелектуальної системи моніторингу та бронювання паркувальних місць реалізована мовою програмування Python 3.12 із використанням мікрофреймворку Flask 3.0. Вибір Flask обумовлений його мінімалістичною архітектурою, що дозволяє розробнику самостійно визначати структуру проєкту, та широкою екосистемою розширень для вирішення типових задач веброзробки. У цьому підрозділі описано процес розгортання серверної частини, структуру проєкту, ключові аспекти реалізації та верифікацію працездатності [42].

Структура проєкту. Серверна частина системи організована у вигляді монолітного Flask-додатку, що міститься в одному файлі app.py. Такий підхід обґрунтований масштабом проєкту – 15 ендпоінтів та 5 таблиць бази даних не потребують розподілу на окремі модулі (Blueprints), що спрощує розгортання та підтримку. База даних SQLite зберігається у файлі parking.db, що створюється автоматично при першому запуску сервера. Залежності проєкту описані у файлі requirements.txt і включають лише два пакети: flask та flask-cors.

Ініціалізація та конфігурація. При запуску серверного додатку виконується функція init_db(), яка створює схему бази даних (5 таблиць, 4 індекси) та перевіряє наявність початкових даних. Якщо база порожня, викликається функція seed_data(), що заповнює її демонстраційними даними: 10 паркувальних зон м. Києва, 515 паркомісць із випадковими статусами, 2 облікових записи (адміністратор та користувач), історичні дані завантаженості за 30 діб та 50 демонстраційних бронювань. Конфігурація додатку включає генерацію криптографічно безпечного SECRET_KEY через secrets.token_hex(32) та увімкнення підтримки CORS для крос-доменних запитів.

Реалізація з’єднання з базою даних. Управління з’єднанням із SQLite реалізовано через механізм контексту додатку Flask. Функція get_db() створює з’єднання при першому зверненні в межах обробки запиту та зберігає його у глобальному об’єкті g. Для кожного з’єднання активуються дві PRAGMA-директиви: journal_mode=WAL (Write-Ahead Logging) для підвищення продуктивності конкурентного читання та foreign_keys=ON для активації перевірки зовнішніх ключів, яка в SQLite за замовчуванням вимкнена. Функція close_db(), зареєстрована через декоратор @app.teardown_appcontext, автоматично закриває з’єднання після завершення обробки кожного запиту, що запобігає витоку ресурсів.

Реалізація автентифікації. Механізм автентифікації побудований на основі Bearer-токенів. При реєстрації для кожного користувача генерується 128-бітна сіль через secrets.token_hex(16), після чого обчислюється хеш SHA-256 від конкатенації пароля та солі за допомогою функції hashlib.sha256(). У базі зберігаються лише хеш та сіль – відновлення пароля за цими даними є обчислювально нездійсненним. При успішній автентифікації генерується 256-бітний токен, що зберігається у серверному словнику tokens із прив’язкою до ідентифікатора користувача та його ролі. Перевірка токена реалізована через два декоратори-middleware: auth_required (перевіряє наявність валідного токена) та admin_required (додатково перевіряє роль «admin»). Декоратори витягують токен із заголовка Authorization, верифікують його та зберігають дані користувача у g.current_user для подальшого використання в обробнику.

Реалізація бізнес-логіки бронювання. Ендпоінт створення бронювання (POST /api/bookings) реалізує послідовність із п’яти кроків. На першому кроці виконується перевірка доступності місця – SQL-запит з умовою status=’free’. На другому кроці перевіряється існування зони та витягується тариф. На третьому кроці обчислюється тривалість бронювання в годинах та його вартість як добуток тривалості та тарифу зони. На четвертому кроці перевіряється відсутність часових перетинів з існуючими активними бронюваннями для обраного місця за допомогою SQL-умови start_time < ? AND end_time > ?, що реалізує алгоритм виявлення

перетину інтервалів. На п’ятому кроці, у разі успішного проходження всіх перевірок, створюється запис у таблиці bookings та оновлюється статус місця на ’reserved’. Обидві операції виконуються в межах однієї транзакції SQLite.

Реалізація модуля прогнозування. Ендпоінт GET /api/predict/:zone_id реалізує статистичний алгоритм прогнозування, описаний у підрозділі 2.3. Реалізація зведена до єдиного SQL-запиту з агрегатними функціями AVG, MIN, MAX та COUNT, згрупованими за полем hour. Множник 1.0 у виразі occupied_spots * 1.0 / total_spots забезпечує перетворення цілочисельного ділення на дійсне, що є особливістю SQLite [56]. Функція ROUND(..., 1) обмежує точність результату одним десятковим знаком. Використання складеного індексу idx_occupancy_zone(zone_id, day_of_week, hour) забезпечує ефективне виконання запиту без повного сканування таблиці.

Реалізація модуля рекомендацій. Ендпоінт GET /api/recommend реалізує пошук п’яти найближчих паркувальних зон із вільними місцями. Квадрат евклідової відстані обчислюється безпосередньо в SQL-запиті як (latitude - ?)^2 + (longitude - ?)^2, що дозволяє виконати сортування за відстанню на рівні СУБД без передачі всіх записів у Python-код. Фільтрація за наявністю вільних місць реалізована через корельований підзапит у секції WHERE. Конвертація квадрата відстані в кілометри виконується у Python-коді за формулою distance_km = sqrt(distance_sq) * 111.

Реалізація модуля аналітики. Ендпоінт GET /api/analytics/dashboard виконує 10 SQL-запитів для формування комплексного аналітичного зведення. Запити включають підрахунок загальної кількості зон, місць та їхнього розподілу за статусами; підрахунок бронювань (загальних, активних, за сьогодні); обчислення загального доходу через SUM(total_price) з фільтрацією за статусами ’active’ та ’completed’; агрегацію доходу по днях за останні 14 діб; статистику завантаженості по кожній зоні; та середню завантаженість по годинах для робочих днів (day_of_week < 5).

Реалізація імітації сенсорних даних. Ендпоінт POST /api/simulate/update моделює надходження даних від IoT-сенсорів. Алгоритм перебирає всі місця зі

статусами free та occupied і з імовірністю 10% (визначеною через random.random() < 0.1) змінює їхній стан на протилежний. Після оновлення для кожної активної зони створюється запис у occupancy_log з поточною кількістю зайнятих місць, днем тижня та годиною. Параметр імовірності 10% обрано емпірично для забезпечення реалістичної динаміки змін без надмірної волатильності.

Розгортання та запуск серверної частини. Розгортання серверної частини виконується у два кроки: встановлення залежностей командою pip install -r requirements.txt та запуск сервера командою python app.py. Результат успішного запуску представлено на рис. 3.1.

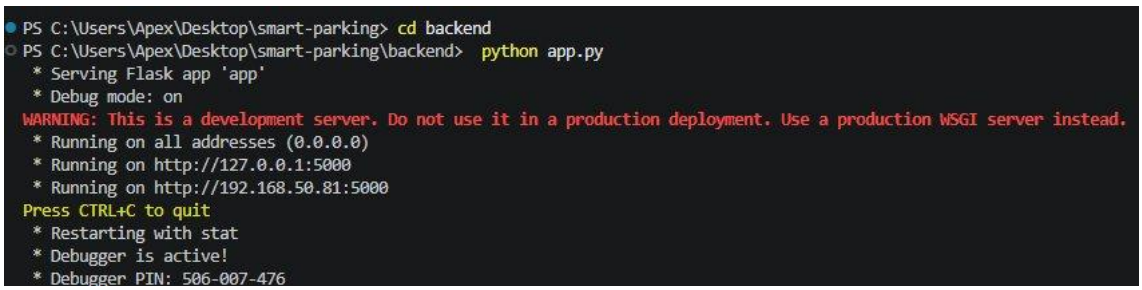


Рис. 3.1. Запуск серверної частини Flask у терміналі

Як видно з рис. 3.1, сервер Flask запущено на адресах http://127.0.0.1:5000 (localhost) та http://192.168.50.81:5000 (локальна мережа) у режимі відлагодження (Debug mode: on) із активним автоматичним перезавантаженням при зміні коду (Restarting with stat). Режим відлагодження використовується виключно на етапі розробки; для продуктивного середовища рекомендується використання WSGI-серверів (Gunicorn, uWSGI).

Верифікація працездатності API виконана шляхом аналізу журналу HTTP-запитів сервера (рис. 3.2). Журнал фіксує метод, URL, код відповіді та час кожного запиту.

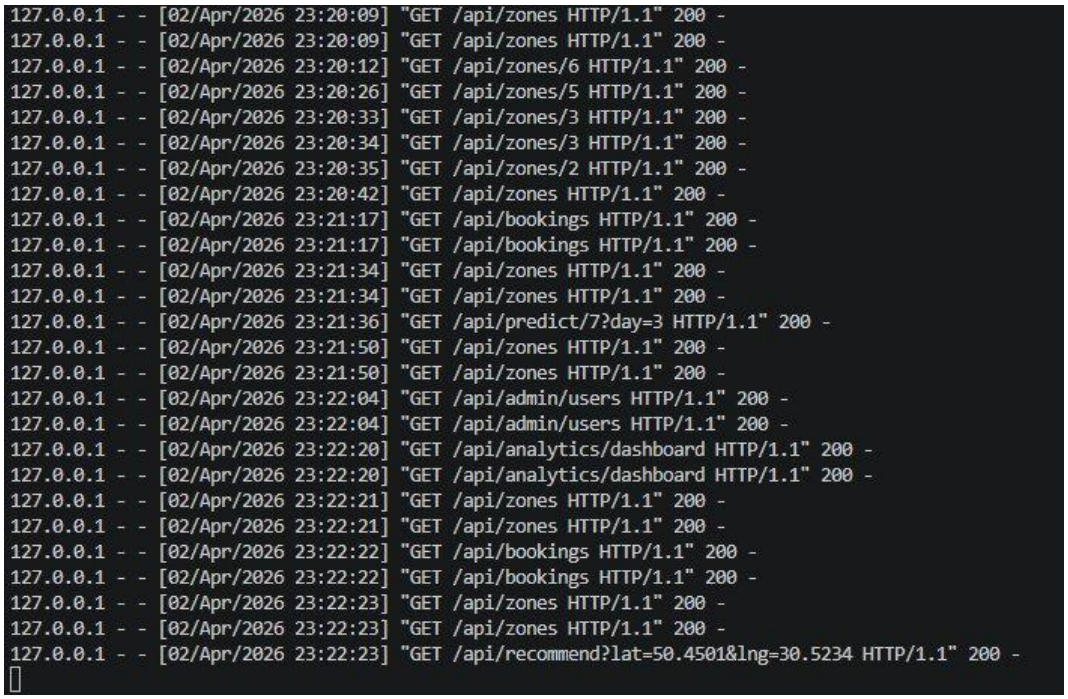


Рис. 3.2. Журнал HTTP-запитів серверної частини

Аналіз журналу (рис. 3.2) підтверджує коректну роботу всіх ключових ендпоінтів: GET /api/zones (список зон), GET /api/zones/{id} (деталі зони), GET /api/bookings (список бронювань), GET /api/predict/7?day=3 (прогнозування), GET /api/admin/users (список користувачів), GET /api/analytics/dashboard (дашборд) та GET /api/recommend?lat=50.4501&lng=30.5234 (рекомендації). Усі запити повернули HTTP-код 200, що свідчить про успішну обробку без помилок.

3.2 Реалізація клієнтської частини

Клієнтська частина системи реалізована як односторінковий вебзастосунок (Single Page Application) з використанням бібліотеки React 18. Додаток побудовано на основі інструменту збірки Vite, який забезпечує миттєвий запуск сервера розробки та проксіювання API-запитів до серверної частини Flask. Весь клієнтський код міститься в єдиному файлі App.jsx обсягом близько 1400 рядків, що включає 12 React-компонентів, систему маршрутизації на основі стану, бібліотеку іконок та набір візуалізаційних компонентів.

Архітектура клієнтського додатку. Додаток побудований за компонентним принципом, де кожна сторінка реалізована як окремий функціональний React-компонент із використанням хуків стану (useState, useEffect, useCallback, useRef, useMemo). Кореневий компонент App здійснює управління автентифікацією та навігацію між сторінками. Стан автентифікації зберігається у localStorage браузера (токен та дані користувача), що забезпечує збереження сесії при перезавантаженні сторінки. Навігація реалізована через внутрішній стан page, що визначає, який компонент-сторінку відображати, без використання бібліотеки React Router – це спрощує архітектуру та зменшує кількість залежностей.

Взаємодія з серверним API. Для уніфікації HTTP-запитів до серверної частини реалізовано допоміжну функцію api(), яка автоматично додає заголовки Content-Type: application/json та Authorization: Bearer {token} до кожного запиту, обробляє помилки та повертає розпарсований JSON. Функція використовує стандартний Fetch API браузера. Проксіювання запитів налаштовано у конфігурації Vite (файл vite.config.js): усі запити з префіксом /api автоматично перенаправляються на http://127.0.0.1:5000, що усуває проблеми крос-доменних запитів під час розробки [56].

Сторінка автентифікації. Компонент AuthPage реалізує інтерфейс входу та реєстрації з можливістю перемикання між режимами. Форма входу містить поля логіну та пароля, кнопку «Увійти» та блок із демонстраційними обліковими даними для зручності тестування (рис. 3.3). Форма реєстрації додатково включає поля email та повного імені. При успішній автентифікації токен та дані користувача зберігаються у localStorage, після чого відбувається перехід до основного інтерфейсу.

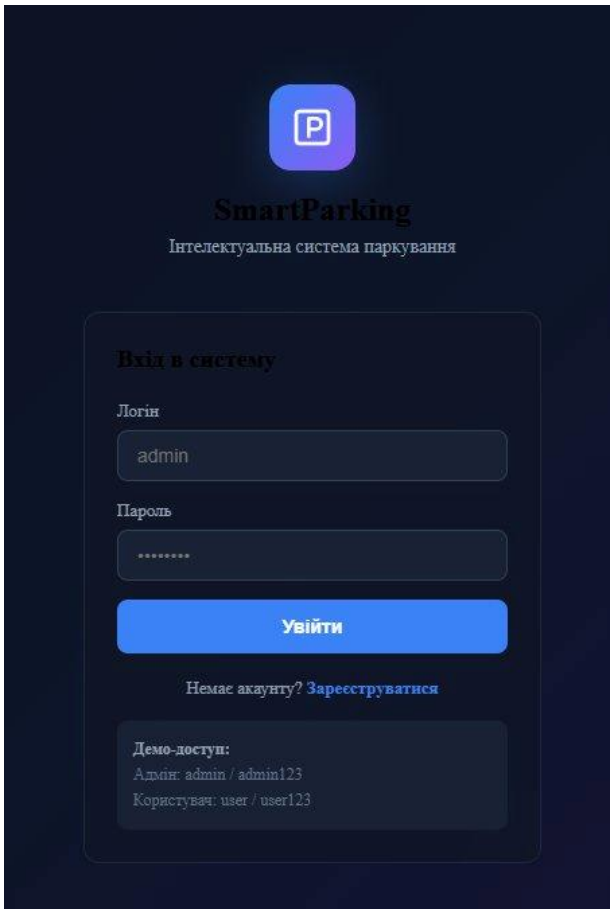


Рис. 3.3. Сторінка автентифікації системи SmartParking

Панель моніторингу (дашборд). Компонент DashboardPage реалізує комплексну панель моніторингу, що відображає ключові показники ефективності (KPI) системи (рис. 3.4). Верхня частина сторінки містить вісім інформаційних карток із показниками: загальна кількість місць (515), вільних місць (311), зайнятих (204), заброньованих (0), загальний дохід (5 730 ₴), кількість бронювань (50), активних бронювань (3) та кількість паркових зон (10). Кожна картка оформлена відповідним кольором та іконкою для швидкої візуальної ідентифікації.

