Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування інституту, факультету)

Кафедра інформаційної безпеки та наноелектроніки
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему «Аналіз продуктивності захищених PHP та Python Web-додатків при доступі до інформації бази даних MySQL»

Виконав: студент 2 курсу, групи БК-814м
Спеціальності 125 Кібербезпека та захист інформації
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Безпека інформаційних і комунікаційних систем

ГРИШАЙ Д.О.
(ПРІЗВИЩЕ та ініціали)

Керівник КОРОЛЬКОВ Р.Ю.
(ПРІЗВИЩЕ та ініціали)

Рецензент САМОЙЛИК С.С.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет інформаційної безпеки та електронних комунікацій
Кафедра інформаційної безпеки та наноелектроніки
Ступінь вищої освіти магістр
Спеціальність 125 Кібербезпека та захист інформації
(код і найменування)
Освітня програма (спеціалізація) Безпека інформаційних і комунікаційних систем

(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІБтаН,
к. ф.-м.. н, доцент
Андрій КОРОТУН
«_____» _____2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ГРИШАЯ Данила Олександровича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Аналіз продуктивності захищених PHP та Python Web-додатків при доступі до інформації бази даних MySQL
Performance analysis of secure PHP and Python web applications when accessing MySQL database information
керівник проєкту (роботи) к.т.н., доцент КОРОЛЬКОВ Роман Юійович
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти № 530 від 26 листопада 2025 р.
2. Строк подання студентом проєкту (роботи) 23 січня 2026 року
3. Вихідні дані до проєкту (роботи): вимоги до функціоналу web-додатку (автентифікація/авторизація), вимоги до безпеки та продуктивності
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити: 1 Аналіз методів доступу до баз даних MYSQL та теоретичні основи web-додатків. 2 Проєктування та реалізація web-додатку на Python з використанням MYSQL. 3 Аналіз продуктивності та тестування web-додатку .
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): презентація роботи (18 слайдів)

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	Прийняв виконане завдання
Розділи 1-3	Корольков Р.Ю., доц. каф. ІБтаН	10.10	
Нормоконтроль	Корольков Р.Ю., доц. каф. ІБтаН	25.12	

7. Дата видачі завдання «10» жовтня 2025 року

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	01.10.2025	Виконано
2	Розгляд архітектури та класифікації web-додатків.	01.10-06.10.2025	Виконано
3	Аналіз методів доступу до баз даних MySQL у web-додатках, створених на PHP та Python.	07.10-15.10.2025	Виконано
4	Огляд основних видів атак на бази даних і методів захисту від них.	16.10-27.10.2025	Виконано
5	Аналіз вимог до досліджуваного web-додатку та вибір технологій його розробки.	28.10-10.11.2025	Виконано
6	Проєктування структури бази даних MySQL.	11.11-17.11.2025	Виконано
7	Реалізація клієнтської та серверної частини web-додатку, забезпечення його взаємодії з базою даних.	18.11-01.12.2025	Виконано
8	Проведення тестування та аналіз результатів.	02.12-08.12.2025	Виконано
9	Оформлення пояснювальної записки та відповідної документації.	09.12-15.12.2025	Виконано
10	Нормоконтроль та рецензування.	16.12-19.12.2025	Виконано
11	Захист дипломної роботи.		Виконано

Студент _____ Данило ГРИШАЙ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Роман КОРОЛЬКОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

АНОТАЦІЯ

Пояснювальна записка до магістерської роботи: 160 с., 2 табл., 42 рис., 7 дод., 54 джерела.

БАЗА ДАНИХ, СИСТЕМА КЕРУВАННЯ БАЗАМИ ДАНИХ, WEB-ДОДАТОК, MYSQL, ПРОДУКТИВНІСТЬ, ORM, SQL-ЗАПИТ, CRUD-ОПЕРАЦІЇ, ЗАХИСТ ДАНИХ, ТЕСТУВАННЯ ПРОДУКТИВНОСТІ.

Об’єктом дослідження є web-додаток з захищеним доступом до бази даних MySQL, що виконує функції обробки та управління даними.

Предметом дослідження є методи доступу до бази даних (RAW SQL, параметризовані запити, ORM) та їхній вплив на продуктивність web-додатку.

Метою роботи є аналіз продуктивності web-додатку при застосуванні різних методів доступу до бази даних MySQL та визначення оптимальних підходів для забезпечення його швидкодії та безпеки.

У даній роботі було розглянуто проблеми підвищення продуктивності та безпеки доступу до баз даних у web-додатках на прикладі системи керування базами даних MySQL. Було проведено аналіз вимог до досліджуваного додатку та обґрунтовано вибір технологій розробки і інструментів для тестування. Виконано послідовне та навантажувальне тестування CRUD-операцій, JOIN-запитів та агрегатних функцій, оцінено вплив різних методів доступу до інформації бази даних на швидкодію, стабільність і споживання ресурсів. Також було проаналізовано результати експериментів, які дозволяють зробити висновки щодо оптимальних підходів до організації доступу до бази даних.

Отримані результати можуть бути використані для підвищення ефективності та безпеки web-додатків при роботі з базами даних.

ABSTRACT

Master's thesis explanatory note: 160 pages, 2 tables, 42 figures, 7 appendices, 54 references.

DATABASE, DATABASE MANAGEMENT SYSTEM, WEB APPLICATION, MYSQL, PERFORMANCE, ORM, SQL QUERY, CRUD OPERATIONS, DATA PROTECTION, PERFORMANCE TESTING.

The object of the study is a web application with secure access to a MySQL database that performs data processing and management functions.

The subject of the study is database access methods (raw SQL, parameterized queries, ORM) and their impact on the performance of the web application.

The purpose of the work is to analyze the performance of a web application when using different methods of accessing a MySQL database and to determine optimal approaches to ensure its speed and security.

This work addresses issues of improving the performance and security of database access in web applications using the MySQL database management system as a case study. The requirements for the application under study were analyzed, and the choice of development technologies and performance testing tools was justified. Sequential and load testing of CRUD operations, JOIN queries, and aggregate functions was conducted, and the influence of different database access methods on performance, stability, and resource consumption was evaluated. The experimental results were also analyzed, making it possible to draw conclusions about optimal approaches to organizing database access.

The obtained results can be used to improve the efficiency and security of web applications when working with databases.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API – Application Programming Interface;

B2B – business-to-business;

CPU – Central processing unit;

CSS – Cascading Style Sheets;

EDA – Event-driven architecture;

ERD – Entity-relationship model;

ERR – Enhanced entity-relationship;

HTML – HyperText Markup Language;

HTTP – HyperText Transfer Protocol;

JS – JavaScript;

ORM – Object-relational mapping;

P2P – peer-to-peer;

PBKDF2 – Password-based key derivation function 2;

PDO – PHP Data Objects;

RDBMS – Relational database management system;

RPS – Requests per second;

SHA-256 – Secure hash algorithm 256-bit;

SOA – Service-oriented architecture;

SOAP – Simple Object Access Protocol;

SSRF – Server-side request forgery;

SQL – Structured Query Language;

SSL – Secure sockets layer;

TLS – Transport layer security;

TPS – Transaction per second;

БД – база даних;

ПЗ – програмне забезпечення;

СКБД – система керування базами даних;

CRUD – абревіатура для чотирьох основних операцій над даними в програмуванні та базах даних: create (створення), read (читання), update (оновлення), delete (видалення), що дозволяють керувати інформацією: додавати, переглядати, змінювати та видаляти записи.

Foreign Key (FK) – поле в таблиці, яке посилається на первинний ключ іншої таблиці, встановлюючи між ними зв’язок. Він забезпечує цілісність даних, не дозволяючи створювати записи без відповідного елементу в пов’язаній таблиці.

MEM – середнє значення використання оперативної пам’яті під час виконання тесту.

pbkdf2:sha256 – алгоритм безпечного хешування паролів, який поєднує механізми PBKDF2 з SHA-256.

PHP – скриптова мова програмування загального призначення, яка найчастіше використовується для розробки серверної частини веб-додатків.

Primary Key (PK) – унікальний ідентифікатор запису в таблиці бази даних, який гарантує, що жоден рядок не повторюється. Він використовується для однозначної ідентифікації кожного елементу даних.

Python – інтерпретована, інтерактивна, об’єктно-орієнтована мова програмування, яка підтримує кілька парадигм програмування, має високий рівень абстракції й чіткий синтаксис.

RAW – SQL-команда, яка безпосередньо відправляється до бази даних, без будь-яких обгорток або додаткових бібліотек (ORM), і виконується саме так, як її написали ("сирий SQL-запит").

RSS – обсяг оперативної пам’яті, яку використовує процес.

Автентифікація – процес підтвердження особи користувача, пристрою або служби, щоб переконатися, що вони є тими, за кого себе видають, перед тим як надати доступ до системи, даних чи ресурсів.

Авторизація – процес надання або перевірки прав доступу користувача до певних ресурсів, функцій чи дій у системі після того, як його особу було встановлено (автентифіковано).

База даних – це впорядкована сукупність взаємопов’язаних даних, призначених для довготривалого зберігання, обробки та швидкого доступу до інформації.

Клієнт – пристрій або програмне забезпечення, яке робить запити до сервера для отримання даних або послуг (наприклад, веб-браузер, що відкриває вебсторінку).

Сервер – пристрій або програмне забезпечення, яке приймає запити від клієнтів і надає їм потрібні дані або послуги (наприклад, вебсервер, що віддає сторінки або обробляє запити API).

Система керування базами даних – спеціалізоване програмне забезпечення, яке слугує посередником між користувачами та даними, дозволяючи створювати, зберігати, оновлювати, шукати та керувати великими обсягами структурованої інформації, забезпечуючи безпеку та цілісність даних.