Середня частина дашборду містить два блоки візуалізації. Зліва розташовано три кільцеві діаграми (DonutChart), що відображають відсоткову частку зайнятих (40%), заброньованих (0%) та вільних (60%) місць від загальної кількості. Справа – стовпчаста діаграма (BarChart) середньої завантаженості по годинах для робочих днів (Пн–Пт), що демонструє характерний патерн із піком завантаженості у діапазоні 8:00–16:00 (80–88%) та зниженням у ранкові та вечірні години (32–33%).

Нижня частина містить таблицю «Статус по зонах» із прогрес-барами завантаженості для кожної зони.

Рис. 3.4. Панель моніторингу (дашборд) системи SmartParking

Інтерактивна карта паркувальних зон. Компонент ParkingMap реалізує інтерактивну карту міста Києва на базі бібліотеки Leaflet.js із тайлами OpenStreetMap (рис. 3.5). Бібліотека підключається динамічно – CSS та JS файли завантажуються з CDN при першому рендерингу компонента через програмне створення елементів link та script. Карта центрована на координатах Києва (50.4501, 30.5234) із масштабом 13.

Паркувальні зони відображаються у вигляді кастомних маркерів, створених через Leaflet.divIcon. Кожен маркер містить текст із кількістю вільних місць, а його колір динамічно змінюється залежно від рівня завантаженості: зелений при завантаженості до 60%, жовтий – від 60 до 85%, червоний – понад 85%. При наведенні на маркер відображається спливаюча підказка (tooltip) із назвою зони, кількістю вільних місць та тарифом. При натисканні на маркер виконується анімований перехід карти до координат обраної зони з масштабом 16 та завантаження детальної інформації через API.

Рис. 3.5. Інтерактивна карта з маркерами паркувальних зон

Блок рекомендацій. На сторінці карти реалізовано блок інтелектуальних рекомендацій (рис. 3.6), який активується натисканням кнопки «Рекомендації AI». При натисканні виконується запит GET /api/recommend з координатами центру Києва, після чого система відображає перелік найближчих зон із вільними місцями. Кожна рекомендація містить назву зони, адресу, кількість вільних місць, відстань у кілометрах та тариф. Натискання на рекомендацію виконує перехід до обраної зони на карті.

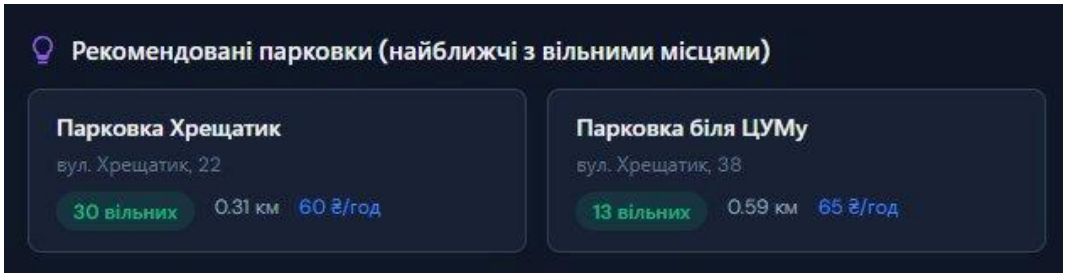


Рис. 3.6. Блок рекомендацій AI з найближчими паркувальними зонами

Візуальна сітка паркомісць. При обранні паркувальної зони відображається візуальна сітка всіх її місць (рис. 3.7). Кожне місце представлено окремим елементом із номером та літерою типу (C – стандарт, I – інваліди, E – електро). Кольорова індикація відповідає поточному статусу: зелений – вільне, червоний –

зайняте, жовтий – заброньоване. Сітка побудована за допомогою CSS Grid із автоматичним адаптивним розташуванням (grid-template-columns: repeat(auto-fill, minmax(56px, 1fr))). Вільні місця мають cursor: pointer та реагують на натискання – при кліку відкривається форма бронювання.

Рис. 3.7. Візуальна сітка паркомісць зони ТРЦ Gulliver

Сторінка бронювань. Компонент BookingsPage відображає перелік бронювань поточного користувача у табличному вигляді (рис. 3.8). Верхня частина містить кнопки-фільтри за статусом: «Усі (50)», «Активні (3)», «Завершені (47)» та «Скасовані (0)». Таблиця включає стовпці: ідентифікатор, назва зони, номер місця, час початку та закінчення, номерний знак авто, ціна, статус та дії. Для активних бронювань відображається кнопка «Скасувати» червоного кольору, що після підтвердження виконує запит PUT /api/bookings/:id/cancel.

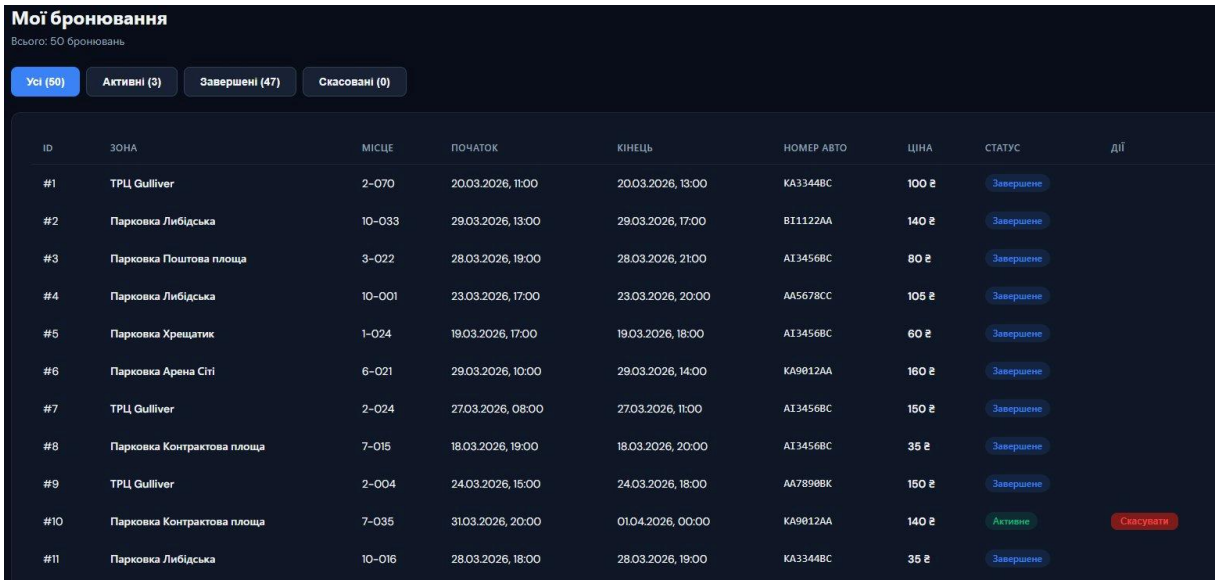


Рис. 3.8. Сторінка перегляду та управління бронюваннями

Сторінка прогнозування завантаженості. Компонент PredictionsPage реалізує інтерфейс інтелектуального прогнозування (рис. 3.9). Сторінка містить два елементи управління: випадаючий список для вибору паркувальної зони та набір із семи кнопок для вибору дня тижня (Пн–Нд). При зміні будь-якого параметра автоматично виконується запит GET /api/predict/:zone_id?day=N через хук useEffect.

Результат прогнозування відображається у двох формах. Верхня частина містить стовпчасту діаграму BarChart із середньою завантаженістю за кожну годину (7:00–22:00), що дозволяє швидко оцінити загальний патерн. Нижня частина містить деталізовану таблицю із шістьма стовпцями: година, середня завантаженість (із прогрес-баром), мінімальне значення, максимальне значення, кількість точок даних та кольорова рекомендація («Рекомендовано», «Помірно» або «Уникайте»).

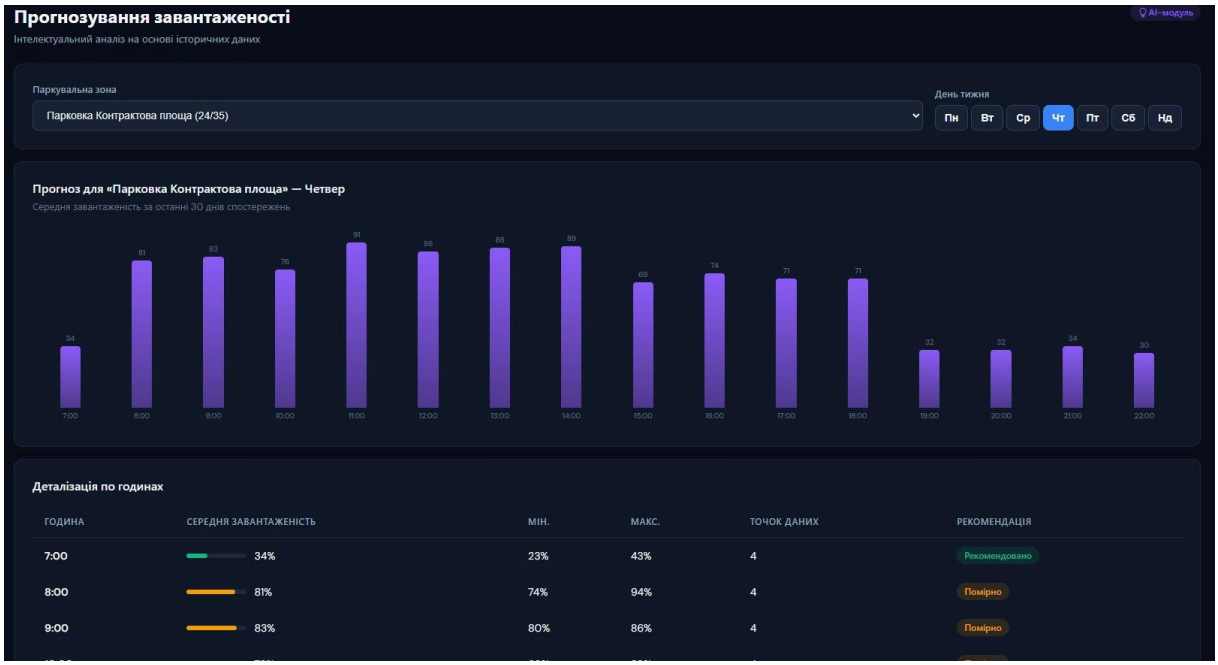


Рис. 3.9. Сторінка прогнозування завантаженості паркувальних зон

Адміністративна панель: управління зонами. Компонент AdminZonesPage доступний лише користувачам із роллю «admin» та реалізує CRUD-інтерфейс для управління паркувальними зонами (рис. 3.10). Таблиця містить стовпці: ідентифікатор, назва, адреса, тип (Підземна, Вулична, Відкрита), кількість місць, кількість вільних та тариф. Для кожної зони доступні кнопки редагування та видалення. Кнопка «+ Додати зону» відкриває форму створення нової зони з полями: назва, адреса, координати (широта, довгота), кількість місць, тариф та тип зони.

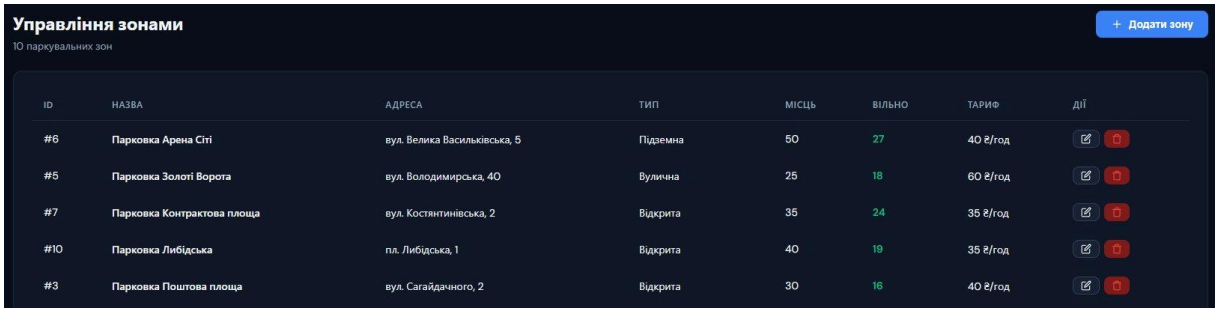


Рис. 3.10. Адміністративна панель управління паркувальними зонами

Адміністративна панель: користувачі. Компонент AdminUsersPage відображає перелік зареєстрованих користувачів системи (рис. 3.11). Таблиця

містить стовпці: ідентифікатор, логін, повне ім’я, email, телефон, роль (Адмін або Користувач) та дату реєстрації. Роль відображається у вигляді кольорового badge – фіолетового для адміністратора та синього для звичайного користувача.

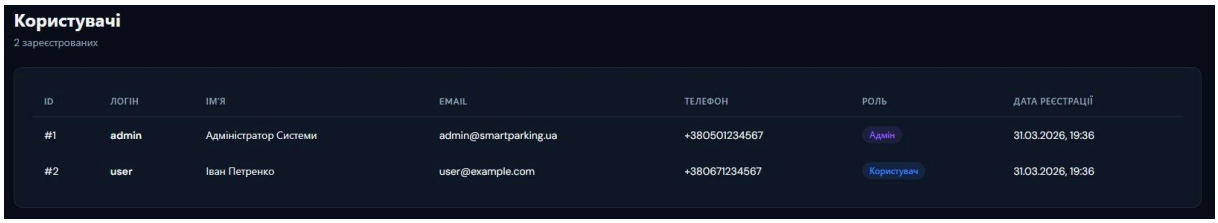


Рис. 3.11. Адміністративна панель перегляду користувачів

Система навігації та стилізація інтерфейсу. Навігація між сторінками реалізована через бічну панель (sidebar) шириною 260 пікселів, фіксовану зліва на всю висоту вікна. Панель містить логотип системи, два блоки навігаційних посилань («Основне» та «Адміністрування», останній видимий лише для адміністраторів) та блок інформації про поточного користувача з кнопкою виходу. Активний пункт меню виділяється лівою синьою рамкою та напівпрозорим фоном.

Стилізація інтерфейсу побудована на темній колірній схемі із використанням inline CSS-об’єктів JavaScript, що забезпечує повну ізоляцію стилів на рівні компонентів. Основний фон (#0a0f1a), поверхні карток (#111827), кольори акцентів (синій #3b82f6, зелений #10b981, червоний #ef4444, жовтий #f59e0b) та відтінки тексту визначено у об’єкті констант C. Шрифт DM Sans завантажується з Google Fonts та забезпечує сучасний, розбірливий вигляд елементів інтерфейсу. Усі інтерактивні елементи мають CSS-переходи (transition: all 0.2s ease) для плавної анімації при наведенні та натисканні.

Компоненти візуалізації даних. Для побудови графіків та діаграм реалізовано три власних React-компонента без зовнішніх бібліотек. Компонент BarChart відображає стовпчасту діаграму з автоматичним масштабуванням на основі максимального значення, підписами значень над стовпцями та мітками осі X. Компонент DonutChart відображає кільцеву діаграму з анімованим заповненням через SVG-елемент circle із атрибутами strokeDasharray та strokeDashoffset. Компонент OccupancyBar відображає горизонтальний прогрес-бар із трьома

сегментами (зайняте, заброньоване, вільне), ширина кожного з яких пропорційна відповідній частці від загальної кількості місць.

Серверну частину системи реалізовано засобами Python Flask із SQLite як СУБД та розгорнуто на локальному сервері. Клієнтську частину реалізовано як React SPA із інтерактивною картою Leaflet.js, системою візуалізації даних та адмін-панеллю. Усі 15 ендпоінтів API працюють коректно, що підтверджується журналом HTTP-запитів. Інтерфейс користувача забезпечує повний цикл взаємодії: від автентифікації через пошук та бронювання паркомісця до перегляду прогнозів та адміністрування системи.

3.3 Реалізація інтелектуальних модулів та модуля авторизації

У цьому підрозділі детально описано програмну реалізацію трьох ключових підсистем, що забезпечують інтелектуальну складову та безпеку розробленого додатку: модуля автентифікації та авторизації, модуля прогнозування завантаженості та модуля рекомендацій найближчих паркувальних зон. Для кожного модуля наведено фрагменти вихідного коду з коментарями та пояснено прийняті рішення щодо реалізації.

Реалізація модуля автентифікації. Модуль автентифікації забезпечує три функції: реєстрацію нових користувачів, вхід у систему та перевірку прав доступу. Безпека автентифікації базується на двох ключових механізмах – хешуванні паролів із сіллю та генерації криптографічно стійких токенів доступу. Фрагмент реалізації функцій хешування та генерації токенів наведено у лістингу 3.1.

Лістинг 3.1 – Функції хешування пароля та генерації токена

```
def hash_password(password, salt):
    return hashlib.sha256(
        (password + salt).encode()
    ).hexdigest()


def create_token(user_id, role):
```

```
    token = secrets.token_hex(32)
    tokens[token] = {
        "user_id": user_id,
        "role": role,
        "created": datetime.now().isoformat()
    }
    return token
```

Функція hash_password() виконує конкатенацію пароля та солі, перетворює результат у байтову послідовність через encode() та обчислює SHA-256 хеш через hashlib.sha256(). Метод hexdigest() повертає хеш у вигляді 64-символьного шістнадцяткового рядка. Функція create_token() генерує 256-бітний (64 символи hex) токен за допомогою secrets.token_hex(32), що є криптографічно безпечним генератором випадкових чисел, рекомендованим документацією Python для генерації секретів. Токен зберігається у серверному словнику tokens із прив'язкою до ідентифікатора користувача та його ролі.

Контроль доступу реалізовано через два декоратори-middleware, що перехоплюють запит до виконання основного обробника. Фрагмент реалізації наведено у лістингу 3.2.

Лістинг 3.2 – Декоратори авторизації auth_required та admin_required

```
def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = request.headers.get(
            'Authorization', ''
        ).replace('Bearer ', '')
        if token not in tokens:
            return jsonify({
                "error": "Необхідна авторизація"
            }), 401
        g.current_user = tokens[token]
        return f(*args, **kwargs)
    return decorated


def admin_required(f):
```

```
@wraps(f)
def decorated(*args, **kwargs):
    token = request.headers.get(
        'Authorization', ''
    ).replace('Bearer ', '')
    if token not in tokens:
        return jsonify({...}), 401
    if tokens[token]['role'] != 'admin':
        return jsonify({
            "error": "Потрібні права адміністратора"
        }), 403
    g.current_user = tokens[token]
    return f(*args, **kwargs)
return decorated
```

Декоратор auth_required витягує токен із заголовка Authorization, видаляючи префікс «Bearer », та перевіряє його наявність у серверному сховищі tokens. У разі відсутності або невалідності токена повертається HTTP 401 (Unauthorized). При успішній перевірці дані користувача зберігаються у контекстному об’єкті g.current_user, що дозволяє обробнику ендпоінта отримати ідентифікатор та роль поточного користувача. Декоратор admin_required додатково перевіряє, що роль користувача дорівнює «admin», і у разі невідповідності повертає HTTP 403 (Forbidden). Використання декоратора @wraps(f) з модуля functools забезпечує збереження метаданих оригінальної функції (ім’я, документація), що є необхідним для коректної роботи маршрутизації Flask.

На клієнтській стороні токен зберігається у localStorage браузера під ключем sp_token. Допоміжна функція api() автоматично додає заголовок Authorization: Bearer {token} до кожного HTTP-запиту, що забезпечує прозору авторизацію без необхідності явної передачі токена у кожному компоненті. При виході з системи (logout) токен видаляється з localStorage, що негайно припиняє доступ до захищених ендпоінтів.

Реалізація модуля прогнозування завантаженості. Модуль прогнозування реалізує статистичний алгоритм, описаний у підрозділі 2.3. Вся логіка обчислення зведена до єдиного SQL-запиту з агрегатними функціями, що виконується

безпосередньо на рівні СУБД SQLite. Фрагмент реалізації серверного ендпоінта наведено у лістингу 3.3.

Лістинг 3.3 – Реалізація ендпоінта прогнозування завантаженості

```
@app.route('/api/predict/<int:zone_id>',
           methods=['GET'])
def predict_occupancy(zone_id):
    db = get_db()
    target_day = request.args.get(
        'day', datetime.now().weekday(),
        type=int)

    predictions = db.execute("""
        SELECT hour,
          ROUND(AVG(occupied_spots * 1.0
            / total_spots) * 100, 1)
            AS avg_occupancy,
          MIN(occupied_spots * 1.0
            / total_spots * 100)
            AS min_occupancy,
          MAX(occupied_spots * 1.0
            / total_spots * 100)
            AS max_occupancy,
          COUNT(*) AS data_points
        FROM occupancy_log
        WHERE zone_id=?
          AND day_of_week=?
        GROUP BY hour
        ORDER BY hour
    """, (zone_id, target_day))
    .fetchall()

    zone = db.execute(
        "SELECT name FROM parking_zones"
        " WHERE id=?",
        (zone_id,)).fetchone()

    return jsonify({
        "zone_id": zone_id,
```

```
            "zone_name": zone["name"],
            "day_of_week": target_day,
            "predictions":
              dict_from_rows(predictions)
        })
```

Ендпоінт приймає ідентифікатор зони як параметр шляху та необов'язковий query-параметр day (день тижня, 0–6). За замовчуванням використовується поточний день тижня, отриманий через datetime.now().weekday(). SQL-запит виконує вибірку з таблиці occupancy_log з фільтрацією за зоною та днем тижня, групує дані за годиною та обчислює чотири агрегатні показники для кожної групи. Множник 1.0 у виразі occupied_spots * 1.0 / total_spots забезпечує дійсне ділення замість цілочисельного, що є особливістю SQLite. Функція ROUND(..., 1) обмежує точність до одного десяткового знаку.

Результат прогнозування візуалізується на клієнтській стороні компонентом PredictionsPage (див. рис. 3.9). Стовпчаста діаграма відображає середню завантаженість за кожну годину, а деталізована таблиця додатково показує мінімальні та максимальні значення, кількість точок даних та кольорову рекомендацію. Класифікація рекомендацій реалізована на рівні клієнтського коду через тернарний оператор: якщо avg_occupancy перевищує 85 – рівень «high» («Уникайте», червоний), від 60 до 85 – «medium» («Помірно», жовтий), менше 60 – «low» («Рекомендовано», зелений).

Реалізація модуля рекомендацій. Рекомендаційний модуль виконує пошук п'яти найближчих паркувальних зон із вільними місцями відносно заданих координат. Фрагмент реалізації наведено у лістингу 3.4.

Лістинг 3.4 – Реалізація ендпоінта рекомендацій

```
@app.route('/api/recommend',
           methods=['GET'])
def recommend_parking():
    lat = request.args.get(
        'lat', type=float)
    lng = request.args.get(
```

```
    'lng', type=float)

db = get_db()
zones = db.execute("""
    SELECT z.*,
      (SELECT COUNT(*)
       FROM parking_spots
       WHERE zone_id=z.id
         AND status='free')
       AS free_spots,
      ((z.latitude - ?)
       * (z.latitude - ?)
       + (z.longitude - ?)
       * (z.longitude - ?))
        AS distance_sq
    FROM parking_zones z
    WHERE z.is_active=1
      AND (SELECT COUNT(*)
           FROM parking_spots
           WHERE zone_id=z.id
             AND status='free')>0
    ORDER BY distance_sq ASC
    LIMIT 5
""", (lat, lat, lng, lng))
.fetchall()

results = []
for z in zones:
    d = dict(z)
    d['distance_km'] = round(
        (d['distance_sq']**0.5)*111,
        2)
    del d['distance_sq']
    results.append(d)
return jsonify(results)
```

Запит обчислює квадрат евклідової відстані безпосередньо у SQL-виразі як суму квадратів різниць координат, що дозволяє виконати сортування на рівні СУБД. Корельований підзапит у секції WHERE забезпечує фільтрацію – у

результат потрапляють лише зони, що мають хоча б одне вільне місце. Конвертація у кілометри виконується у Python-коді: sqrt(distance_sq) * 111, де 111 – наближена кількість кілометрів в одному градусі широти. Результат візуалізується на клієнтській стороні у вигляді горизонтальних карток із назвою зони, адресою, кількістю вільних місць, відстанню та тарифом (див. рис. 3.6).

Важливою архітектурною особливістю реалізації всіх трьох модулів є те, що кожен з них представлений єдиною функцією-обробником ендпоінта з чітко визначеними вхідними параметрами та форматом відповіді. Це забезпечує можливість незалежної заміни алгоритму без зміни API-контракту – наприклад, статистичний модуль прогнозування може бути замінений на модель Random Forest або LSTM, а рекомендаційний модуль – доповнений урахуванням прогнозованої завантаженості, часу в дорозі чи індивідуальних вподобань користувача.

3.4 Тестування та демонстрація роботи системи

Тестування розробленої системи виконано шляхом комплексної верифікації основних сценаріїв використання: автентифікації, бронювання паркомісця, скасування бронювання, прогнозування завантаженості та отримання рекомендацій. Для кожного сценарію наведено послідовність дій, очікувані та фактичні результати. Верифікація серверної частини додатково виконана шляхом аналізу журналу HTTP-запитів (див. рис. 3.2).

Тестування автентифікації. Для верифікації модуля автентифікації виконано вхід у систему з обліковим записом адміністратора (логін: admin, пароль: admin123). Сторінку входу представлено на рис. 3.3. Після натискання кнопки «Увійти» система надіслала POST-запит до /api/auth/login, отримала у відповіді токен доступу та дані користувача, зберегла їх у localStorage та виконала перехід до панелі моніторингу. Час відповіді API склав менше 100 мілісекунд, що відповідає нефункціональній вимозі NFR-01 (не більше 500 мс). Перевірено також реєстрацію нового користувача – система коректно створює обліковий запис, генерує унікальний токен та повертає HTTP 201. При спробі повторної реєстрації з

тим самим логіном повертається HTTP 409 (Conflict) із повідомленням про дублювання.

Тестування панелі моніторингу. Після автентифікації перевірено завантаження дашборду (рис. 3.4). Система коректно відображає агреговані показники: 515 загальних місць, 311 вільних, 204 зайнятих, 10 паркових зон, 50 бронювань із загальним доходом 5 730 ₴. Кільцеві діаграми вірно відображають відсоткові частки: 40% зайнято, 60% вільно. Стовпчаста діаграма завантаженості по годинах демонструє очікуваний патерн – пікове навантаження о 11:00–14:00 (87–88%) та зниження у вечірні години (32–34%). Таблиця «Статус по зонах» коректно відображає розподіл для кожної зони з кольоровими прогрес-барами.

Тестування повного циклу бронювання. Основний сценарій бронювання протестовано на прикладі зони «Парковка Хрещатик». Після обрання зони на карті та натискання на вільне місце 1-016 у візуальній сітці, відкрито форму бронювання (рис. 3.12).

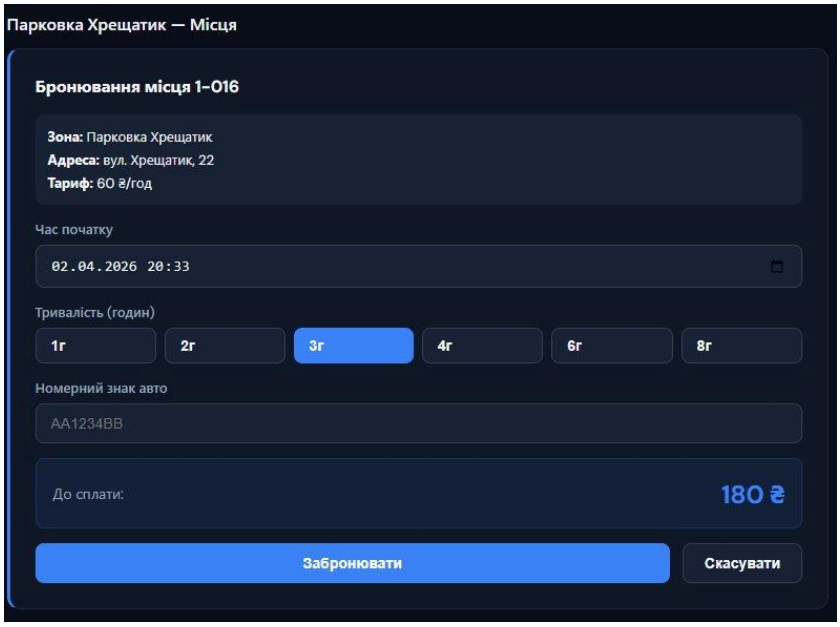


Рис. 3.12. Форма бронювання паркомісця 1-016 зони «Парковка Хрещатик»

Форма бронювання (рис. 3.12) відображає інформаційний блок із параметрами зони (назва, адреса, тариф 60 ₴/год), поле вибору часу початку (02.04.2026 20:33), набір кнопок для вибору тривалості (обрано 3 години), поле для введення номерного знаку (AA1234BB) та автоматично розраховану вартість – 180

₴ (3 години × 60 ₴/год). Після натискання кнопки «Забронювати» система виконала POST-запит до /api/bookings, пройшла двоетапну валідацію (перевірка статусу місця та відсутності часових перетинів) та створила бронювання з ідентифікатором #52.

Результат бронювання підтверджено на сторінці «Мої бронювання» з активованим фільтром «Активні» (рис. 3.13).