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій web-додатки стали невід'ємною складовою більшості інформаційних систем. Вони широко використовуються у багатьох сферах людського життя – від електронної комерції та банківських сервісів до освіти, охорони здоров'я й державного управління.

У більшості випадків вебдодатки працюють із великими обсягами даних, збережених у системах керування базами даних, що зумовлює необхідність забезпечення швидкого, надійного та безпечного доступу до інформації. Ефективність роботи вебдодатку значною мірою визначається не тільки обраною системою керування базами даних, а і методами взаємодії з нею.

В умовах цифровізації суспільства до web-додатків висуваються підвищені вимоги щодо рівня безпеки та продуктивності. Сучасний вебдодаток повинен не лише забезпечувати коректну обробку запитів користувачів, але й гарантувати захист даних від несанкціонованого доступу, атак та витоків інформації. Водночас застосування механізмів безпеки не повинно суттєво погіршувати швидкодію системи, особливо під час виконання операцій доступу до бази даних.

Оптимізація процесів взаємодії з базою даних є одним із ключових чинників забезпечення високої продуктивності веб додатку. Неефективні SQL-запити, надмірна кількість звернень до бази даних або некоректне використання механізмів доступу можуть призводити до значних затримок у роботі системи та зниження її масштабованості. Водночас реалізація захисних механізмів, таких як автентифікація, авторизація, перевірка вхідних даних тощо, створює додаткове навантаження на систему.

У зв'язку з цим особливої актуальності набуває дослідження та аналіз впливу методів забезпечення безпеки на продуктивність web-додатків, розроблених мовами PHP та Python – одними з найпопулярніших мов

програмування, – під час доступу до інформації бази даних MySQL, яка наразі є однією з флагманських систем керування базами даних.

У даній роботі буде розглянуто основні методи доступу до бази даних MySQL у вебдодатках, розроблених мовами PHP та Python, а також проаналізовано підходи до забезпечення безпеки під час виконання операцій взаємодії з базою даних. Особливу увагу приділено дослідженню впливу захисних механізмів на продуктивність web-додатків, зокрема під час виконання типових операцій читання, запису, створення та видалення даних.

Завершення проведеного дослідження сприятиме визначенню оптимальних підходів до проєктування та реалізації захищених і водночас високопродуктивних web-додатків на PHP та Python, а також формуванню практичних рекомендацій щодо вибору методів доступу до бази даних і механізмів безпеки з урахуванням вимог до швидкодії та надійності системи.

1 АНАЛІЗ МЕТОДІВ ДОСТУПУ ДО БАЗ ДАНИХ MYSQL ТА ТЕОРЕТИЧНІ ОСНОВИ WEB-ДОДАТКІВ

1.1 Поняття та класифікація web-додатків

Web-додаток – це клієнт-серверна програма, яка використовує web-браузер в якості клієнта для взаємодії з сервером. Він функціонує наступним чином: користувач через браузер надсилає запити, сервер обробляє їх і повертає відповіді. На відміну від звичайного вебсайту, вебдодаток дозволяє користувачам виконувати дії, які змінюють стан бізнес-логіки або даних на сервері. По суті, web-додаток використовує вебсайт як інтерфейс для взаємодії з серверною програмою, забезпечуючи динамічну функціональність та інтерактивність для користувача [1].

Більшість вебдодатків реалізовані за клієнт-серверною архітектурою, де веббраузер виступає клієнтом для взаємодії із сервером, що обробляє бізнес-логіку та зберігає дані. Користувач надсилає запити через браузер, сервер обробляє їх і повертає відповіді. Водночас існують вебдодатки, які можуть працювати автономно або використовувати інші архітектурні моделі, наприклад "Peer-to-peer".

Web-додатки відрізняються високим рівнем інтерактивності, що досягається обробкою запитів користувача в режимі реального часу. Це дозволяє реалізовувати різноманітні функції – від простого перегляду інформації до складних транзакцій, взаємодії з базами даних та автоматизованої обробки бізнес-процесів.

Вебдодатки можна класифікувати за їх функціональним призначенням і способом взаємодії з користувачем [2]:

– Document-centric web applications – статичні вебсторінки, які оновлюються вручну та надають інформацію у вигляді документів;

– Interactive web applications – дозволяють користувачу взаємодіяти з інтерфейсом через форми, меню та елементи керування;

– Transactional web applications – підтримують зміну даних та виконання транзакцій у базі даних;

– Work-flow based web applications – забезпечують управління робочими процесами між організаціями або підрозділами. Найкращим прикладом таких додатків є рішення для електронної комерції B2B;

– Collaborative web applications – призначені для групової роботи та спільного обміну інформацією, наприклад онлайн-форуми або навчальні платформи;

– Portal-Oriented web applications – надають єдину точку доступу до різних джерел інформації та сервісів, наприклад пошукові системи чи корпоративні портали;

– Ubiquitous web applications – забезпечують доступність сервісів на різних пристроях у будь-який час, часто враховують геолокацію користувача;

– Knowledge-based web applications – вебдодатки, орієнтовані на управління знаннями та надання інформації як для людини, так і для машин (наприклад, систем рекомендацій або пошукових систем), із використанням семантичних вебтехнологій.

Важливо розуміти, що зазначені типи вебдодатків не є взаємовиключними. Один і той же web-додаток може одночасно належати до кількох категорій залежно від свого функціоналу та способу взаємодії з користувачем. Наприклад, навчальна платформа може бути одночасно Collaborative (для спільної роботи студентів та викладачів) та Portal-Oriented (забезпечувати єдину точку доступу до курсів і ресурсів). Аналогічно, B2B-платформа електронної комерції може поєднувати ознаки Transactional (забезпечувати обробку замовлень), Work-flow based (організовувати управління бізнес-процесами) та Interactive (надавати форми, меню та інші динамічні елементи для взаємодії з користувачем) функцій.

Таким чином, наведена класифікація [2] слугує концептуальною схемою для розуміння основних видів web-додатків, але на практиці межі між типами часто розмиті, а реальні проєкти можуть включати ознаки кількох типів

одночасно, забезпечуючи комплексний та багатофункціональний користувацький досвід.

1.2 Архітектура web-додатків

Архітектура web-додатку визначає його внутрішню структуру, способи взаємодії компонентів, обробку запитів користувачів, роботу з базами даних та можливості подальшого масштабування системи. Вона безпосередньо впливає на продуктивність, надійність, безпеку, підтримуваність і здатність web-додатку до розширення. У розробці на PHP та Python застосовуються різні архітектурні підходи, вибір яких здійснюється залежно від розміру та складності проєкту, потреб у масштабованості та продуктивності, характеру бізнес-логіки та функціоналу додатку, вимог до інтерактивності тощо.

Найпоширенішими архітектурними моделями вебдодатків, написаних на PHP та Python, є (табл. 1.1):

– клієнт-серверна;

– монолітна;

– мікросервісна;

– сервісно-орієнтована;

– подієво-орієнтована архітектура.

Клієнт-серверна архітектура (Client-Server) – це модель побудови програмного забезпечення, у якій додаток розділено на дві основні частини: клієнт і сервер, що взаємодіють через комп’ютерну мережу. Клієнт відповідає за ініціацію запитів, тобто надсилає серверу команди на виконання певних дій, наприклад, отримання даних або обробку певних операцій. Сервер, у свою чергу, приймає ці запити, виконує необхідні обчислення або взаємодію з базами даних, а потім повертає результати клієнту [3].

Монолітна архітектура (Monolithic architecture) – це архітектурний підхід, у якому всі компоненти та модулі вебдодатку тісно пов’язані між собою та залежать один від одного в межах єдиного проєкту [4]. При такому підході інтерфейс користувача, бізнес-логіка та доступ до даних інтегровані в єдиний кодовий простір і розгортаються як одне ціле. Це спрощує деплоймент та розробку на початкових етапах проєкту, оскільки всі компоненти додатку знаходяться в одному місці. Водночас така структура обмежує можливості гнучкого масштабування, часткового оновлення та незалежного тестування окремих модулів. Монолітні додатки доводиться масштабувати повністю, а не їх окремі підсистеми, і будь-які зміни в одному модулі можуть вплинути на роботу всього додатки, що ускладнює підтримку великих і складних проєктів.

Мікросервісна архітектура (Microservices) – це архітектурне рішення, яке базується на розподілі модулів на окремі системи, які спілкуються між собою за допомогою повідомлень [4]. Відповідні підсистеми незалежні одна від одної. Кожна з них відповідає за окрему функціональність і може розвиватися, масштабуватися та розгортатися автономно. Така архітектура підвищує гнучкість і надійність системи, дозволяє масштабувати окремі компоненти та застосовувати різні технології для реалізації різних компонентів. Водночас вона ускладнює розробку, роблячи процес відлагодження та розгортання більш трудомістким і часозатратним.

Сервісно-орієнтована архітектура (Service-oriented architecture) – це архітектурний підхід, за якого додаток будується як сукупність окремих сервісів, кожен з яких реалізує певну функціональність. Сервіси взаємодіють між собою за допомогою стандартних протоколів обміну даними, таких як HTTP або SOAP, що забезпечує гнучкість, повторне використання компонентів і високу масштабованість системи [5]. Водночас, на відміну від мікросервісної архітектури, сервіси в SOA зазвичай більші, менш автономні та потребують централізованої координації, що ускладнює розробку та підтримку.

Подієво-орієнтована архітектура (Event-driven architecture) – це підхід, за якого web-додаток реагує на події, наприклад дії користувача або зміни даних (вони є головним тригером для дій системи і керують виконанням логіки). Подібна система побудована як набір сервісів, що взаємодіють через обробку подій [5]. Така архітектура забезпечує високу масштабованість і швидку реакцію на зміни, проте її реалізація досить складна і вимагає глибоких знань у цій сфері. Зазвичай для EDA широко застосовують serverless-технології, де логіка обробки подій виконується у хмарі, що усуває необхідність постійно керувати серверною інфраструктурою і дає змогу динамічно використовувати ресурси лише під час обробки подій.

Таблиця 1.1 – Основні типи архітектур веб-додатків на PHP та Python

Архітек-тура	**Ідея**	**Переваги**	**Недоліки**	**Техно-логії (PHP)**	**Техно-логії (Python)**
1	2	3	4	5	6
Клієнт-серверна	Поділ на клієнтську та серверну частини	Простота реалізації, централізоване управління, масштабованість	Проблеми продуктивності та потреба в потужному обладнанні при великій кількості користувачів, залежність від сервера	PHP-FPM, Laravel, Symfony	Django, Flask, FastAPI, Tornado
Монолітна	Єдиний кодовий простір для користувацького інтер-фейсу.	Простий деплоймент, зручність тестування та налагодження, відносно низька вартість.	Складність масштабування та впровадження нових технологій через вплив змін у коді на весь додаток.	Laravel, Symfony, Yii2	Django, Flask, Pyramid

Продовження табл. 1.1

1	2	3	4	5	6
Клієнт-серверна	Поділ на клієнтську та серверну частини	Простота реалізації, централізоване управління, масштабованість	Проблеми продуктивності та потреба в потужному обладнанні при великій кількості користувачів, залежність від сервера	PHP-FPM, Laravel, Symfony	Django, Flask, FastAPI, Tornado
Монолітна	Єдиний кодовий простір для користувацького інтерфейсу, бізнес-логіки та обробки даних	Простий деплоймент, зручність тестування та налагодження, відносно низька вартість інфраструктури	Складність масштабування та впровадження нових технологій, бо будь-які зміни в коді впливають на роботу всього додатку	Laravel, Symfony, Yii2	Django, Flask, Pyramid
Мікросервісна	Поділ системи на незалежні автономні компоненти	Технологічна гнучкість, незалежність функцій, масштабованість окремих компонентів, загальна відмовостійкість	Складність тестування та налагодження, значні мережеві витрати, необхідність потужної автоматизації	Laravel (API), Symfony	FastAPI, Flask, Nameko, Django REST Framework

Кінець табл. 1.1

1	2	3	4	5	6
Сервісно-орієнтована	Поділ на централізовано керовані сервіси, які взаємодіють між собою через стандартизовані протоколи	Незалежність технологій, легка інтеграція та масштабованість, повторне використання коду, адаптивність до змін	Складність координації та часозатратність тестування, потенційний вплив змін в одному сервісі на інші	Symfony, Laravel	Flask, Spyne, Django REST Framework, Zeep
Подієво-орієнтована	Обробка подій як основний тригер дій системи	Ефективне використання ресурсів, асинхронна обробка, швидка реакція на події, легке масштабування, розподілена надійність	Складність реалізації, громіздке тестування, необхідність додаткової обробки помилок та відмов, потреба у глибоких знаннях архітектури	Laravel Queues, Symfony Messenger	Celery, Apache Kafka, RabbitMQ, Redis Streams

Отже, архітектура вебдодатків визначає не лише внутрішню структуру системи та способи взаємодії її компонентів, а й продуктивність, масштабованість, стійкість до навантажень, безпеку та зручність подальшої підтримки. Від обраної архітектури залежить, наскільки ефективно система обробляє запити користувачів, реагує на події, інтегрує сторонні сервіси та використовує ресурси сервера. Розуміння основних архітектурних моделей дозволяє приймати обґрунтовані рішення щодо визначення структури додатку та ефективного використання серверних ресурсів.

1.3 База даних MySQL та методи доступу до неї у web-додатках

Бази даних є невід'ємною частиною сучасних web-додатків. Саме вони відповідають за збереження, обробку та швидкий доступ до структурованих даних, забезпечуючи коректну роботу бізнес-логіки додатку, збереження стану системи та взаємодію між різними її компонентами навіть при великій кількості користувачів.

MySQL – це відкрита система керування реляційними базами даних, яка використовується для зберігання, обробки та доступу до структурованих даних. У ній дані зберігаються в таблицях, пов'язаних між собою, і для роботи з інформацією використовується стандартна мова запитів SQL [6]. MySQL відома своєю продуктивністю, надійністю та широким використанням у вебпроєктах різного масштабу.

1.3.1 Методи доступу до бази даних MySQL у web-додатках, створених на PHP

У web-додатках, розроблених мовою PHP, доступ до бази даних MySQL здійснюється трьома основними способами:

– за допомогою розширення MySQLi (MySQL Improved) [7];

– з використанням об'єктно-реляційних відображень (ORM);

– із використанням PHP Data Objects (PDO) [8].

Базовим методом доступу до бази даних MySQL у вебдодатках, розроблених на PHP, є розширення MySQLi (MySQL Improved). Воно є сучасним і водночас досить простим інструментом, що дає розробникам змогу працювати з базою даних як у процедурному, так і в об'єктно-орієнтованому стилі. MySQLi підтримує виконання підготовлених запитів (prepared

statements), що значно підвищує безпеку вебдодатку, оскільки мінімізує ризик SQL-ін’єкцій. Крім того, розширення забезпечує роботу з транзакціями, мультизапитами та асоціативними масивами результатів, що надає можливість ефективно обробляти великі обсяги даних і підвищує продуктивність системи. Основними перевагами такого підходу є повний контроль над SQL-запитами, підтримка підготовлених виразів і транзакцій, можливість роботи з мультизапитами та висока сумісність із сучасними версіями MySQL. До недоліків можна віднести відсутність автоматичного об’єктно-реляційного відображення даних, що досить незручно при роботі з великими моделями даних. Також мінусами такого методу є необхідність ручного написання більшості SQL-запитів та обробки результатів, що робить підтримку великих проєктів більш складною.

Нижче наведено приклад підключення до MySQL за допомогою MySQLi (рис. 1.1).

```php
<?php

$mysqli = new mysqli("localhost", "user", "password", "database");

echo $mysqli->host_info . "\n";

$mysqli = new mysqli("127.0.0.1", "user", "password", "database", 3306);

echo $mysqli->host_info . "\n";
```

Рисунок 1.1 – Приклад підключення до MySQL за допомогою MySQL Improved [9]

Наступним методом доступу до бази даних у вебдодатках PHP є використання об’єктно-реляційних відображень (ORM). ORM дозволяє працювати з базою даних через об’єкти та класи, автоматично відображаючи таблиці бази даних на відповідні об’єкти у коді. Подібний підхід значно спрощує взаємодію з даними, оскільки розробник може виконувати операції створення, читання, оновлення та видалення (CRUD) без необхідності написання повних SQL-запитів.

Перевагами цього підходу є підвищена зручність розробки, зменшення кількості потенційних помилок у SQL-запитах, покращена підтримка великих і складних моделей даних, можливість легко змінювати структуру бази даних без значних змін у коді, а також підвищення безпеки вебдодатку завдяки автоматичній обробці параметрів запитів. До недоліків цього підходу можна віднести потенційне зниження продуктивності у порівнянні із використанням прямих (стандартних) SQL-запитів, особливо при роботі з великими обсягами даних, а також додатковий шар абстракції, який іноді ускладнює процес оптимізації вебдодатку.

Серед найбільш поширених ORM-фреймворків для PHP слід виділити Eloquent [10] (є частиною фреймворку Laravel [11]), Doctrine [12] та Propel [13]. Ці фреймворки дозволяють ефективно застосовувати ORM у вебдодатках, написаних на PHP, підвищуючи продуктивність розробки та зменшуючи ризик помилок при роботі з базою даних.

Приклад підключення вебдодатку до MySQL за допомогою Doctrine ORM наведено на рис. 1.2.

```php
<?php
// bootstrap.php
require_once "vendor/autoload.php";

use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\ORMSetup;

$paths = ['/path/to/entity-files'];
$isDevMode = false;

// the connection configuration
$dbParams = [
    'driver'   => 'pdo_mysql',
    'user'     => 'root',
    'password' => '',
    'dbname'   => 'foo',
];

$config = ORMSetup::createAttributeMetadataConfig($paths, $isDevMode);
// on PHP < 8.4, use ORMSetup::createAttributeMetadataConfiguration() instead
$connection = DriverManager::getConnection($dbParams, $config);
$entityManager = new EntityManager($connection, $config);
```

Рисунок 1.2 – Приклад підключення до MySQL за допомогою Doctrine ORM [14]

Ще одним популярним методом доступу до бази даних MySQL у вебдодатках, створених на PHP, є використання PHP Data Objects (PDO). PDO представляє собою уніфікований інтерфейс доступу до баз даних у PHP, який дає змогу працювати з різними системами керування базами даних (СКБД), а не лише з MySQL, без необхідності значних змін у коді при зміні самої СКБД.

Цей метод доступу також підтримує виконання підготовлених запитів, що підвищує безпеку вебдодатків, мінімізуючи ризик SQL-ін’єкцій, а також дозволяє працювати з транзакціями та обробляти результати запитів у вигляді асоціативних або об’єктних масивів. Завдяки цим можливостям PDO забезпечує ефективну роботу з базами даних, полегшує обробку великих обсягів інформації, робить код більш читабельним та підтримуваним. Перевагами цього підходу є універсальність роботи з різними СКБД, підвищена безпека запитів, зручна обробка помилок і можливість використання транзакцій, тоді як до недоліків можна віднести необхідність написання складних SQL-запитів вручну, потенційне зниження

продуктивності у випадках, коли один і той самий запит виконується дуже часто без оптимізації або кешування, складність налаштування обробки помилок та транзакцій у великих проєктах, особливо при роботі з кількома підключеннями до бази даних одночасно.