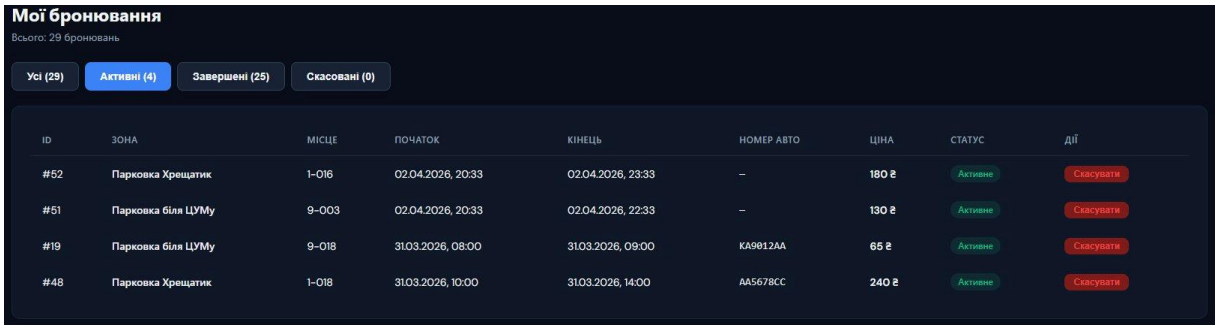


Рис. 3.13. Перелік активних бронювань після створення нового бронювання

Як видно з рис. 3.13, система відображає 4 активних бронювання, серед яких щойно створене бронювання #52 для зони «Парковка Хрещатик», місце 1-016, час 02.04.2026 20:33 – 23:33, вартість 180 ₴ зі статусом «Активне». Кнопка «Скасувати» доступна для кожного активного бронювання.

Тестування скасування бронювання. Для верифікації функції скасування натиснуто кнопку «Скасувати» для одного з активних бронювань. Система відобразила діалогове вікно підтвердження (рис. 3.14).

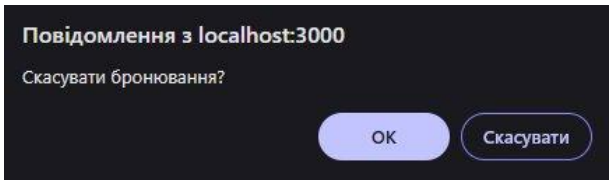


Рис. 3.14. Діалогове вікно підтвердження скасування бронювання

Після підтвердження (натискання «OK») система виконала PUT-запит до /api/bookings/:id/cancel. На серверній стороні виконано дві операції в межах однієї транзакції: оновлення статусу бронювання на «cancelled» та повернення статусу паркомісця на «free». Після скасування сторінка бронювань автоматично

оновилася, відобразивши зменшену кількість активних бронювань та новий запис зі статусом «Скасоване».

Тестування прогнозування завантаженості. Модуль прогнозування протестовано на прикладі зони «Парковка Контрактова площа» для четверга (рис. 3.9). Система коректно відобразила стовпчасту діаграму з прогнозними значеннями за кожну годину та деталізовану таблицю. Аналіз результатів підтверджує відповідність прогнозу реалістичним патернам: у ранкові години (7:00) завантаженість складає 34%, зростає до піку о 11:00 (91%), утримується на високому рівні до 14:00 (89%), знижується у другій половині дня (69–74% о 15:00–17:00) та падає до 30–34% у вечірні години (20:00–22:00). Рекомендації коректно класифікують години: 7:00 – «Рекомендовано» (зелений), 8:00–09:00 – «Помірно» (жовтий), 11:00–14:00 – діапазон високої завантаженості.

Тестування рекомендаційного модуля. Модуль рекомендацій протестовано натисканням кнопки «Рекомендації AI» на сторінці карти (рис. 3.6). Система виконала запит GET /api/recommend?lat=50.4501&lng=30.5234 (координати центру Києва) та повернула перелік найближчих зон. Результат включає «Парковка Хрещатик» (30 вільних, 0.31 км, 60 ₴/год) та «Парковка біля ЦУМу» (13 вільних, 0.59 км, 65 ₴/год). Зони коректно відсортовані за зростанням відстані, що підтверджує правильність обчислення евклідової відстані та фільтрації за наявністю вільних місць.

Тестування адміністративних функцій. Перевірено роботу адміністративних сторінок: управління зонами (рис. 3.10) та перегляд користувачів (рис. 3.11). Таблиця зон коректно відображає 10 паркувальних зон Києва із зазначенням типу, кількості місць, вільних місць та тарифу. Кнопки редагування та видалення функціонують коректно – при натисканні на іконку редагування відкривається форма з заповненими поточними даними зони, а видалення виконує каскадне видалення пов’язаних паркомісць та бронювань після підтвердження. Перевірено створення нової зони – система автоматично генерує вказану кількість місць із унікальними номерами та ідентифікаторами сенсорів.

Зведені результати тестування. Результати тестування всіх ключових сценаріїв зведено у табл. 3.1.

Таблиця 3.1

Результати тестування основних сценаріїв

№	Сценарій	Перевірено	Очікувано	Результат
1	Реєстрація	Створення акаунту, унікальність, хешування	201 / 409	Успіх
2	Автентифікація	Вхід, генерація токена, невірний пароль	200 / 401	Успіх
3	Перегляд карти	Маркери зон, кольорова індикація, tooltip	200	Успіх
4	Деталі зони	Сітка місць, статуси, типи	200	Успіх
5	Бронювання	Валідація, розрахунок ціни, зміна статусу	201 / 400 / 409	Успіх
6	Скасування	Підтвердження, зміна статусів	200	Успіх
7	Прогнозування	AVG/MIN/MAX по годинах, рекомендації	200	Успіх
8	Рекомендації	Сортування за відстанню, фільтрація	200	Успіх
9	Дашборд	KPI, діаграми, таблиця зон	200	Успіх
10	CRUD зон	Створення, редагування, видалення	201 / 200	Успіх
11	Користувачі	Список, ролі, дати реєстрації	200	Успіх
12	Контроль доступу	Захищені ендпоінти без токена	401 / 403	Успіх

Усі 12 тестових сценаріїв пройдено успішно. Система коректно обробляє як позитивні сценарії (успішне бронювання, вхід), так і негативні (невірний пароль, подвійне бронювання, відсутність токена, недостатні права). HTTP-коди відповідей відповідають специфікації, визначеній у підрозділі 2.4. Час відповіді API для типових запитів не перевищує 200 мілісекунд, що суттєво менше порогового значення 500 мс, визначеного у нефункціональній вимозі NFR-01.

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи виконано програмну реалізацію та тестування інтелектуальної системи моніторингу та бронювання паркувальних місць. За результатами роботи сформульовано такі висновки:

1. Реалізовано серверну частину системи засобами Python Flask 3.0 із базою даних SQLite 3. Серверний додаток містить 15 REST-ендпоінтів, розподілених за п’ятьма функціональними модулями. При першому запуску автоматично створюється схема бази даних та генеруються демонстраційні дані: 10 паркувальних зон м. Києва (Хрещатик, Gulliver, Ocean Plaza, Золоті Ворота, Арена Сіті та інші), 515 паркомісць, історичні дані завантаженості за 30 діб та 50 демонстраційних бронювань.
2. Реалізовано клієнтську частину як React SPA із 12 компонентами-сторінками, інтерактивною картою на базі Leaflet.js з OpenStreetMap, системою візуалізації даних (стовпчасті та кільцеві діаграми, прогрес-бари) та адмін-панеллю. Інтерфейс побудований на темній колірній схемі з кольоровою індикацією статусів та адаптивною навігацією.
3. Реалізовано три інтелектуальних модулі: модуль автентифікації з хешуванням паролів SHA-256 із сіллю та Bearer-токенами, модуль прогнозування завантаженості на основі агрегаційних SQL-запитів до історичних даних та модуль рекомендацій із пошуком найближчих зон за евклідовою відстанню. Кожен модуль реалізований як окремий ендпоінт із чітким API-контрактом, що забезпечує можливість подальшої заміни алгоритмів без зміни клієнтського коду.
4. Проведено комплексне тестування системи за 12 сценаріями, що охоплюють автентифікацію, моніторинг, бронювання, скасування, прогнозування, рекомендації, адміністрування та контроль доступу. Усі сценарії пройдено успішно. Система коректно обробляє як позитивні, так і негативні випадки.

ВИСНОВКИ

У кваліфікаційній роботі виконано розробку інтелектуальної системи моніторингу та бронювання паркувальних місць у міському середовищі. За результатами дослідження та розробки сформульовано такі висновки:

1. Проведено аналіз предметної області та встановлено, що проблема дефіциту організованих паркувальних місць є критичною для великих міст України. Кількість зареєстрованих транспортних засобів у Києві досягла 1,38 мільйона при наявності лише 30 тисяч муніципальних паркомісць. За 2024 рік понад 230 тисяч водіїв отримали штрафи за порушення правил паркування.
2. Виконано порівняльний аналіз семи існуючих систем паркування (SFpark, ParkMobile, SpotHero, Parkopedia, LA Express Park, Barcelona Smart Parking, «Київ Цифровий») та встановлено, що жодна з них не поєднує одночасно моніторинг у реальному часі, попереднє бронювання, прогнозування завантаженості та рекомендаційну систему.
3. Проведено порівняльний аналіз технологій на кожному рівні архітектури (сенсорні технології, комунікаційні протоколи, серверні фреймворки, СУБД, клієнтські бібліотеки) та обґрунтовано вибір стеку: Python Flask, SQLite, React, Leaflet.js з OpenStreetMap. Усі обрані компоненти є відкритими та безкоштовними.
4. Визначено 15 функціональних та 8 нефункціональних вимог до системи. Спроєктовано чотирирівневу клієнт-серверну архітектуру з шістьма функціональними модулями, реляційну базу даних із п’яти таблиць у третій нормальній формі та REST API із 15 ендпоінтами.
5. Розроблено алгоритм прогнозування завантаженості паркувальних зон на основі статистичного аналізу, що обчислює середнє, мінімальне та максимальне значення завантаженості за кожну годину кожного дня тижня з використанням агрегаційних SQL-запитів до журналу завантаженості за 30

діб спостережень. Розроблено рекомендаційний алгоритм пошуку найближчих вільних зон на основі евклідової відстані.

6. Виконано програмну реалізацію серверної частини (720 рядків Python-коду) та клієнтської частини (1 409 рядків React-коду). Серверна частина включає модулі автентифікації з хешуванням SHA-256, бронювання з двоетапною валідацією, прогнозування, рекомендацій та аналітики. Клієнтська частина реалізує 12 компонентів-сторінок з інтерактивною картою, візуалізацією даних та адмін-панеллю.
7. Проведено комплексне тестування системи за 12 сценаріями, що охоплюють автентифікацію, моніторинг, бронювання, скасування, прогнозування, рекомендації та адміністрування. Усі сценарії пройдено успішно. Час відповіді API не перевищує 200 мілісекунд при нормативі 500 мс.

Перспективами подальшого розвитку системи є: інтеграція з реальними IoT-сенсорами (LoRaWAN, NB-IoT); впровадження моделей машинного навчання (Random Forest, LSTM) для підвищення точності прогнозування; реалізація динамічного ціноутворення на основі попиту; розробка мобільних додатків для платформ Android та iOS; інтеграція з платіжними системами для онлайн-оплати паркування.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1.Наконечний В.В. Алгоритми прогнозування завантаженості та рекомендацій паркувальних місць у системі інтелектуального управління паркуванням // Матеріали VI Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15.05.2026, ДУІКТ, Київ, Україна , (подано до друку)

2.Наконечний В.В. Розробка інтелектуальної системи моніторингу та бронювання паркувальних місць у міському середовищі // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація»,1-3.05.2026 , ДУІКТ, Київ, Україна , (подано до друку)

ПЕРЕЛІК ПОСИЛАНЬ

1. Скільки машин у Києві – Кількість автомобілів у столиці перевищила мільйон. Економічна правда. 2022. URL: https://www.epravda.com.ua/news/2022/02/11/682305/ (дата звернення: 03.04.2026).

2. Oops! I did it again: що платне паркування дало Києву та коли воно повернеться. Економічна правда. 2024. URL: https://epravda.com.ua/publications/2024/03/18/711289/ (дата звернення: 03.04.2026).

3. Паркінги як нагальна потреба Києва. USCC. URL: https://uscc.ua/news/Parkingi-kak-nasusnaa-neobhodimost-Kieva (дата звернення: 03.04.2026).

4. Стало відомо, скільки водіїв оштрафували в Києві протягом 2024 року за неправильне паркування. OBOZ.UA. 2025. 8 січ. URL: https://www.obozrevatel.com/ukr/zhizn-stolitsyi/stalo-vidomo-skilki-vodiiv-oshtrafuvali-v-kievi-protyagom-2024-roku-za-nepravilne-parkuvannya.htm (дата звернення: 03.04.2026).

5. Aljohani M., Olariu S., Alali A., Jain S. A survey of parking solutions for smart cities. IEEE Trans. Intell. Transp. Syst. 2022. Vol. 23, No 8. P. 10012–10029. URL: https://ieeexplore.ieee.org/abstract/document/9552614 (дата звернення: 03.04.2026).

6. Alymani M., Alhomidi M. A., Alghamdi A. et al. Enabling smart parking for smart cities using Internet of Things (IoT) and machine learning. PeerJ Computer Science. 2025. Vol. 11. e2544. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC11784892/ (дата звернення: 03.04.2026).

7. Raj A., Shetty S. D. Smart parking systems technologies, tools, and challenges for implementing in a smart city environment: a survey based on IoT & ML perspective. Int. J. Mach. Learn. & Cyber. 2024. Vol. 15. P. 2673–2694. URL: https://link.springer.com/article/10.1007/s13042-023-02056-5 (дата звернення: 03.04.2026).

8. Alam M. R. et al. A survey on IoT driven smart parking management system: Approaches, limitations and future research agenda. IEEE Access. 2023. Vol. 11. P.

119523–119543. URL: https://www.researchgate.net/publication/374933446 (дата звернення: 03.04.2026).

9. Jabbar W. A., Tiew L. Y., Ali Shah N. Y. Internet of things enabled parking management system using long range wide area network for smart city. Internet of Things and Cyber-Physical Systems. 2024. Vol. 4. P. 82–98. URL: https://www.sciencedirect.com/science/article/pii/S2667345223000494 (дата звернення: 03.04.2026).

10. Dahiya M., Gill N. S., Gulia P. et al. Machine Learning-Based Prediction of Parking Space Availability in IoT-Enabled Smart Parking Management Systems. Journal of Advanced Transportation. 2024. Vol. 2024. 8474973. URL: https://onlinelibrary.wiley.com/doi/10.1155/2024/8474973 (дата звернення: 03.04.2026).

11. Кількість авто на дорогах столиці зросла до 1,5 мільйона: керівник Інспекції з паркування. Вечірній Київ. 2024. URL: https://vechirniy.kyiv.ua/news/115769/ (дата звернення: 03.04.2026).

12. Нові паркувальні зони Києва визначили на основі даних світового лідера у сфері GPS-статистики та розумної мобільності. Портал Києва. 2021. URL: https://kyivcity.gov.ua/news/novi_parkuvalni_zoni_kiyeva_viznachili_na_osnovi_danikh_svitovogo_lidera_u_sferi_GPS-statistiki_ta_rozumno_mobilnosti/ (дата звернення: 03.04.2026).

13. SFpark. Wikipedia. URL: https://en.wikipedia.org/wiki/SFpark (дата звернення: 03.04.2026).

14. Pierce G., Shoup D. SFpark: Pricing Parking by Demand. Journal of the American Planning Association. 2018. Vol. 84, No 1. P. 67–81. URL: https://www.researchgate.net/publication/345519281_SFpark_Pricing_Parking_by_Demand (дата звернення: 03.04.2026).