Далі наведено приклад підключення до MySQL із використанням PDO (рис. 1.3).

```php
$host="localhost";
$port=3306;
$socket="";
$user="root";
$password="";
$dbname="sakila";

try {
    $dbh = new PDO("mysql:host={$host};port={$port};dbname={$dbname}", $user, $password));
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}
```

Рисунок 1.3 – Приклад підключення до MySQL із використанням PDO [15]

Хоча MySQLi, ORM та PDO є найпоширенішими методами доступу до MySQL у вебдодатках, створених на PHP, існують і інші способи підключення. Вони використовуються рідше через складність налаштування, обмежену сумісність, специфічні вимоги тощо. Отже, розглянуті підходи є основними при підключенні PHP вебдодатку до СКБД MySQL і забезпечують високу продуктивність, безпеку та зручність розробки. Саме тому їх найчастіше застосовують на практиці.

1.3.2 Методи доступу до бази даних MySQL у web-додатках, створених на Python

Одним із базових способів доступу до бази даних MySQL у Python web-додатках є пряме підключення через спеціалізовані драйвери, що реалізують

стандарт DB-API [16]. До них належать, зокрема, mysql.connector [17], PyMySQL [18], mysqlclient [19] – всі вони функціонують за однаковим принципом.

Найпопулярнішим офіційним драйвером MySQL для Python є mysql.connector. Він дозволяє встановлювати з'єднання з сервером БД, виконувати SQL-запити та обробляти результати без додаткових рівнів абстракції. Драйвер підтримує транзакції, параметризовані запити та базові механізми обробки помилок, що робить його придатним для вебдодатків малого та середнього масштабу. Пряме підключення через драйвер передбачає ручне керування з'єднанням, формування SQL-запитів і контроль коректності обробки даних, покладаючи відповідальність за безпеку на розробника. Переваги з точки зору безпеки полягають у можливості використання параметризованих запитів для захисту від SQL-ін'єкцій та повному контролі над автентифікацією й доступом до даних. Недоліками є високий ризик помилок при некоректній обробці введення користувача та відсутність автоматичного рівня захисту, який надають ORM-фреймворки.

Приклади простого підключення до MySQL через mysql.connector та приклад підключення з обробкою помилок наведено на рис. 1.4 та 1.5 відповідно:

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', password='password',
                              host='127.0.0.1',
                              database='employees')
cnx.close()
```

Рисунок 1.4 – Просте підключення до MySQL за допомогою mysql.connector [20]

```
import mysql.connector
from mysql.connector import errorcode

try:
  cnx = mysql.connector.connect(user='scott',
                                database='employ')
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
    print("Something is wrong with your user name or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exist")
  else:
    print(err)
else:
  cnx.close()
```

Рисунок 1.5 – Підключення до MySQL із обробкою помилок за допомогою mysql.connector [20]

Ще одним поширеним способом доступу до MySQL у Python web-додатках є використання Object-relational mapping (ORM) – фреймворків, які дозволяють працювати з базою даних через об’єкти Python замість написання сирих SQL-запитів [21]. ORM автоматично формує SQL-запити на основі операцій з об’єктами, керує транзакціями, забезпечує безпечну обробку даних, підтримує зв’язки між таблицями та абстрагує роботу з низькорівневими деталями з’єднання. Використання ORM спрощує інтеграцію бізнес-логіки з базою даних, покращує читабельність коду, зменшує ризик помилок і SQL-ін’єкцій та дозволяє розробникам зосередитися на логіці додатку, а не на тонкощах SQL. Переваги ORM полягають у безпечній обробці даних, автоматичному формуванні запитів та спрощеному тестуванні й підтримці коду, тоді як потенційними недоліками є зниження продуктивності при складних запитах, обмежена гнучкість у специфічних SQL-операціях і додатковий шар абстракції, що іноді ускладнює оптимізацію.

Найбільш популярними ORM-фреймворками для роботи з MySQL у Python є SQLAlchemy [22], Django ORM [23] та Peewee [24]. Вони дозволяють розробникам працювати з базою даних через об’єкти Python, автоматично формують SQL-запити, керують транзакціями та забезпечують безпечну обробку даних, спрощуючи інтеграцію бізнес-логіки та підтримку коду.

Приклад підключення до MySQL за допомогою SQLAlchemy наведено на рис. 1.6.

```python
from sqlalchemy import create_engine, text

DATABASE_URL = "mysql+mysqlclient://user:password@localhost/db_name"

engine = create_engine(DATABASE_URL)

try:
    with engine.connect() as connection:
        result = connection.execute(text("SELECT 1"))
        print("Підключення до MySQL успішне:", result.scalar())
except Exception as e:
    print(f"Помилка підключення: {e}")

```

Рисунок 1.6 – Приклад підключення до MySQL за допомогою SQLAlchemy

Ще одним підходом є використання вебфреймворків із вбудованою підтримкою доступу до бази даних, таких як Flask [25], Django [26], FastAPI [27] з плагінами на кшталт Flask-SQLAlchemy [28], Django ORM [23] або SQLModel [29] для FastAPI. Вони інтегрують роботу з базою даних безпосередньо у структуру додатку та спрощують управління моделями, підключеннями і транзакціями. На відміну від випадків використання ORM окремо від фреймворку, де підключення та налаштування бази даних виконуються вручну, у цих фреймворках доступ до MySQL інтегрований у проєкт: конфігурація бази задається у файлах проєкту, а робота з таблицями здійснюється через моделі Python. Це дозволяє автоматично керувати підключеннями, транзакціями та зв’язками між таблицями, спрощує обробку помилок і забезпечує більш безпечну взаємодію з даними, мінімізуючи ризик SQL-ін’єкцій. Крім того, такий підхід покращує підтримку та тестування коду, оскільки доступ до бази даних організовано централізовано та узгоджено з архітектурою вебдодатку. Недоліками підходу є певна залежність від конкретного фреймворку, додатковий рівень абстракції та можливе зниження продуктивності при складних запитах.

Приклад підключення до MySQL у Flask з використанням Flask-SQLAlchemy наведено на рис. 1.7.

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+mysqlclient://user:password@localhost/db_name'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
# Модель таблиці
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), nullable=False, unique=True)

with app.app_context():
    db.create_all()

@app.route('/')
def index():
    try:
        result = db.session.execute('SELECT 1').scalar()
        return f'Підключення успішне: {result}'
    except Exception as e:
        return f'Помилка підключення: {e}'

if __name__ == '__main__':
    app.run(debug=True)
```

Рисунок 1.7 – Приклад підключення до MySQL у Flask з використанням Flask-SQLAlchemy

У сучасних вебдодатках також використовують асинхронний доступ до MySQL, який дозволяє їм обробляти запити неблокуючим способом та підвищує продуктивність при великій кількості одночасних користувачів. Для цього використовуються асинхронні драйвери, такі як aiomysql [30], та ORM з підтримкою async/await, наприклад Tortoise ORM [31] або SQLAlchemy Async [32]. Вони дозволяють виконувати запити та керувати транзакціями асинхронно, автоматично формуючи SQL-запити та працюючи з моделями Python. Асинхронна робота забезпечує швидшу обробку паралельних запитів і ефективніше використання ресурсів, але потребує сумісних бібліотек та більш складної реалізації порівняно з синхронними підходами.

На рис. 1.8 наведено приклад асинхронного підключення до MySQL у Python за допомогою бібліотеки aiomysql.

```
import asyncio
import aiomysql

loop = asyncio.get_event_loop()

@asyncio.coroutine
def test_example():
    conn = yield from aiomysql.connect(host='127.0.0.1', port=3306,
                                       user='root', password='', db='mysql',
                                       loop=loop)

    cur = yield from conn.cursor()
    yield from cur.execute("SELECT Host,User FROM user")
    print(cur.description)
    r = yield from cur.fetchall()
    print(r)
    yield from cur.close()
    conn.close()

loop.run_until_complete(test_example())
```

Рисунок 1.8 – Приклад підключення до MySQL з використанням aiomysql [30]

Отже, у більшості випадків для доступу до MySQL web-додатках, написаних на Python, використовується один із наведених вище способів – пряме підключення через драйвери, робота через ORM або інтеграція у вебфреймворк із підтримкою бази даних. Окрім того, у сучасних вебдодатках часто застосовується асинхронний доступ до MySQL. Існують також й інші методи, проте вони менш популярні та застосовуються рідше через складність реалізації, обмежену підтримку та меншу сумісність із сучасними вебфреймворками.

1.4 Безпека web-додатків при роботі з базами даних MySQL

Безпека web-додатків є однією з ключових складових успішного функціонування сучасних інформаційних систем. Особливо це стосується роботи з базами даних (зокрема у MySQL), які містять конфіденційну інформацію про користувачів, транзакції, налаштування системи та інші критично важливі дані. Недотримання принципів безпеки може призвести до

витоку даних, порушення цілісності бази та несанкціонованого доступу до системи.

1.4.1 Основні види атак на бази даних MySQL у web-додатках

Бази даних MySQL у вебдодатках зберігають величезні обсяги інформації, у тому числі персональні дані користувачів і комерційні записи, що робить їх привабливим об'єктом для кіберзлочинців. Одним із найпоширеніших і найбільш небезпечних типів атак на БД є SQL Injection – метод, при якому зловмисник вводить шкідливі SQL-запити у поля вводу вебдодатку, щоб отримати несанкціонований доступ до конфіденційної інформації або змінити дані у базі. Основною причиною виникнення SQL-ін'єкцій є використання динамічно сформованих SQL-запитів без належної перевірки та валідації користувацького вводу. Саме тому SQL-ін'єкції вважаються одним із найбільш небезпечних типів атак на бази даних у веб-додатках [33].