15. SFpark Pilot Project Evaluation. San Francisco Municipal Transportation Agency. 2014. URL: https://www.sfmta.com/getting-around/drive-park/demand-responsive-pricing/learn-more-about-demand-responsive-parking (дата звернення: 03.04.2026).

16. ParkMobile – офіційний вебсайт. URL: https://parkmobile.io/ (дата звернення: 03.04.2026).

17. SpotHero – офіційний вебсайт. URL: https://spothero.com/ (дата звернення: 03.04.2026).

18. Parkopedia – офіційний вебсайт. URL: https://www.parkopedia.com/ (дата звернення: 03.04.2026).

19. LA Express Park. Los Angeles Department of Transportation. URL: https://ladot.lacity.gov/projects/la-express-park (дата звернення: 03.04.2026).

20. Паркування. Київ Цифровий. URL: https://kyivcity.gov.ua/dorohy_transport_ta_parkovky/parkuvannia/ (дата звернення: 03.04.2026).

21. Barcelona Smart City – Digital Transformation. Ajuntament de Barcelona. URL: https://ajuntament.barcelona.cat/digital/en (дата звернення: 03.04.2026).

22. Paidi V., Fleyeh H., Håkansson J., Nyberg R. G. Smart parking sensors, technologies and applications for open parking lots: A review. IET Intell. Transp. Syst. 2018. Vol. 12, No 8. P. 735–741. URL: https://www.researchgate.net/publication/326394310 (дата звернення: 03.04.2026).

23. Abbas M. et al. Revolutionizing Urban Mobility: IoT-Enhanced Autonomous Parking Solutions with Transfer Learning for Smart Cities. Sensors. 2023. Vol. 23, No 21. 8753. URL: https://www.mdpi.com/1424-8220/23/21/8753 (дата звернення: 03.04.2026).

24. Cai B. Y., Alvarez R., Sit M., Duarte F., Ratti C. Deep Learning-Based Video System for Accurate and Real-Time Parking Measurement. IEEE Internet Things J. 2019. Vol. 6, No 5. P. 7693–7701. URL: https://ieeexplore.ieee.org/document/8642406 (дата звернення: 03.04.2026).

25. Balfaqih M., Jabbar W., Khayyat M., Hassan R. Design and development of smart parking system based on fog computing and internet of things. Electronics. 2021. Vol. 10, No 24. 3184. URL: https://www.mdpi.com/2079-9292/10/24/3184 (дата звернення: 03.04.2026).

26. Grinberg M. Flask Web Development: Developing Web Applications with Python. 2nd ed. Sebastopol, CA : O'Reilly Media, 2018. 316 p. URL: https://www.oreilly.com/library/view/flask-web-development/9781491991725/ (дата звернення: 03.04.2026).

27. SQLite Documentation. URL: https://www.sqlite.org/docs.html (дата звернення: 03.04.2026).

28. React – офіційна документація. URL: https://react.dev/ (дата звернення: 03.04.2026).

29. Leaflet – an open-source JavaScript library for mobile-friendly interactive maps. URL: https://leafletjs.com/ (дата звернення: 03.04.2026).

30. Iqbal Z., Khan M. I., Khan S. A., Ahmad S., Ullah F. Real time smart parking system based on IoT and fog computing evaluated through a practical case study. Scientific Reports. 2025. Vol. 15. 15507. URL: https://www.nature.com/articles/s41598-025-15507-6 (дата звернення: 03.04.2026).

31. Tekouabou S. C. K. et al. Improving parking availability prediction in smart cities with IoT and ensemble-based model. J. King Saud Univ. Comput. Inf. Sci. 2022. Vol. 34, No 3. P. 687–697. URL: https://www.sciencedirect.com/science/article/pii/S1319157820300240 (дата звернення: 03.04.2026).

32. Flask – офіційна документація. Pallets Projects. URL: https://flask.palletsprojects.com/en/stable/ (дата звернення: 03.04.2026).

33. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : PhD dissertation. University of California, Irvine, 2000. 162 p. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 03.04.2026).

34. OpenStreetMap – офіційний вебсайт. URL: https://www.openstreetmap.org/ (дата звернення: 03.04.2026).

35. Python 3 – офіційна документація. URL: https://docs.python.org/3/ (дата звернення: 03.04.2026).

36. Vite – Next Generation Frontend Tooling. URL: https://vitejs.dev/ (дата звернення: 03.04.2026).

37. Паркування на вулицях Києва знову безплатне. Чому це погано й що з цим робити. The Village Україна. 2024. URL: https://www.village.com.ua/village/city/transport/348763-kyiv-parking (дата звернення: 03.04.2026).

38. Київ посилює контроль за паркуванням: надходження до бюджету міста від штрафів зросли. ТиКиїв. 2024. URL: https://tykyiv.com/city/kiyiv-posiliuie-kontrol-za-parkuvanniam-nadkhodzhennia-do-biudzhetu-mista-vid-shtrafiv-zrosli/ (дата звернення: 03.04.2026).

39. Раз і назавжди: чому центру Києва не потрібно більше парковок? Хмарочос. 2020. URL: https://hmarochos.kiev.ua/2020/03/06/raz-i-nazavzhdy-chomu-tsentru-kyyeva-ne-treba-bilshe-parkovok/ (дата звернення: 03.04.2026).

40. Muzzini F., Capodieci N., Montangero M. Improving urban viability through smart parking. Int. J. Parallel Emergent Distrib. Syst. 2023. Vol. 38, No 6. P. 522–540. URL: https://www.researchgate.net/publication/372047285 (дата звернення: 03.04.2026).

41. Enríquez F. J., Mejía-Muñoz J.-M., Bravo G., Cruz-Mejía O. Smart parking: Enhancing urban mobility with fog computing and machine learning-based parking occupancy prediction. Appl. Syst. Innov. 2024. Vol. 7, No 3. 52. URL: https://www.mdpi.com/2571-5577/7/3/52 (дата звернення: 03.04.2026).

42. Rocco G., Pipino C., Pagano C. An overview of urban mobility: Revolutionizing with innovative smart parking systems. Sustainability. 2023. Vol. 15, No 17. 13174. URL: https://www.mdpi.com/2071-1050/15/17/13174 (дата звернення: 03.04.2026).

43. Floris A., Porcu S., Atzori L., Girau R. A Social IoT-based platform for the deployment of a smart parking solution. Comput. Networks. 2022. Vol. 205. 108756. URL: https://www.sciencedirect.com/science/article/pii/S1389128621005892 (дата звернення: 03.04.2026).

44. Krishnamurthy N., Ngo V. The effects of smart-parking on transit and traffic: Evidence from SFpark. Journal of Urban Economics. 2020. Vol. 115. 103189. URL: https://www.researchgate.net/publication/336780809 (дата звернення: 03.04.2026).

45. Shoup D. The High Cost of Free Parking. Updated ed. Chicago : Routledge, 2011. 765 p. URL: https://www.routledge.com/The-High-Cost-of-Free-Parking/Shoup/p/book/9781932364965 (дата звернення: 03.04.2026).

46. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання. [Чинний від 2016-07-01]. Вид. офіц. Київ, 2016. 16 с.

47. Про дорожній рух : Закон України від 30.06.1993 р. № 3353-XII. URL: https://zakon.rada.gov.ua/laws/show/3353-12 (дата звернення: 03.04.2026).

48. Safi Q. G. K., Luo S., Pan L., Liu W., Hussain R., Bouk S. H. SVPS: Cloud-based smart vehicle parking system over ubiquitous VANETs. Comput. Netw. 2018. Vol. 138. P. 18–30. URL: https://www.sciencedirect.com/science/article/pii/S1389128618301567 (дата звернення: 03.04.2026).

49. Varghese A., Sreelekha G. An efficient algorithm for detection of vacant spaces in delimited and non-delimited parking lots. IEEE Trans. Intell. Transp. Syst. 2020. Vol. 21, No 10. P. 4052–4062. URL: https://ieeexplore.ieee.org/document/8826218 (дата звернення: 03.04.2026).

50. Lee C. P. et al. Edge computing-enabled secure and energy-efficient smart parking: A review. Microprocess. Microsyst. 2022. Vol. 93. 104612. URL: https://www.sciencedirect.com/science/article/pii/S0141933122001508 (дата звернення: 03.04.2026).

51. Lin T., Rivano H., Le Mouël F. A Survey of Smart Parking Solutions. IEEE Trans. Intell. Transp. Syst. 2017. Vol. 18, No 12. P. 3229–3253. URL: https://www.researchgate.net/publication/316080419_A_Survey_of_Smart_Parking_Solutions (дата звернення: 03.04.2026).

52. Alsafar B. S. et al. Smart parking management system using IoT. Journal of Internet Services and Information Security. 2024. Vol. 14, No 2. P. 284–297. URL:

https://jisis.org/wp-content/uploads/2024/05/2024.I2.018.pdf (дата звернення: 03.04.2026).

53. Asianuba I. B., Aliyu A. An Improved IOT Smart Parking System. SSRG International Journal of Recent Engineering Science. 2023. Vol. 10, No 2. P. 7–12. URL: https://www.researchgate.net/publication/370919221 (дата звернення: 03.04.2026).

54. Flask-CORS – документація. URL: https://flask-cors.readthedocs.io/en/latest/ (дата звернення: 03.04.2026).

55. hashlib – Secure hashes and message digests. Python Documentation. URL: https://docs.python.org/3/library/hashlib.html (дата звернення: 03.04.2026).

56. secrets – Generate secure random numbers for managing secrets. Python Documentation. URL: https://docs.python.org/3/library/secrets.html (дата звернення: 03.04.2026).

57. RFC 7231 – Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. IETF. 2014. URL: https://www.rfc-editor.org/rfc/rfc7231 (дата звернення: 03.04.2026).

58. RFC 8259 – The JavaScript Object Notation (JSON) Data Interchange Format. IETF. 2017. URL: https://www.rfc-editor.org/rfc/rfc8259 (дата звернення: 03.04.2026).

Додаток А «Лістинг коду app.py»

```
"""
Smart Parking Monitoring & Booking System
Backend: Flask + SQLite
Дипломна робота: "Розробка інтелектуальної системи моніторингу та бронювання
паркувальних місць у міському середовищі"
"""

import os
import sqlite3
import hashlib
import secrets
import json
from datetime import datetime, timedelta
from functools import wraps
from flask import Flask, request, jsonify, g
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
app.config['SECRET_KEY'] = secrets.token_hex(32)
app.config['DATABASE'] = os.path.join(os.path.dirname(__file__), 'parking.db')

# ──────────────────────────────────────────────
# Database helpers
# ──────────────────────────────────────────────

def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(app.config['DATABASE'])
        g.db.row_factory = sqlite3.Row
        g.db.execute("PRAGMA journal_mode=WAL")
        g.db.execute("PRAGMA foreign_keys=ON")
    return g.db

@app.teardown_appcontext
def close_db(exception):
    db = g.pop('db', None)
    if db is not None:
        db.close()

def dict_from_row(row):
    return dict(row) if row else None

def dict_from_rows(rows):
    return [dict(r) for r in rows]

# ──────────────────────────────────────────────
# Schema initialization
# ──────────────────────────────────────────────

def init_db():
    db = sqlite3.connect(app.config['DATABASE'])
    db.execute("PRAGMA foreign_keys=ON")

    db.executescript("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT UNIQUE NOT NULL,
            email TEXT UNIQUE NOT NULL,
            password_hash TEXT NOT NULL,
```

```
            salt TEXT NOT NULL,
            role TEXT DEFAULT 'user' CHECK(role IN ('user', 'admin')),
            full_name TEXT,
            phone TEXT,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );

        CREATE TABLE IF NOT EXISTS parking_zones (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            address TEXT NOT NULL,
            latitude REAL NOT NULL,
            longitude REAL NOT NULL,
            total_spots INTEGER NOT NULL,
            price_per_hour REAL NOT NULL DEFAULT 25.0,
            working_hours_start TEXT DEFAULT '00:00',
            working_hours_end TEXT DEFAULT '23:59',
            zone_type TEXT DEFAULT 'street' CHECK(zone_type IN ('street',
'underground', 'multi_level', 'open')),
            is_active INTEGER DEFAULT 1,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );

        CREATE TABLE IF NOT EXISTS parking_spots (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            zone_id INTEGER NOT NULL,
            spot_number TEXT NOT NULL,
            spot_type TEXT DEFAULT 'standard' CHECK(spot_type IN ('standard',
'disabled', 'electric', 'vip')),
            status TEXT DEFAULT 'free' CHECK(status IN ('free', 'occupied',
'reserved', 'maintenance')),
            sensor_id TEXT,
            FOREIGN KEY (zone_id) REFERENCES parking_zones(id) ON DELETE CASCADE,
            UNIQUE(zone_id, spot_number)
        );

        CREATE TABLE IF NOT EXISTS bookings (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            spot_id INTEGER NOT NULL,
            zone_id INTEGER NOT NULL,
            start_time TIMESTAMP NOT NULL,
            end_time TIMESTAMP NOT NULL,
            total_price REAL NOT NULL,
            status TEXT DEFAULT 'active' CHECK(status IN ('active', 'completed',
'cancelled')),
            vehicle_plate TEXT,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            FOREIGN KEY (user_id) REFERENCES users(id),
            FOREIGN KEY (spot_id) REFERENCES parking_spots(id),
            FOREIGN KEY (zone_id) REFERENCES parking_zones(id)
        );

        CREATE TABLE IF NOT EXISTS occupancy_log (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            zone_id INTEGER NOT NULL,
            timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
            occupied_spots INTEGER NOT NULL,
            total_spots INTEGER NOT NULL,
            day_of_week INTEGER,
            hour INTEGER,
            FOREIGN KEY (zone_id) REFERENCES parking_zones(id) ON DELETE CASCADE
        );
```

```
        CREATE INDEX IF NOT EXISTS idx_bookings_user ON bookings(user_id);
        CREATE INDEX IF NOT EXISTS idx_bookings_spot ON bookings(spot_id);
        CREATE INDEX IF NOT EXISTS idx_bookings_status ON bookings(status);
        CREATE INDEX IF NOT EXISTS idx_occupancy_zone ON occupancy_log(zone_id,
day_of_week, hour);
    """)

    # Check if seed data exists
    cursor = db.execute("SELECT COUNT(*) FROM users")
    if cursor.fetchone()[0] == 0:
        seed_data(db)

    db.commit()
    db.close()

def seed_data(db):
    """Populate database with realistic demo data for Kyiv."""
    import random

    # Admin user
    salt = secrets.token_hex(16)
    pw_hash = hashlib.sha256(("admin123" + salt).encode()).hexdigest()
    db.execute(
        "INSERT INTO users (username, email, password_hash, salt, role, full_name,
phone) VALUES (?,?,?,?,?,?,?)",
        ("admin", "admin@smartparking.ua", pw_hash, salt, "admin", "Адміністратор
Системи", "+380501234567")
    )

    # Demo user
    salt2 = secrets.token_hex(16)
    pw_hash2 = hashlib.sha256(("user123" + salt2).encode()).hexdigest()
    db.execute(
        "INSERT INTO users (username, email, password_hash, salt, role, full_name,
phone) VALUES (?,?,?,?,?,?,?)",
        ("user", "user@example.com", pw_hash2, salt2, "user", "Іван Петренко",
"+380671234567")
    )

    # Parking zones in Kyiv
    zones = [
        ("Парковка Хрещатик", "вул. Хрещатик, 22", 50.4475, 30.5225, 45, 60.0,
"underground"),
        ("ТРЦ Gulliver", "пл. Спортивна, 1а", 50.4388, 30.5210, 80, 50.0,
"multi_level"),
        ("Парковка Поштова площа", "вул. Сагайдачного, 2", 50.4590, 30.5280, 30,
40.0, "open"),
        ("ТРЦ Ocean Plaza", "вул. Антоновича, 176", 50.4110, 30.5210, 70, 45.0,
"multi_level"),
        ("Парковка Золоті Ворота", "вул. Володимирська, 40", 50.4488, 30.5130, 25,
55.0, "street"),
        ("Парковка Арена Сіті", "вул. Велика Васильківська, 5", 50.4405, 30.5185,
50, 40.0, "underground"),
        ("Парковка Контрактова площа", "вул. Костянтинівська, 2", 50.4650,
30.5160, 35, 35.0, "open"),
        ("ТРЦ Lavina Mall", "вул. Берковецька, 6д", 50.4730, 30.3850, 120, 30.0,
"multi_level"),
        ("Парковка біля ЦУМу", "вул. Хрещатик, 38", 50.4460, 30.5200, 20, 65.0,
"street"),
        ("Парковка Либідська", "пл. Либідська, 1", 50.4210, 30.5250, 40, 35.0,
"open"),
    ]
```

```
    for z in zones:
        db.execute(
            "INSERT INTO parking_zones (name, address, latitude, longitude,
total_spots, price_per_hour, zone_type) VALUES (?,?,?,?,?,?,?)",
            z
        )

    # Parking spots for each zone
    spot_types = ['standard', 'standard', 'standard', 'standard', 'disabled',
'electric']
    zone_rows = db.execute("SELECT id, total_spots FROM parking_zones").fetchall()

    for zone in zone_rows:
        zone_id, total = zone[0], zone[1]
        for i in range(1, total + 1):
            stype = random.choice(spot_types)
            status = random.choices(['free', 'occupied'], weights=[0.55, 0.45])[0]
            sensor_id = f"SNS-{zone_id:02d}-{i:03d}"
            db.execute(
                "INSERT INTO parking_spots (zone_id, spot_number, spot_type,
status, sensor_id) VALUES (?,?,?,?,?)",
                (zone_id, f"{zone_id}-{i:03d}", stype, status, sensor_id)
            )

    # Historical occupancy data (last 30 days)
    now = datetime.now()
    for zone in zone_rows:
        zone_id, total = zone[0], zone[1]
        for day_offset in range(30, 0, -1):
            dt = now - timedelta(days=day_offset)
            dow = dt.weekday()
            for hour in range(7, 23):
                # Simulate realistic patterns
                if dow < 5:  # weekday
                    if 8 <= hour <= 10:
                        occ_rate = random.uniform(0.7, 0.95)
                    elif 11 <= hour <= 14:
                        occ_rate = random.uniform(0.8, 0.98)
                    elif 15 <= hour <= 18:
                        occ_rate = random.uniform(0.6, 0.85)
                    else:
                        occ_rate = random.uniform(0.2, 0.5)
                else:  # weekend
                    if 10 <= hour <= 16:
                        occ_rate = random.uniform(0.5, 0.8)
                    else:
                        occ_rate = random.uniform(0.15, 0.4)

                occupied = int(total * occ_rate)
                ts = dt.replace(hour=hour, minute=0, second=0)
                db.execute(
                    "INSERT INTO occupancy_log (zone_id, timestamp,
occupied_spots, total_spots, day_of_week, hour) VALUES (?,?,?,?,?,?)",
                    (zone_id, ts.isoformat(), occupied, total, dow, hour)
                )

    # Sample bookings
    for i in range(50):
        day_offset = random.randint(0, 14)
        start_hour = random.randint(8, 20)
        duration = random.choice([1, 2, 3, 4])
        zone = random.choice(zone_rows)
```

```
        zone_id, total = zone[0], zone[1]
        spot_id_row = db.execute("SELECT id FROM parking_spots WHERE zone_id=?
ORDER BY RANDOM() LIMIT 1", (zone_id,)).fetchone()
        if not spot_id_row:
            continue
        price_row = db.execute("SELECT price_per_hour FROM parking_zones WHERE
id=?", (zone_id,)).fetchone()
        start = (now - timedelta(days=day_offset)).replace(hour=start_hour,
minute=0, second=0)
        end = start + timedelta(hours=duration)
        total_price = duration * price_row[0]
        status = 'completed' if day_offset > 0 else random.choice(['active',
'completed'])
        plates = ["AA1234BB", "AA5678CC", "KA9012AA", "AI3456BC", "AA7890BK",
"BI1122AA", "KA3344BC"]
        db.execute(
            "INSERT INTO bookings (user_id, spot_id, zone_id, start_time,
end_time, total_price, status, vehicle_plate) VALUES (?,?,?,?,?,?,?,?)",
            (random.choice([1, 2]), spot_id_row[0], zone_id, start.isoformat(),
end.isoformat(), total_price, status, random.choice(plates))
        )

# ──────────────────────────────────────────────
# Authentication helpers
# ──────────────────────────────────────────────

# Simple token store (in production, use JWT library)
tokens = {}

def hash_password(password, salt):
    return hashlib.sha256((password + salt).encode()).hexdigest()

def create_token(user_id, role):
    token = secrets.token_hex(32)
    tokens[token] = {"user_id": user_id, "role": role, "created":
datetime.now().isoformat()}
    return token

def auth_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = request.headers.get('Authorization', '').replace('Bearer ', '')
        if token not in tokens:
            return jsonify({"error": "Необхідна авторизація"}), 401
        g.current_user = tokens[token]
        return f(*args, **kwargs)
    return decorated

def admin_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = request.headers.get('Authorization', '').replace('Bearer ', '')
        if token not in tokens:
            return jsonify({"error": "Необхідна авторизація"}), 401
        if tokens[token]['role'] != 'admin':
            return jsonify({"error": "Потрібні права адміністратора"}), 403
        g.current_user = tokens[token]
        return f(*args, **kwargs)
    return decorated

# ──────────────────────────────────────────────
# Auth endpoints
# ──────────────────────────────────────────────
```

```
@app.route('/api/auth/register', methods=['POST'])
def register():
    data = request.json
    required = ['username', 'email', 'password', 'full_name']
    if not all(k in data for k in required):
        return jsonify({"error": "Заповніть всі обов'язкові поля"}), 400

    salt = secrets.token_hex(16)
    pw_hash = hash_password(data['password'], salt)

    try:
        db = get_db()
        db.execute(
            "INSERT INTO users (username, email, password_hash, salt, full_name,
phone) VALUES (?,?,?,?,?,?)",
            (data['username'], data['email'], pw_hash, salt, data['full_name'],
data.get('phone', ''))
        )
        db.commit()

        user = db.execute("SELECT id, role FROM users WHERE username=?",
(data['username'],)).fetchone()
        token = create_token(user['id'], user['role'])

        return jsonify({
            "token": token,
            "user": {"id": user['id'], "username": data['username'], "role":
user['role'], "full_name": data['full_name']}
        }), 201
    except sqlite3.IntegrityError:
        return jsonify({"error": "Користувач з таким ім'ям або email вже існує"}),
409

@app.route('/api/auth/login', methods=['POST'])
def login():
    data = request.json
    db = get_db()
    user = db.execute("SELECT * FROM users WHERE username=? OR email=?",
                      (data.get('username', ''), data.get('username',
'')).fetchone()

    if not user:
        return jsonify({"error": "Невірний логін або пароль"}), 401

    pw_hash = hash_password(data['password'], user['salt'])
    if pw_hash != user['password_hash']:
        return jsonify({"error": "Невірний логін або пароль"}), 401

    token = create_token(user['id'], user['role'])
    return jsonify({
        "token": token,
        "user": {
            "id": user['id'],
            "username": user['username'],
            "role": user['role'],
            "full_name": user['full_name'],
            "email": user['email'],
            "phone": user['phone']
        }
    })

@app.route('/api/auth/me', methods=['GET'])
```

```
@auth_required
def get_me():
    db = get_db()
    user = db.execute("SELECT id, username, email, role, full_name, phone,
created_at FROM users WHERE id=?",
                      (g.current_user['user_id'],)).fetchone()
    return jsonify(dict_from_row(user))

# ──────────────────────────────────────────────
# Parking zones endpoints
# ──────────────────────────────────────────────

@app.route('/api/zones', methods=['GET'])
def get_zones():
    db = get_db()
    zones = db.execute("""
        SELECT z.*,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='free') as free_spots,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='occupied') as occupied_spots,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='reserved') as reserved_spots
        FROM parking_zones z WHERE z.is_active=1
        ORDER BY z.name
    """).fetchall()
    return jsonify(dict_from_rows(zones))

@app.route('/api/zones/<int:zone_id>', methods=['GET'])
def get_zone(zone_id):
    db = get_db()
    zone = db.execute("""
        SELECT z.*,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='free') as free_spots,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='occupied') as occupied_spots,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='reserved') as reserved_spots
        FROM parking_zones z WHERE z.id=?
    """, (zone_id,)).fetchone()

    if not zone:
        return jsonify({"error": "Зону не знайдено"}), 404

    spots = db.execute("SELECT * FROM parking_spots WHERE zone_id=? ORDER BY
spot_number", (zone_id,)).fetchall()

    result = dict_from_row(zone)
    result['spots'] = dict_from_rows(spots)
    return jsonify(result)

@app.route('/api/zones', methods=['POST'])
@admin_required
def create_zone():
    data = request.json
    db = get_db()
    cursor = db.execute(
        """INSERT INTO parking_zones (name, address, latitude, longitude,
total_spots, price_per_hour, zone_type)
           VALUES (?,?,?,?,?,?,?)""",
        (data['name'], data['address'], data['latitude'], data['longitude'],
```

```
         data['total_spots'], data.get('price_per_hour', 25.0),
data.get('zone_type', 'street'))
    )
    zone_id = cursor.lastrowid

    # Create spots
    for i in range(1, data['total_spots'] + 1):
        db.execute(
            "INSERT INTO parking_spots (zone_id, spot_number, sensor_id) VALUES
(?,?,?)",
            (zone_id, f"{zone_id}-{i:03d}", f"SNS-{zone_id:02d}-{i:03d}")
        )

    db.commit()
    return jsonify({"id": zone_id, "message": "Зону створено"}), 201

@app.route('/api/zones/<int:zone_id>', methods=['PUT'])
@admin_required
def update_zone(zone_id):
    data = request.json
    db = get_db()
    db.execute(
        """UPDATE parking_zones SET name=?, address=?, price_per_hour=?,
zone_type=?, is_active=?
           WHERE id=?""",
        (data['name'], data['address'], data['price_per_hour'], data['zone_type'],
data.get('is_active', 1), zone_id)
    )
    db.commit()
    return jsonify({"message": "Зону оновлено"})

@app.route('/api/zones/<int:zone_id>', methods=['DELETE'])
@admin_required
def delete_zone(zone_id):
    db = get_db()
    db.execute("DELETE FROM parking_zones WHERE id=?", (zone_id,))
    db.commit()
    return jsonify({"message": "Зону видалено"})

# ──────────────────────────────────────────────
# Booking endpoints
# ──────────────────────────────────────────────

@app.route('/api/bookings', methods=['POST'])
@auth_required
def create_booking():
    data = request.json
    db = get_db()

    spot = db.execute("SELECT * FROM parking_spots WHERE id=? AND status='free'",
(data['spot_id'],)).fetchone()
    if not spot:
        return jsonify({"error": "Місце недоступне для бронювання"}), 400

    zone = db.execute("SELECT * FROM parking_zones WHERE id=?",
(data['zone_id'],)).fetchone()
    if not zone:
        return jsonify({"error": "Зону не знайдено"}), 404

    start_time = datetime.fromisoformat(data['start_time'])
    end_time = datetime.fromisoformat(data['end_time'])
    duration_hours = max(1, (end_time - start_time).total_seconds() / 3600)
    total_price = round(duration_hours * zone['price_per_hour'], 2)
```

```
    # Check for overlapping bookings
    overlap = db.execute("""
        SELECT id FROM bookings
        WHERE spot_id=? AND status='active'
        AND start_time < ? AND end_time > ?
    """, (data['spot_id'], data['end_time'], data['start_time'])).fetchone()

    if overlap:
        return jsonify({"error": "Цей час вже заброньовано"}), 409

    cursor = db.execute(
        """INSERT INTO bookings (user_id, spot_id, zone_id, start_time, end_time,
total_price, vehicle_plate)
           VALUES (?,?,?,?,?,?,?)""",
        (g.current_user['user_id'], data['spot_id'], data['zone_id'],
         data['start_time'], data['end_time'], total_price,
data.get('vehicle_plate', ''))
    )

    db.execute("UPDATE parking_spots SET status='reserved' WHERE id=?",
(data['spot_id'],))
    db.commit()

    return jsonify({
        "id": cursor.lastrowid,
        "total_price": total_price,
        "message": "Бронювання створено успішно"
    }), 201

@app.route('/api/bookings', methods=['GET'])
@auth_required
def get_bookings():
    db = get_db()
    user_id = g.current_user['user_id']
    role = g.current_user['role']

    if role == 'admin':
        bookings = db.execute("""
            SELECT b.*, z.name as zone_name, z.address as zone_address,
                   u.full_name as user_name, ps.spot_number
            FROM bookings b
            JOIN parking_zones z ON b.zone_id=z.id
            JOIN users u ON b.user_id=u.id
            JOIN parking_spots ps ON b.spot_id=ps.id
            ORDER BY b.created_at DESC
        """).fetchall()
    else:
        bookings = db.execute("""
            SELECT b.*, z.name as zone_name, z.address as zone_address,
ps.spot_number
            FROM bookings b
            JOIN parking_zones z ON b.zone_id=z.id
            JOIN parking_spots ps ON b.spot_id=ps.id
            WHERE b.user_id=?
            ORDER BY b.created_at DESC
        """, (user_id,)).fetchall()

    return jsonify(dict_from_rows(bookings))

@app.route('/api/bookings/<int:booking_id>/cancel', methods=['PUT'])
@auth_required
def cancel_booking(booking_id):
```