Не менш критичною є атака, пов'язана з порушенням контролю доступу (Broken access control). Вона полягає у наданні зловмиснику можливості отримати доступ до ресурсів або даних без належних прав через неправильно налаштовані механізми контролю доступу. У такому випадку зловмисник може отримати доступ до даних, які йому не призначені, у тому числі й особисті профілі інших користувачів або адміністративні таблиці бази даних [34].

Серйозну загрозу становлять криптографічні вразливості. Використання слабких алгоритмів шифрування для збереження особистих даних користувачів (або їх відсутність) може призвести до того, що конфіденційна інформація опиниться в руках зловмисників, що загрожує компрометацією облікових записів та витоком персональних даних [34].

Суттєву загрозу становить використання застарілих або вразливих компонентів програмного забезпечення. Незахищені бібліотеки, фреймворки чи СКБД (або їх застарілі версії) можуть стати точкою входу для атак, що потенційно дозволять зловмисникам обійти захисні механізми, отримати несанкціонований доступ до даних або викликати відмову у роботі бази даних.

Серед більш нових видів атак, що набирають популярність, варто виділити Server-side request forgery (SSRF) – підхід, коли зловмисник змушує сервер виконувати запити до внутрішніх ресурсів мережі. Така атака може використовуватися для доступу до бази даних, яка прихована від зовнішніх користувачів, а також для отримання інформації про внутрішню структуру мережі та інші критично важливі системи [34].

Усі перелічені типи атак мають прямий вплив на цілісність, конфіденційність та доступність даних у базах, тому їх своєчасне виявлення та запобігання є ключовими завданнями розробників та адміністраторів web-додатків.

1.4.2 Методи забезпечення безпеки web-додатків при роботі з MySQL

Захист web-додатків при роботі з базами даних MySQL передбачає комплекс заходів, спрямованих на мінімізацію ризиків, пов’язаних із можливими атаками.

Одним із найефективніших методів захисту є використання параметризованих запитів (prepared and parameterized queries), що дозволяє відокремити SQL-код від користувацького вводу та запобігти виконанню потенційно шкідливих команд. Разом з тим, важливим заходом є валідація та санітизація даних, що передбачає перевірку формату та типу введеної інформації і видалення потенційно небезпечних символів [35].

Важливу роль у забезпеченні безпеки відіграє коректна реалізація механізмів автентифікації та контролю доступу. Застосування принципу "найменших привілеїв" передбачає надання користувачам лише тих прав доступу, які необхідні для виконання їхніх функцій, що значно знижує ризик несанкціонованого доступу до критичних таблиць або адміністративних ресурсів бази даних.

Не менш важливим є застосування шифрування даних як під час передачі (TLS/SSL для з’єднання з MySQL), так і при зберіганні (шифрування паролів за допомогою алгоритмів bcrypt або Argon2). Такий підхід дає змогу забезпечити захист конфіденційної інформації навіть у разі витоку бази даних.

Для підвищення загального рівня захищеності вебдодатків також доцільно впроваджувати багатофакторну автентифікацію та додаткові механізми контролю доступу, що ускладнюють компрометацію облікових записів навіть за умови викрадення пароля.

Окрему увагу слід приділяти регулярному оновленню самої СКБД, бібліотек і фреймворків web-додатку. Своєчасне встановлення оновлень безпеки дозволяє усувати відомі вразливості та зменшувати ризик атак, пов’язаних із використанням застарілих компонентів ПЗ. Додатковим рівнем захисту може слугувати впровадження механізмів логування та моніторингу запитів до БД, що дасть змогу виявляти підозрілу активність і своєчасно реагувати на потенційні загрози.

Комплексне застосування зазначених методів дозволяє значно підвищити рівень безпеки вебдодатків, забезпечуючи захист конфіденційності, цілісності та доступності даних у базах даних MySQL.

1.5 Огляд інструментів аналізу продуктивності

Аналіз продуктивності web-додатків є важливою складовою забезпечення їх ефективної роботи та надійності. У загальному сенсі, продуктивність вебдодатку визначається швидкістю обробки запитів, часом відповіді на дії користувача, здатністю витримувати одночасне навантаження та ефективністю використання ресурсів сервера.

Оскільки більшість сучасних вебдодатків працюють з динамічними даними, повільні або неоптимальні SQL-запити можуть створювати затримки у завантаженні сторінок, збільшувати час обробки дій користувача та підвищувати навантаження на сервер, що безпосередньо впливаватиме на загальну продуктивність системи. Наприклад, великі або неоптимальні вибірки даних можуть збільшувати навантаження на сервер і призводити до затримок у відповіді вебдодатку, значно збільшуючи час очікування користувача.

Оцінка продуктивності вебдодатків здійснюється з урахуванням мови програмування та технологій, на яких побудовано сам додаток, оскільки інструменти та методи вимірювання можуть відрізнятися. У подальших підрозділах розглянуто основні підходи та інструменти оцінки продуктивності для web-додатків, створених на мовах PHP та Python, при роботі з базами даних.

1.5.1 Методи та інструменти аналізу продуктивності web-додатків, створених на PHP, при роботі з базою даних MySQL

Для аналізу продуктивності вебдодатків, розроблених на PHP, при взаємодії з СКБД MySQL застосовують комплексний підхід, який включає

оцінку часу виконання серверних скриптів, ефективності SQL-запитів, навантаження на сервер та використання ресурсів системи. Основною метою такого аналізу є виявлення "вузьких місць" (частин системи, які обмежують її загальну продуктивність і швидкодію) у коді та запитах до БД, що дозволяє оптимізувати вебдодаток для забезпечення стабільної та швидкої роботи під високим навантаженням.

Одним із ключових методів аналізу є профілювання PHP-коду – процес вимірювання часу виконання функцій, використання пам’яті та системних ресурсів. Для цього використовуються такі інструменти:

- Xdebug – розширення, яке дозволяє трасувати виконання функцій і методів скриптів, вимірювати час виконання окремих функцій, обсяг використаної пам’яті та створювати детальні профілі продуктивності [36];
- Blackfire – інструмент для глибокого профілювання PHP-додатків, який дозволяє порівнювати різні версії коду та виявляти їх найповільніші ділянки [37];
- Tideways – сервіс, що забезпечує збір метрик продуктивності, моніторінг запитів до бази даних і інтегрування з CI/CD для автоматичного моніторингу [38].

Для оцінки ефективності взаємодії вебдодатку з БД застосовують спеціальні бібліотеки та розширення PHP, які дозволяють контролювати виконання SQL-запитів та відстежувати їх час виконання та загальну продуктивність:

- PDO – розширення для роботи з БД, яке надає інструменти для контролю часу виконання SQL-запитів при роботі з різними СКБД, включно з MySQL [8];
- MySQLi – альтернативне розширення для роботи з MySQL, що дозволяє не тільки виконувати підготовлені запити, але й відстежувати помилки та збирати статистику виконання запитів [7];
- Doctrine DBAL – бібліотека для роботи з базами даних, яка надає функції логування запитів, аналізу їх часу виконання та виявлення

повторюваних запитів (N+1 problem), що часто є причиною зниження продуктивності додатку [39];

˜ Laravel Debugbar – розширення для фреймворку Laravel, яке відображає панель профілювання у браузері, показує кількість SQL-запитів, час їх виконання та структуру, дозволяючи швидко виявляти проблемні ділянки коду [40].

Застосування вищезазначених методів і інструментів забезпечує комплексну оцінку продуктивності вебдодатку, створеного на PHP, дозволяє виявляти критичні точки, оптимізувати "вузькі місця" та гарантувати стабільну і швидку роботу системи під різним навантаженням.

1.5.2 Методи та інструменти аналізу продуктивності web-додатків, створених на Python, при роботі з базою даних MySQL

При аналізі продуктивності створених на Python вебдодатків із доступом до MySQL або інших баз даних, використовують комплексний підхід, що поєднує логування виконання функцій, профілювання коду, моніторинг пам’яті і системних ресурсів, а також оцінку ефективності роботи з БД. Такий підхід дозволяє зробити процес аналізу більш вдосконалим та ефективним.

Як і при роботі з PHP-додатками, для оцінки продуктивності вебдодатків, розроблених мовою Python, використовується профілювання. Найбільш поширеними інструментами для цього є:

˜ cProfile – стандартний Python-профайлер, який дозволяє збирати статистику викликів функцій і методів, вимірювати час їх виконання та створювати детальні профілі продуктивності [41];

˜ line_profiler – інструмент для профілювання на рівні окремих рядків коду, що дозволяє виявляти найповільніші ділянки скриптів [42];

˜ memory_profiler – програмна бібліотека для моніторингу використання пам’яті вебдодатком, що сприяє виявленню неефективних операцій та потенційних витоків пам’яті [43].

Для безпосереднього контролю продуктивності вебдодатку при доступі до бази даних використовуються наступні інструменти:

˜ логування SQL-запитів та часу їх виконання – реалізується засобами стандартного модуля logging [44] або можливостями ORM (наприклад, SQLAlchemy), що дозволяє відстежувати кількість запитів, їх структуру та тривалість виконання під час обробки HTTP-запитів;

˜ вимірювання часу виконання функцій і методів – здійснюється за допомогою модулів time [45] або timeit [46] і потрібне для оцінки затрат часу на виконання окремих операцій доступу до даних;

˜ аналіз ефективності ORM-запитів – проводиться шляхом порівняння кількості та складності SQL-запитів, згенерованих ORM (наприклад, SQLAlchemy), що дозволяє виявляти надмірні або повторювані звернення до бази даних;

˜ моніторинг використання системних ресурсів – проводиться за допомогою Python-бібліотек psutil [47] або memory_profiler [43]. Вони дозволяють відстежувати використання оперативної пам’яті та процесорного часу вебдодатком під час роботи з базою даних;

˜ Load-тестування (тестування під навантаженням) – виконується за допомогою потужних інструментів, таких як Locust [48], pytest-benchmark [49] та aiohttp-bench [50]. Вони дозволяють імітувати одночасну роботу великої кількості користувачів, вимірювати час відповіді вебдодатку, інтенсивність запитів до бази даних та оцінювати стабільність системи під навантаженням;

˜ моніторинг помилок і затримок виконання – здійснюється з використанням інструментів логування та збору виключень, зокрема інструмента Sentry SDK for Python [51], який є одним із найбільш потужних та поширених у наш час. Він забезпечує автоматичне відстеження помилок,

винятків і повільних транзакцій, що виникають під час виконання операцій доступу до БД.