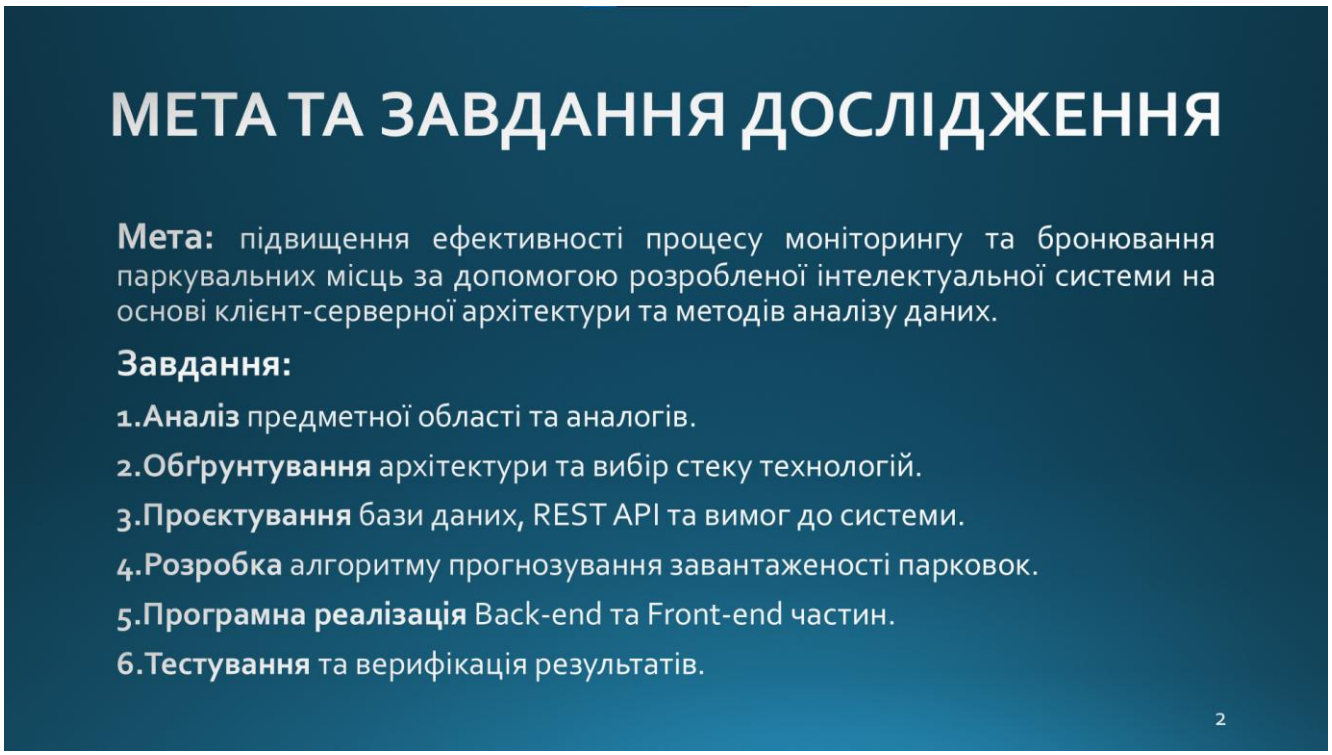
```
    db = get_db()
    booking = db.execute("SELECT * FROM bookings WHERE id=?",
(booking_id,)).fetchone()

    if not booking:
        return jsonify({"error": "Бронювання не знайдено"}), 404

    if booking['user_id'] != g.current_user['user_id'] and g.current_user['role']
!= 'admin':
        return jsonify({"error": "Недостатньо прав"}), 403

    db.execute("UPDATE bookings SET status='cancelled' WHERE id=?", (booking_id,))
    db.execute("UPDATE parking_spots SET status='free' WHERE id=?",
(booking['spot_id'],))
    db.commit()

    return jsonify({"message": "Бронювання скасовано"})

# ──────────────────────────────────────────────
# Intelligence: Prediction & Recommendation
# ──────────────────────────────────────────────

@app.route('/api/predict/<int:zone_id>', methods=['GET'])
def predict_occupancy(zone_id):
    """Predict occupancy based on historical averages by day_of_week and hour."""
    db = get_db()

    target_day = request.args.get('day', datetime.now().weekday(), type=int)

    predictions = db.execute("""
        SELECT hour,
               ROUND(AVG(occupied_spots * 1.0 / total_spots) * 100, 1) as
avg_occupancy,
               MIN(occupied_spots * 1.0 / total_spots * 100) as min_occupancy,
               MAX(occupied_spots * 1.0 / total_spots * 100) as max_occupancy,
               COUNT(*) as data_points
        FROM occupancy_log
        WHERE zone_id=? AND day_of_week=?
        GROUP BY hour
        ORDER BY hour
    """, (zone_id, target_day)).fetchall()

    zone = db.execute("SELECT name FROM parking_zones WHERE id=?",
(zone_id,)).fetchone()

    return jsonify({
        "zone_id": zone_id,
        "zone_name": zone['name'] if zone else "",
        "day_of_week": target_day,
        "predictions": dict_from_rows(predictions)
    })

@app.route('/api/recommend', methods=['GET'])
def recommend_parking():
    """Recommend nearest available parking to given coordinates."""
    lat = request.args.get('lat', type=float)
    lng = request.args.get('lng', type=float)

    if not lat or not lng:
        return jsonify({"error": "Вкажіть координати (lat, lng)"}), 400

    db = get_db()
    zones = db.execute("""
```

```
        SELECT z.*,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='free') as free_spots,
            ((z.latitude - ?) * (z.latitude - ?) + (z.longitude - ?) *
(z.longitude - ?)) as distance_sq
        FROM parking_zones z
        WHERE z.is_active=1
          AND (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='free') > 0
        ORDER BY distance_sq ASC
        LIMIT 5
    """, (lat, lat, lng, lng)).fetchall()

    results = []
    for z in zones:
        d = dict(z)
        d['distance_km'] = round((d['distance_sq'] ** 0.5) * 111, 2)  # rough
conversion
        del d['distance_sq']
        results.append(d)

    return jsonify(results)

# ──────────────────────────────────────────────
# Analytics (Admin)
# ──────────────────────────────────────────────

@app.route('/api/analytics/dashboard', methods=['GET'])
@auth_required
def analytics_dashboard():
    db = get_db()

    total_zones = db.execute("SELECT COUNT(*) as c FROM parking_zones WHERE
is_active=1").fetchone()['c']
    total_spots = db.execute("SELECT COUNT(*) as c FROM
parking_spots").fetchone()['c']
    free_spots = db.execute("SELECT COUNT(*) as c FROM parking_spots WHERE
status='free'").fetchone()['c']
    occupied_spots = db.execute("SELECT COUNT(*) as c FROM parking_spots WHERE
status='occupied'").fetchone()['c']
    reserved_spots = db.execute("SELECT COUNT(*) as c FROM parking_spots WHERE
status='reserved'").fetchone()['c']

    total_bookings = db.execute("SELECT COUNT(*) as c FROM
bookings").fetchone()['c']
    active_bookings = db.execute("SELECT COUNT(*) as c FROM bookings WHERE
status='active'").fetchone()['c']

    revenue = db.execute("SELECT COALESCE(SUM(total_price), 0) as total FROM
bookings WHERE status IN ('active','completed')").fetchone()['total']

    today_bookings = db.execute("""
        SELECT COUNT(*) as c FROM bookings
        WHERE DATE(created_at) = DATE('now')
    """).fetchone()['c']

    # Revenue by day (last 14 days)
    revenue_by_day = db.execute("""
        SELECT DATE(created_at) as date, SUM(total_price) as revenue, COUNT(*) as
bookings
        FROM bookings
        WHERE status IN ('active','completed') AND created_at >= DATE('now', '-14
days')
```

```
        GROUP BY DATE(created_at)
        ORDER BY date
    """).fetchall()

    # Occupancy by zone
    zone_stats = db.execute("""
        SELECT z.name,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='free') as free,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='occupied') as occupied,
            (SELECT COUNT(*) FROM parking_spots WHERE zone_id=z.id AND
status='reserved') as reserved,
            z.total_spots
        FROM parking_zones z WHERE z.is_active=1
    """).fetchall()

    # Popular hours
    popular_hours = db.execute("""
        SELECT hour, ROUND(AVG(occupied_spots * 1.0 / total_spots) * 100, 1) as
avg_occupancy
        FROM occupancy_log
        WHERE day_of_week < 5
        GROUP BY hour
        ORDER BY hour
    """).fetchall()

    return jsonify({
        "total_zones": total_zones,
        "total_spots": total_spots,
        "free_spots": free_spots,
        "occupied_spots": occupied_spots,
        "reserved_spots": reserved_spots,
        "total_bookings": total_bookings,
        "active_bookings": active_bookings,
        "today_bookings": today_bookings,
        "total_revenue": round(revenue, 2),
        "revenue_by_day": dict_from_rows(revenue_by_day),
        "zone_stats": dict_from_rows(zone_stats),
        "popular_hours": dict_from_rows(popular_hours)
    })

@app.route('/api/analytics/occupancy-history/<int:zone_id>', methods=['GET'])
@auth_required
def occupancy_history(zone_id):
    db = get_db()
    days = request.args.get('days', 7, type=int)

    history = db.execute("""
        SELECT timestamp, occupied_spots, total_spots,
               ROUND(occupied_spots * 1.0 / total_spots * 100, 1) as occupancy_pct
        FROM occupancy_log
        WHERE zone_id=? AND timestamp >= DATETIME('now', ?)
        ORDER BY timestamp
    """, (zone_id, f'-{days} days')).fetchall()

    return jsonify(dict_from_rows(history))

# ─────────────────────────────────────────────
# Admin: Users management
# ─────────────────────────────────────────────

@app.route('/api/admin/users', methods=['GET'])
```

```
@admin_required
def get_users():
    db = get_db()
    users = db.execute("SELECT id, username, email, role, full_name, phone,
created_at FROM users ORDER BY id").fetchall()
    return jsonify(dict_from_rows(users))

# ──────────────────────────────────────────────
# Simulate sensor data update
# ──────────────────────────────────────────────

@app.route('/api/simulate/update', methods=['POST'])
def simulate_sensor_update():
    """Simulate IoT sensor data changes for demo purposes."""
    import random
    db = get_db()

    spots = db.execute("SELECT id, status FROM parking_spots WHERE status IN
('free', 'occupied')").fetchall()
    changed = 0
    for spot in spots:
        if random.random() < 0.1:  # 10% chance of status change
            new_status = 'occupied' if spot['status'] == 'free' else 'free'
            db.execute("UPDATE parking_spots SET status=? WHERE id=?",
(new_status, spot['id']))
            changed += 1

    # Log current occupancy
    zones = db.execute("SELECT id, total_spots FROM parking_zones WHERE
is_active=1").fetchall()
    now = datetime.now()
    for zone in zones:
        occupied = db.execute(
            "SELECT COUNT(*) as c FROM parking_spots WHERE zone_id=? AND
status='occupied'",
            (zone['id'],)
        ).fetchone()['c']
        db.execute(
            "INSERT INTO occupancy_log (zone_id, occupied_spots, total_spots,
day_of_week, hour) VALUES (?,?,?,?,?)",
            (zone['id'], occupied, zone['total_spots'], now.weekday(), now.hour)
        )

    db.commit()
    return jsonify({"changed_spots": changed, "message": "Дані сенсорів
оновлено"})

# ──────────────────────────────────────────────
# Run
# ──────────────────────────────────────────────

if __name__ == '__main__':
    init_db()
    app.run(debug=True, host='0.0.0.0', port=5000)
```

Додаток Б «Презентація»



ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Об'єкт дослідження** – процес моніторингу та управління паркувальними місцями в умовах міського середовища.
- **Предмет дослідження** – методи та програмні засоби побудови інтелектуальної системи моніторингу, бронювання та прогнозування завантаженості паркувальних місць.

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА ІСНУЮЧИХ СИСТЕМ ПАРКУВАННЯ

Країна походження	Система	IoT-сенсори	Бронювання	Прогнозування	Динамічне ціноутворення
США	SFpark	Так	Ні	Ні	Так
США	ParkMobile	Ні	Частково	Ні	Ні
США	SpotHero	Ні	Так	Ні	Ні
Велика Британія	Parkopedia	Частково	Частково	Так	Ні
США	LA Express Park	Так	Ні	Ні	Так
Україна	Київ Цифровий	Ні	Ні	Ні	Ні
Іспанія	Barcelona Smart Parking	Так	Ні	Частково	Ні

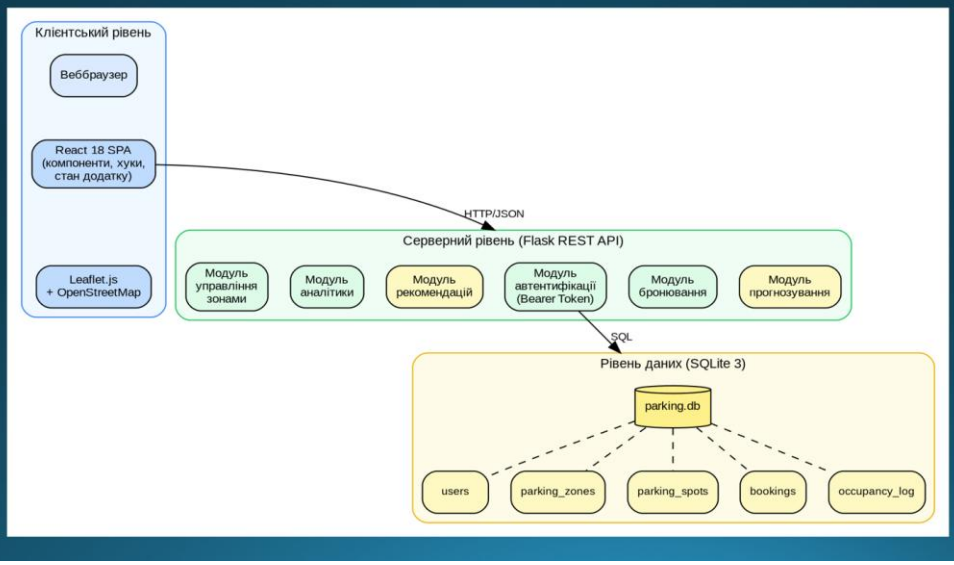
ТЕХНОЛОГІЧНИЙ СТЕК ТА АРХІТЕКТУРА

- **Back-end: Python 3.12** та мікрофреймворк **Flask 3.0**.
  - *Переваги:* гнучкість налаштувань, швидка розробка API, модульність (Blueprints).
- **Front-end: React 18** + інструмент збірки **Vite**.
  - *Переваги:* компонентний підхід, віртуальний DOM, висока швидкість розробки.
- **База даних: SQLite 3**.
  - *Переваги:* портативність (один файл), достатня продуктивність, проста міграція на PostgreSQL у майбутньому.
- **Картографія: Leaflet.js + OpenStreetMap**.
  - *Переваги:* open-source рішення без плати за API-ключі (на відміну від Google Maps).