Таким чином, для оцінки продуктивності вебдодатків, створених на Python, при доступі до бази даних застосовується комплексний набір інструментів та методів. Вони дозволяють відстежувати час виконання функцій, ефективність ORM-запитів, використання пам’яті та системних ресурсів, а також проводити навантажувальні тести і моніторинг помилок. Комплексне використання розглянутих підходів забезпечує всебічну оцінку продуктивності, допомагає виявляти неефективні ділянки коду та запитів до БД і сприяє підвищенню стабільності та швидкодії вебдодатку.

1.6 Висновки до розділу

У цьому розділі було досліджено теоретичні основи web-додатків, їх класифікацію та специфіку взаємодії користувача із сервером. Показано, що вебдодатки відрізняються високим рівнем інтерактивності та здатністю виконувати складні операції з даними, надаючи користувачам динамічний та зручний інтерфейс.

Крім того, було розглянуто архітектурні моделі вебдодатків, зокрема монолітну, мікросервісну, подієво-орієнтовану та сервісно-орієнтовану. Встановлено, що обрана архітектура безпосередньо впливає на продуктивність, масштабованість, надійність та підтримуваність системи, а також визначає ефективність обробки запитів користувачів і використання ресурсів сервера.

Також було проаналізовано методи доступу до баз даних MySQL у вебдодатках, створених на PHP та Python. З’ясовано, що пряме підключення через драйвери, використання ORM та інтеграція у вебфреймворки дозволяють ефективно управляти даними, автоматизувати формування SQL-

запитів та зменшувати ризик помилок, забезпечуючи зручність розробки та підтримки коду.

Особливу увагу приділено питанням безпеки при роботі з базами даних. Зроблено висновок, що використання параметризованих запитів, валідація даних, контроль доступу, шифрування та регулярне оновлення компонентів системи дозволяють мінімізувати ризики SQL-ін’єкцій, несанкціонованого доступу та інших атак, забезпечуючи цілісність, конфіденційність та доступність даних.

Було визначено, що для проведення дослідження продуктивності вебдодатків необхідно використовувати спеціалізовані інструменти, такі як профайлер коду, системи логування та тестування навантаження. У розділі було розглянуто найпопулярніші з них для web-додатків, створених на PHP та Python. Такі інструменти дозволяють виявляти "вузькі місця" у роботі з базою даних та оптимізувати обробку запитів.

Загалом, аналіз показав, що успішна розробка вебдодатків потребує комплексного підходу, який поєднує правильний вибір архітектури, оптимальні методи доступу до баз даних і ефективні заходи безпеки. Такий підхід забезпечуватиме стабільну, надійну та продуктивну роботу вебсистеми під різним навантаженням.

2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ WEB-ДОДАТКУ НА PYTHON З ВИКОРИСТАННЯМ MYSQL

2.1 Вимоги до web-додатку

Web-додаток повинен забезпечувати можливість проведення аналізу його продуктивності при доступі до інформації з бази даних MySQL. Додаток має реалізовувати стандартну клієнт-серверну модель взаємодії та підтримувати обробку HTTP-запитів з боку користувача з подальшим зверненням до бази даних.

Функціональні вимоги передбачають реалізацію операцій читання, додавання, оновлення та видалення даних (CRUD), що дозволить оцінити швидкодію виконання запитів до MySQL за різних умов навантаження. Додаток повинен підтримувати автентифікацію та авторизацію користувачів і розмежування доступу до даних з метою перевірки впливу механізмів безпеки на продуктивність системи.

Для забезпечення коректного тестування безпеки web-додаток має використовувати захищені методи доступу до бази даних, зокрема параметризовані SQL-запити або ORM-технології, а також механізми захисту від SQL-ін'єкцій і несанкціонованого доступу. Обробка помилок повинна здійснюватися без розкриття внутрішньої структури бази даних чи серверної логіки.

Нефункціональні вимоги включають можливість вимірювання часу відповіді сервера, кількості запитів за одиницю часу та споживання системних ресурсів. Архітектура додатку повинна бути достатньо гнучкою для проведення експериментів з різними методами доступу до MySQL і налаштуваннями безпеки без суттєвих змін програмного коду.

Крім того, вебдодаток має бути розгорнутий у контрольованому середовищі, що дозволить повторювати експерименти та порівнювати

отримані результати, забезпечуючи об’єктивність аналізу продуктивності та безпеки.

2.2 Вибір технологій та обґрунтування архітектурних рішень

Для реалізації досліджуваного web-додатку було обрано мову програмування Python, оскільки на сьогоднішній день вона є однією з найпоширеніших мов у сфері веброзробки та обробки даних. Python вирізняється простотою синтаксису, високою читабельністю коду та широкою підтримкою сучасних фреймворків. Використання Python дозволяє швидко реалізовувати бізнес-логіку додатку, забезпечувати зручну інтеграцію з системами керування базами даних та проводити детальний аналіз продуктивності виконання SQL-запитів. Крім того, наявність великої кількості бібліотек для захисту від SQL-ін’єкцій та інших типів загроз, дає можливість порівнювати ефективність "сирих" та захищених запитів до бази даних MySQL, що є ключовим аспектом даного дослідження.

Архітектура web-додатку буде побудована за типовою клієнт–серверною моделлю, яка є стандартною для більшості сучасних вебсистем. Для реалізації клієнтської частини додатку використовуватиметься базовий набір вебтехнологій: HTML, CSS та JavaScript, що забезпечить створення простого, але зручного, інтуїтивно зрозумілого та адаптивного користувацького інтерфейсу, ефективну обробку подій із використанням стандартних вебмеханізмів, а також надійну й контрольовану взаємодію з серверною частиною системи.

Серверна логіка web-додатку буде реалізована за допомогою інструментів фреймворку Flask, що зумовлено його мінімалістичною, але досить ефективною, архітектурою, свободою організації коду без жорстко заданих структур та можливістю повного контролю над обробкою HTTP-

запитів. Це дозволить гнучко реалізувати механізми взаємодії між клієнтською та серверною частинами системи, а також забезпечити прямий доступ до засобів роботи з базою даних. Обраний підхід надасть можливість свідомо використовувати як необроблені (сирі) SQL-запити, так і захищені запити (prepared statements, ORM-механізми тощо), що є необхідним для проведення порівняльного аналізу їх продуктивності при доступі до інформації бази даних MySQL.

Таким чином, обране архітектурне та технологічне рішення відповідає цілям дослідження, оскільки воно дозволить оцінити вплив різних способів доступу до бази даних MySQL на продуктивність вебдодатку.

2.3 Проєктування та реалізація web-додатку

В межах дослідження було прийняте рішення розробити web-додаток поштової служби, призначений для автоматизації роботи операторів відділень. Додаток дозволяє обробляти запити користувачів, переглядати інформацію про посилки, відділення, кур’єрів, маршрути доставки, клієнтів та операторів, перевіряти статус відправлень, а також виконувати авторизацію та автентифікацію користувачів (операторів та адміністраторів) для забезпечення безпечного доступу до системи. Оператори (як користувачі) можуть переглядати власний профіль, отримувати актуальні дані про посилки та клієнтів, додавати нові записи у БД, а адміністратори – повністю керувати відділеннями, операторами, кур’єрами, маршрутами та іншими сутностями, здійснюючи повний контроль за роботою поштової служби.

Для реалізації досліджуваного web-додатку було спроектовано клієнтську та серверну частини з урахуванням ролей користувачів додатку, а саме операторів почтового відділення і адміністраторів. Клієнтська частина забезпечує відображення даних, форми для введення та редагування

інформації, а також інтерфейс навігації та відображення статистики. Серверна логіка відповідає за обробку запитів, управління сесіями користувачів, перевірку прав доступу та взаємодію з базою даних.

Спроєктована система підтримує різні рівні доступу: адміністратори мають повний доступ до всіх сутностей системи, а оператори – обмежений доступ до даних свого відділення та деяких операцій з посилками. Для зберігання облікових даних користувачів заздалегідь передбачено захищені методи роботи з паролями за допомогою werkzeug.security (алгоритм pbkdf2:sha256) [52], що не тільки гарантуватимуть безпеку інформації, але й дозволять провести аналіз продуктивності запитів по авторизації та автентифікації користувачів у реалістичних умовах.

2.3.1 Проєктування структури бази даних MySQL

Проєктування структури бази даних є одним із ключових етапів розробки web-додатку, оскільки від коректності логічної та фізичної моделі даних безпосередньо залежать ефективність зберігання інформації, продуктивність виконання запитів, цілісність даних і масштабованість системи в цілому. Грамотно спроєктована база даних забезпечує можливість реалізації складних вибірок, мінімізує надлишковість інформації та спрощує супровід програмного продукту. У цьому проєкті база даних реалізована у СКБД MySQL.

На початковому етапі було виконано логічне проєктування БД, під час якого було визначено основні сутності предметної області, їх атрибути та зв’язки між ними. У процесі логічного проєктування виділено основні сутності предметної області: адміністратори та оператори (користувачі системи), поштові відділення, клієнти та кур’єри, а також маршрути і посилки.

Для наочного відображення структури бази даних було побудовано Entity-relationship diagram (ERD), яка дозволяє візуально представити модель даних у вигляді сутностей з їх атрибутами, а також зв’язків між таблицями, реалізованих за допомогою первинних і зовнішніх ключів.

На рис. 2.1 представлено ERD бази даних проєкту, яка відображає всі основні таблиці, їх атрибути та взаємозв’язки. Усі сутності мають чітко визначені первинні ключі, які відповідають за ідентифікацію записів, а також зовнішні ключі, що реалізують зв’язки між таблицями та гарантують посилальну цілісність даних.

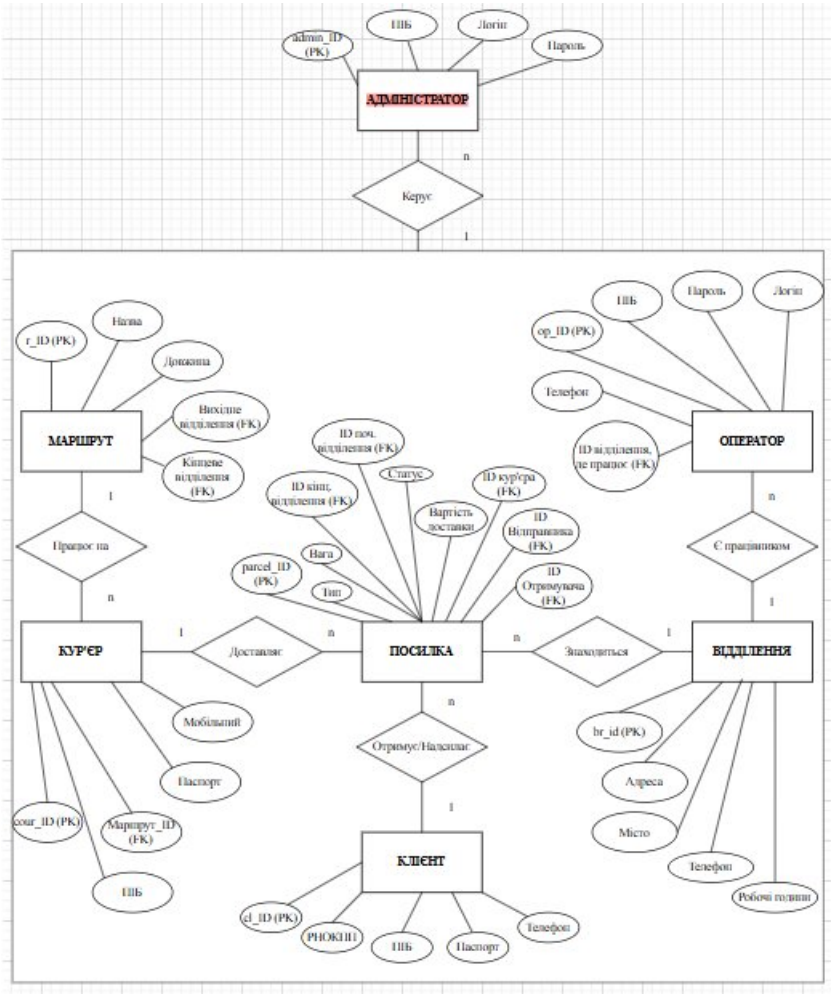


Рисунок 2.1 – ERD структури бази даних

Після етапу логічного проєктування було виконано фізичне проєктування бази даних, яке передбачало визначення типів даних атрибутів, обмежень цілісності, первинних і зовнішніх ключів, а також реалізацію

структури таблиць у середовищі СКБД MySQL. Для кожної сутності, представленої на ER-діаграмі, було створено окрему таблицю з урахуванням вимог нормалізації, що дозволяє мінімізувати надлишковість даних і забезпечити коректну обробку інформації у межах предметної області.

Скрипт створення бази даних, а також усіх таблиць і зв’язків між ними реалізовано мовою SQL та наведено у Додатку А. Наявність скрипта надає можливість відтворення структури БД і подальшого розгортання web-додатку у будь-якому середовищі.

Для перевірки коректності реалізації структури бази даних використано інструмент MySQL Workbench [53], за допомогою якого було побудовано EER-діаграму (Enhanced entity-relationship) на основі створених таблиць. Зіставлення отриманої EER-діаграми з початковою ERD показало повну відповідність, що підтверджує правильність логічного та фізичного проєктування бази даних.

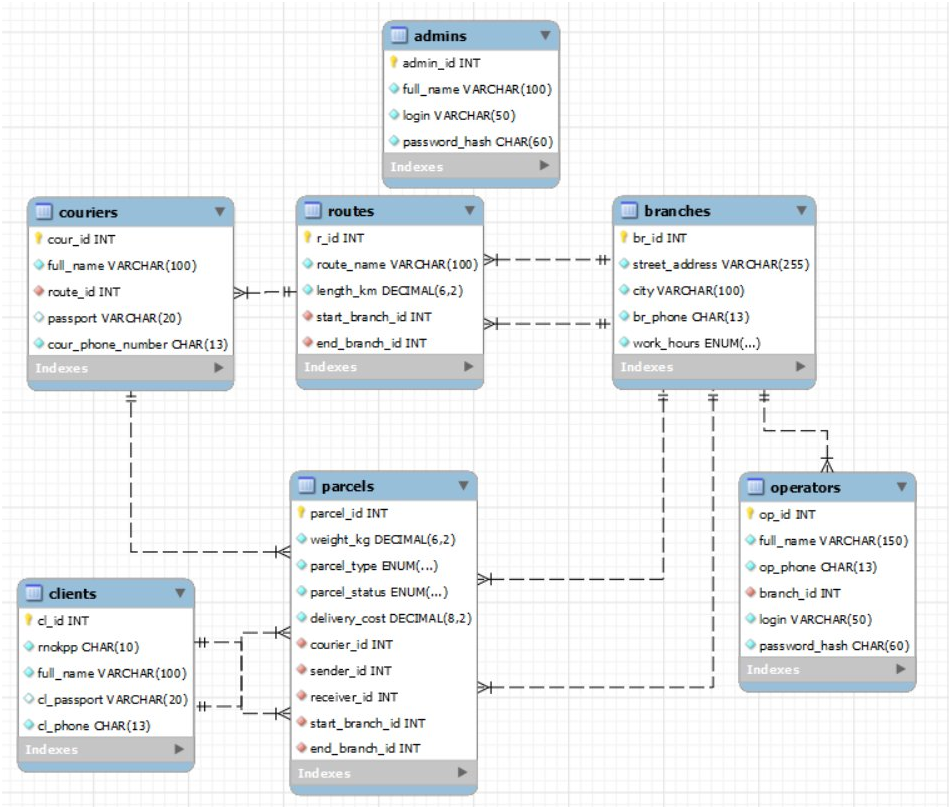


Рисунок 2.2 – Побудована ERR структури бази даних у MySQL

Таким чином, структура бази даних повністю відповідає вимогам предметної області, принципам нормалізації та забезпечує надійний і ефективний доступ до інформації для web-додатку.

2.3.2. Реалізація клієнтської частини web-додатку

Реалізація web-додатку виконана з використанням класичної клієнт-серверної архітектури, що забезпечує розмежування логіки відображення даних та обробки запитів. Такий підхід дозволяє підвищити масштабованість, зручність супроводу та безпеку додатку.

Клієнтська частина web-додатку створена з використанням стандартного набору вебтехнологій (HTML, CSS, JS) для розмітки сторінок, стилізації інтерфейсу та впровадження базової клієнтської логіки. Вона охоплює сторінки авторизації користувачів, перегляду особистого профілю, а також сторінки з таблицями всіх основних сутностей системи. Окрім цього, клієнтська частина містить форми для введення, редагування та додавання записів до бази даних, що забезпечує зручну взаємодію користувачів із системою. Для забезпечення єдиного візуального стилю додатку використовується загальний файл style.css, а також декілька інших файлів стилів, призначених для окремих сторінок або груп сторінок. Такий підхід дозволяє спростити підтримку інтерфейсу та зменшити дублювання коду.

Базові клієнтські сценарії реалізовані за допомогою скриптів, зокрема окремого сценарію для роботи бургер-меню, що забезпечує адаптивну навігацію додатку. Структура більшості сторінок є однотипною, що сприяє уніфікації інтерфейсу та полегшує подальше розширення функціоналу web-додатку.

Приклади окремих фрагментів клієнтської частини, зокрема HTML-розмітки, CSS-оформлення та JS-скрипта для реалізації бургер-меню, наведено у Додатку Б.

2.3.3. Реалізація серверної частини web-додатку

Серверна частина web-додатку реалізує основну бізнес-логіку системи та забезпечує обробку запитів, що надходять від клієнтської частини. Вона відповідає за взаємодію з базою даних, управління сесіями користувачів, перевірку прав доступу, а також виконання операцій, пов’язаних із обробкою та збереженням даних.

У межах реалізації серверної частини додатку було організовано логіку обробки запитів відповідно до ролей користувачів системи. Сервер забезпечує розмежування між адміністраторами та операторами, що дозволяє контролювати доступ до функціональних можливостей web-додатку залежно від рівня прав користувача. Такий підхід підвищує безпеку системи та відповідає типовим вимогам до сучасних інформаційних web-рішень.

Особливим елементом серверної логіки є маршрутизація, реалізована за допомогою механізму Flask @app.route. Вона визначає, яка функція обробляє конкретний запит, формує відповідь та взаємодіє з базою даних. Маршрутизація дозволяє централізовано організувати логіку роботи з різними сутностями системи – відділеннями, клієнтами, кур’єрами, маршрутами та посилками – та розмежувати операції додавання, редагування, перегляду й видалення записів у залежності від ролі користувача. Завдяки цьому забезпечується чітка організація потоків даних і контроль за бізнес-логікою web-додатку.