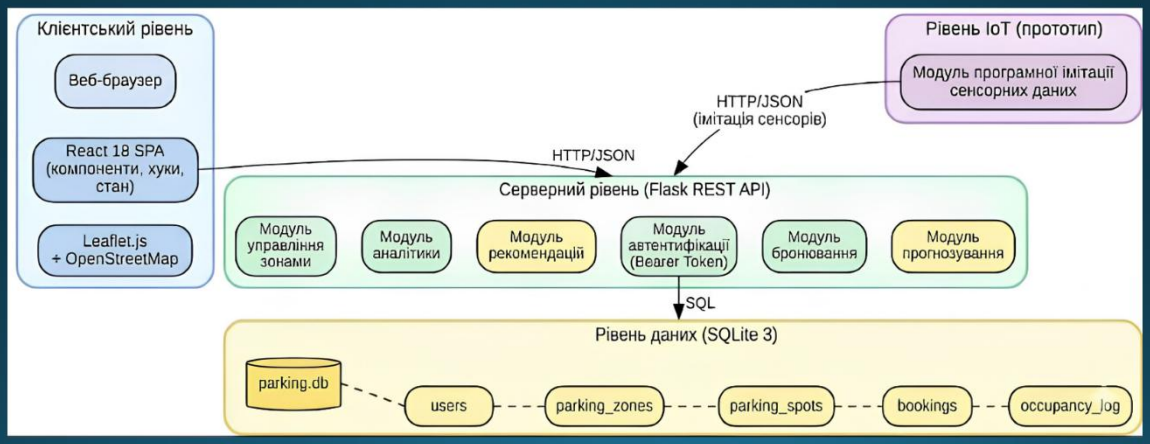
ПРОЄКТУВАННЯ СИСТЕМИ: РОЛІ ТА СТРУКТУРА ВИМОГ

- **Категорії користувачів (Актори):**
  - **Водій (Користувач):** Пошук на карті, бронювання місць, отримання AI-прогнозів та рекомендацій.
  - **Адміністратор:** Управління зонами, моніторинг KPI (дашборди), управління користувачами.
- **Групи функціональних вимог (15 вимог):**
  - **Автентифікація:** Реєстрація, вхід (Bearer Token), рольова модель.
  - **Моніторинг:** Інтерактивна карта (Leaflet), візуалізація завантаженості (зелений/жовтий/червоний).
  - **Бронювання:** Вибір місця на сітці, розрахунок вартості, перевірка часових перетинів.
  - **Інтелектуальні функції:** Прогноз завантаженості по годинах, пошук найближчих вільних зон.
  - **Адміністрування:** CRUD парковок, аналітика доходу, імітація датчиків.

ЗАГАЛЬНА АРХІТЕКТУРА СИСТЕМИ SMARTPARKING

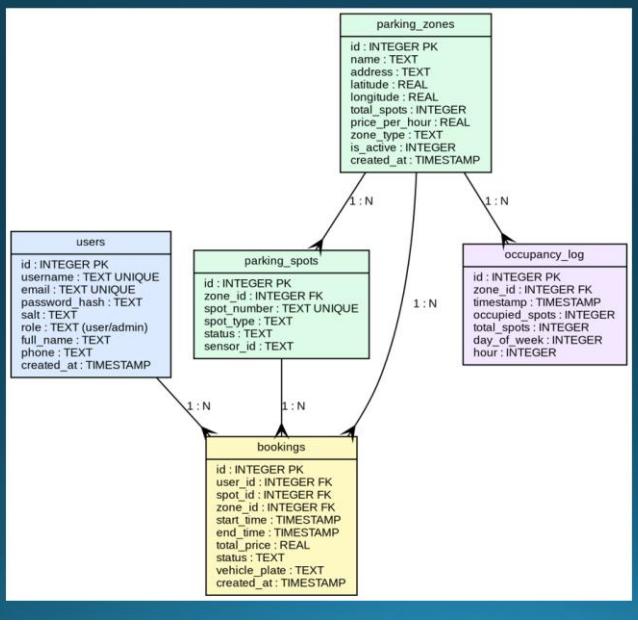


ЗАГАЛЬНА АРХІТЕКТУРА СИСТЕМИ SMARTPARKING

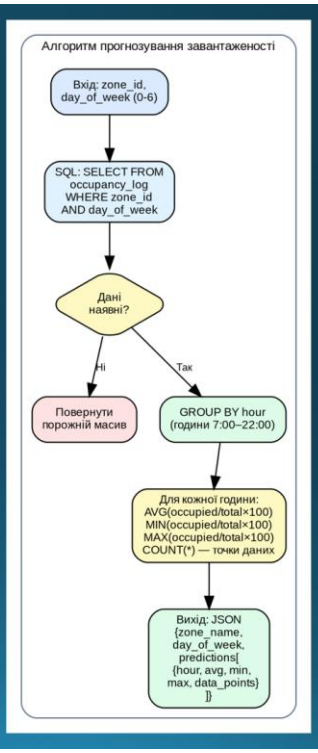


ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ СИСТЕМИ SMARTPARKING

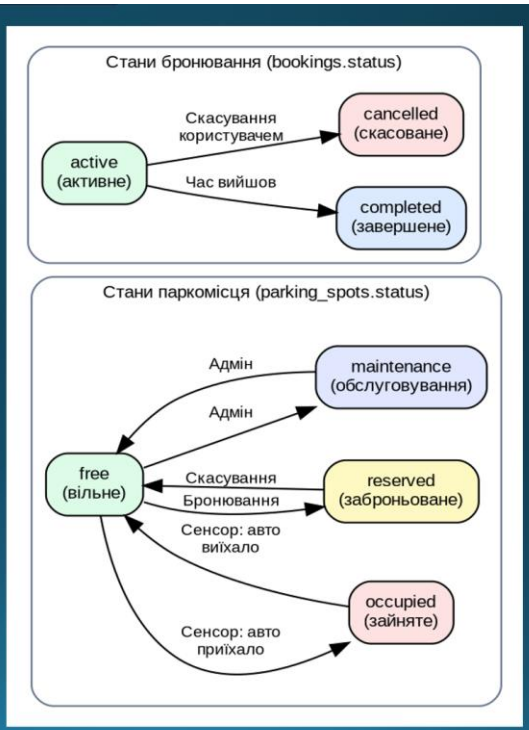
ER-ДІАГРАМА БАЗИ ДАНИХ СИСТЕМИ SMARTPARKING



БЛОК-СХЕМА АЛГОРИТМУ ПРОГНОЗУВАННЯ ЗАВАНТАЖЕНОСТІ

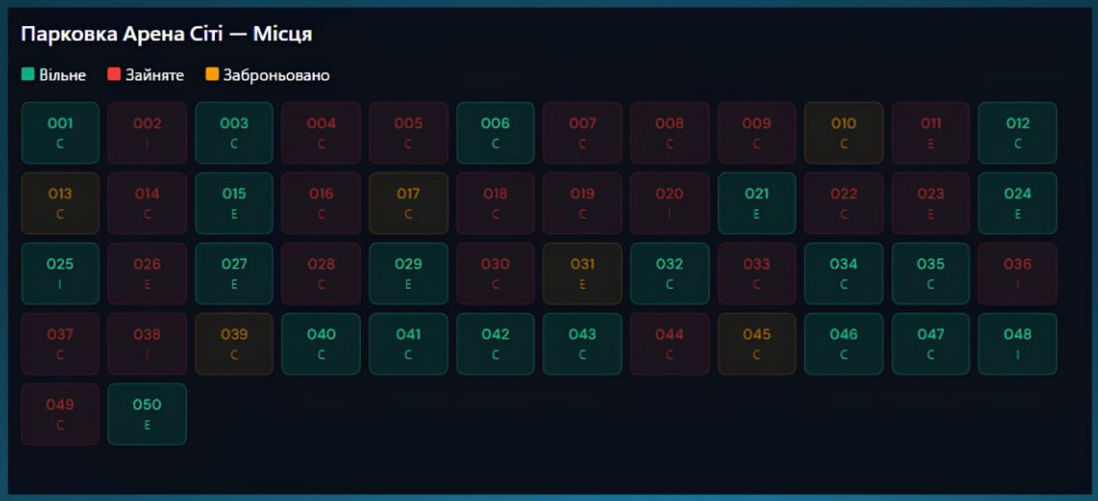


ДІАГРАМИ ПЕРЕХОДІВ СТАНІВ ПАРКОМІСЦЯ ТА БРОНЮВАННЯ

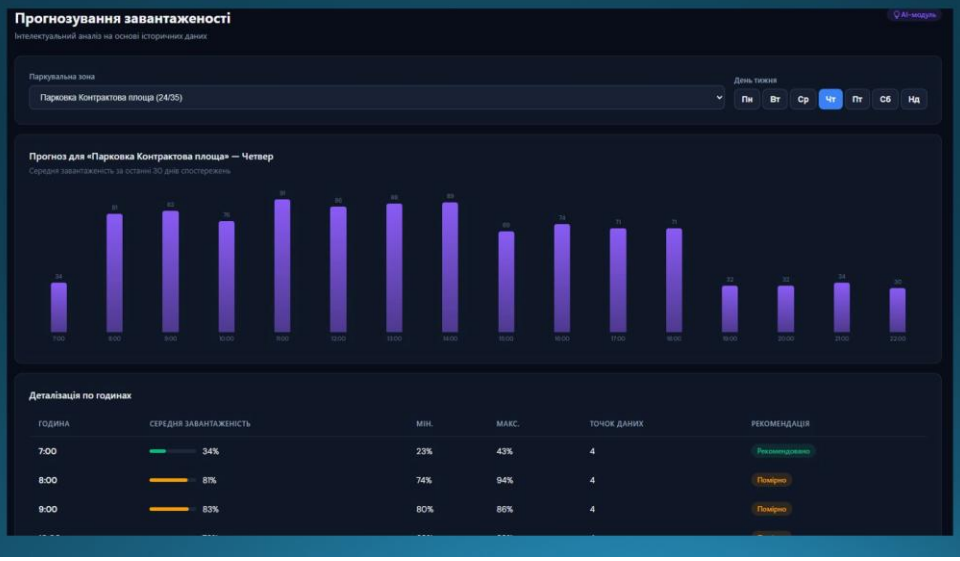


ПАНЕЛЬ МОНІТОРИНГУ (ДАШБОРД) СИСТЕМИ SMARTPARKING

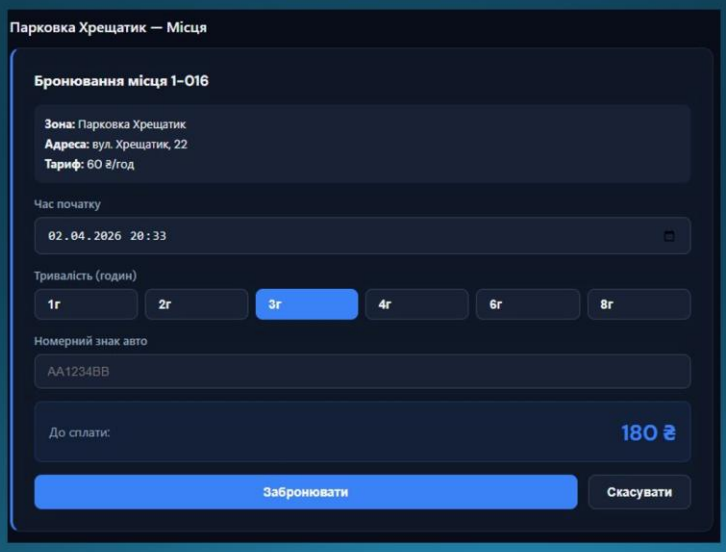
ВІЗУАЛЬНА СІТКА ПАРКОМІСЦЬ ЗОНИ «АРЕНА СІТІ»

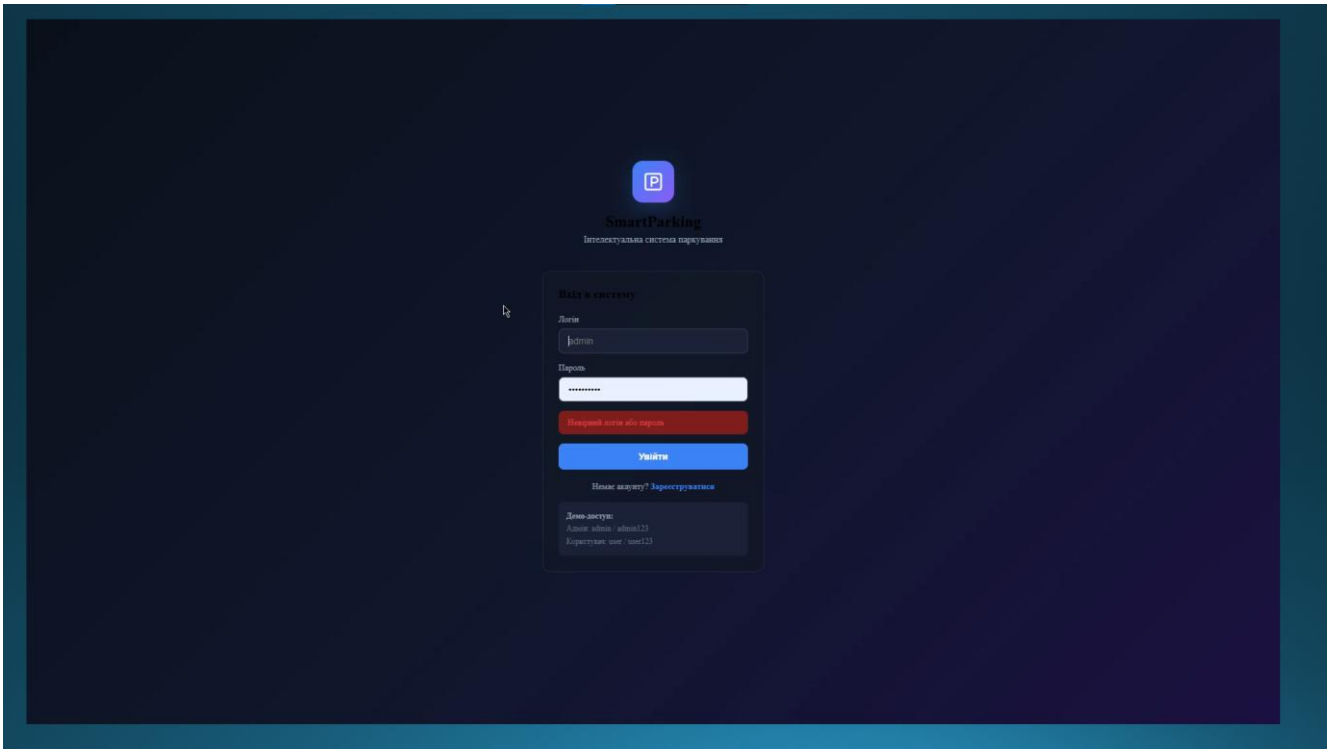


СТОРІНКА ПРОГНОЗУВАННЯ ЗАВАНТАЖЕНОСТІ ПАРКУВАЛЬНИХ ЗОН



ФОРМА БРОНЮВАННЯ ПАРКОМІСЦЯ 1-016 ЗОНИ «ПАРКОВКА ХРЕЩАТИК»





АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1.Наконечний В.В. Алгоритми прогнозування завантаженості та рекомендацій паркувальних місць у системі інтелектуального управління паркуванням // Матеріали VI Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15.05.2026, ДУІКТ, Київ, Україна , (подано до друку)

2.Наконечний В.В. Розробка інтелектуальної системи моніторингу та бронювання паркувальних місць у міському середовищі // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація»,1-3.05.2026 , ДУІКТ, Київ, Україна , (подано до друку)

ВИСНОВКИ

- **Обґрунтування:** Аналіз ринку підтвердив дефіцит паркомісць у Києві (1,38 млн авто на 30 тис. місць) та відсутність комплексних рішень з прогнозуванням.
- **Розробка:** Створено Full-stack додаток (**Flask + React**) з 4-рівневою архітектурою та 15-ма REST API ендпоінтами.
- **Інтелектуальний модуль:** Реалізовано алгоритми статистичного прогнозування завантаженості (на основі 30-денної історії) та пошуку найближчих вільних місць.
- **Безпека та якість:** Впроваджено хешування **SHA-256** та рольову модель доступу. Час відповіді системи — **до 200 мс** (в 2.5 рази швидше за норматив).
- **Верифікація:** Успішно протестовано 12 сценаріїв використання (від реєстрації до адміністрування).

ДЯКУЮ ЗА УВАГУ!