Серверна логіка реалізує обробку операцій створення, перегляду, редагування та видалення даних для основних сутностей системи, зокрема

відділень, клієнтів, кур’єрів, маршрутів і посилок. Обмін даними між серверною частиною та базою даних здійснюється через окремий шар доступу до даних, що дозволяє централізувати роботу з підключенням до бази даних і виконанням SQL-запитів. Це спрощує супровід коду та дозволяє проводити аналіз продуктивності запитів у межах дослідження.

Окрему увагу приділено реалізації механізмів авторизації та автентифікації користувачів. Серверна частина забезпечує перевірку облікових даних користувачів, управління сесіями та захист доступу до закритих ресурсів додатку. Збереження паролів у БД реалізовано із застосуванням захищених методів хешування, що дозволяє мінімізувати ризики компрометації облікових даних та створює умови для коректного тестування і порівняння захищених і незахищених SQL-запитів у процесі дослідження.

Загальна структура серверної частини web-додатку спроєктована з урахуванням можливості подальшого розширення функціоналу, зокрема додавання нових ролей користувачів, бізнес-операцій або механізмів оптимізації доступу до даних, без суттєвих змін існуючої архітектури. Фрагмент реалізації серверної логіки для авторизації та автентифікації користувачів наведено у Додатку В.

2.3.4 Реалізація та організація взаємодії web-додатку з MySQL

В межах досліджуваного web-додатку організація взаємодії із базою даних MySQL реалізована через окремий клас DBConnection, який відповідає за підключення до БД, виконання SQL-запитів та закриття з’єднання. Такий підхід дозволяє централізувати роботу з базою даних, спростити супровід коду та забезпечити контроль за виконанням запитів.

Облікові дані для підключення зберігаються у файлі для зберігання змінних середовища .env, що забезпечує безпечне використання паролів та конфіденційність інформації. Підключення до бази даних здійснюється через метод connect(), а закриття ресурсу відбувається через метод close(). Така організація дозволяє контролювати ресурси і спрощує супровід коду.

Виконання запитів у web-додатку відбувається переважно через параметризовані SQL-запити (окрім фрагментів, де відбувається тестування "сирих" та ORM-запитів), які дозволяють уникнути ризику SQL-ін’єкцій і безпечно передавати дані від клієнтської частини до бази. Параметризовані запити використовуються для авторизації та аутентифікації користувачів, а також при створенні, редагуванні та видаленні записів у таблицях основних сутностей – відділень, кур’єрів, маршрутів, клієнтів, посилок та операторів.

Організація роботи з базою даних враховує розмежування доступу залежно від ролі користувача. Адміністратори мають повний доступ до всіх сутностей системи, оператори – обмежений доступ лише до даних свого відділення та частини функціоналу, пов’язаного з обробкою посилок. Це підвищує безпеку і відповідає типовим вимогам сучасних web-додатків.

Обмін даними між серверною частиною та базою даних здійснюється за допомогою SQL-запитів, але для ORM-підходу (при проведенні тестування) використовується інша логіка. ORM, реалізована за допомогою SQLAlchemy, дозволяє працювати з об’єктами Python замість безпосередніх SQL-запитів, що спрощує інтеграцію та підвищує зручність розробки, особливо при масштабуванні системи. Для ORM створено базовий клас Base через declarative_base(), налаштовано engine для підключення до MySQL та SessionLocal для управління сесіями.

Параметризовані SQL-запити та ORM-методи у додатку можуть використовуватись паралельно залежно від потреб конкретної операції. Таке поєднання дозволяє ефективно керувати даними, забезпечує безпечне виконання запитів і зручне масштабування функціоналу web-додатку.

Основний файл підключення до бази даних та файл підключення до БД за допомогою SQLAlchemy наведено у Додатку Г.

2.4 Реалізація механізмів безпеки web-додатку

Безпека web-додатку є одним із ключових аспектів при його розробці, оскільки система працює з конфіденційними даними користувачів та виконує операції з БД. У межах розробленого вебдодатку було реалізовано комплекс механізмів безпеки, спрямованих на захист даних, запобігання несанкціонованому доступу та зниження ризиків типових атак.

Одним із основних заходів безпеки є використання параметризованих SQL-запитів під час взаємодії з базою даних MySQL. Такий підхід дозволяє уникнути SQL-ін’єкцій, оскільки дані, що надходять від користувача, передаються окремо від SQL-інструкцій і не можуть впливати на структуру запиту. Під час тестування вебдодаток може використовувати як параметризовані SQL-запити, так і ORM-підхід відповідно до поставлених цілей дослідження.

Для зберігання конфіденційної інформації, зокрема облікових даних для підключення до БД, використовується файл змінних середовища .env. Він дозволяє відокремити конфігураційні параметри від програмного коду та знизити ризик компрометації даних у разі доступу до репозиторію проєкту.

Механізми автентифікації та авторизації реалізовано з урахуванням ролей користувачів. Після успішної автентифікації система визначає рівень доступу користувача до функціоналу вебдодатку, що дозволяє обмежити виконання критичних операцій лише уповноваженими особами та підвищити загальний рівень безпеки.

Крім того, у web-додатку реалізовано перевірку та валідацію вхідних даних, що надходять від користувачів. Подібна практика дозволяє запобігти

помилкам при обробці даних і зменшити ризик експлуатації вразливостей, пов’язаних з введенням некоректних або шкідливих даних. Таким чином, сукупність реалізованих механізмів безпеки забезпечує надійну та стабільну роботу вебдодатку в умовах реального використання.

2.5 Опис функціоналу та сценаріїв роботи web-додатку

Розроблений web-додаток адаптований як для комп’ютерної, так і для мобільної версії. Він призначений для автоматизації роботи служби доставки та забезпечує взаємодію користувачів із системою через веббраузер. Функціонування додатку ґрунтується на клієнт–серверній архітектурі та передбачає розмежування доступу залежно від ролі користувача. Перед початком роботи користувач проходить процедуру авторизації (рис. 2.3), під час якої відбувається перевірка облікових даних та визначення рівня доступу до функціональних можливостей системи.

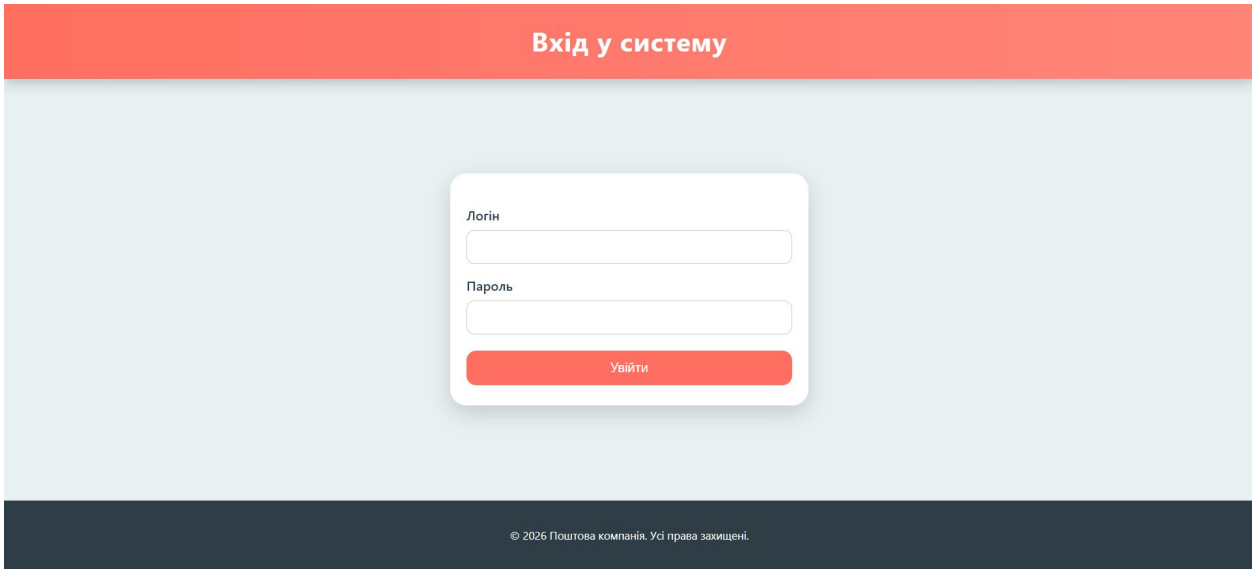


Рисунок 2.3 – Сторінка авторизації

Після успішної авторизації користувач отримує доступ до відповідних сторінок web-додатку, а саме до сторінок "Дашборд", "Посилки", Клієнти",

"Кур'єри", "Маршрути", "Оператори", "Відділення" та "Профіль". Розроблена панель навігації наведена на рис. 2.4.

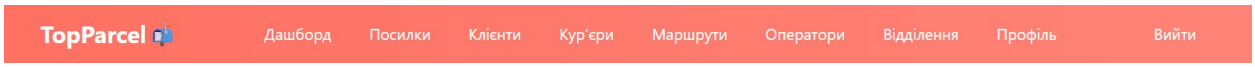


Рисунок 2.4 – Розроблена панель навігації

Сторінка "Дашборд" (рис. 2.5) виконує інформаційно-аналітичну функцію та слугує стартовою сторінкою після входу в систему. На ній відображається узагальнена інформація про поточний стан системи доставки, зокрема ключові показники діяльності, кількість зареєстрованих об’єктів та актуальні дані, що дозволяють швидко оцінити роботу сервісу.

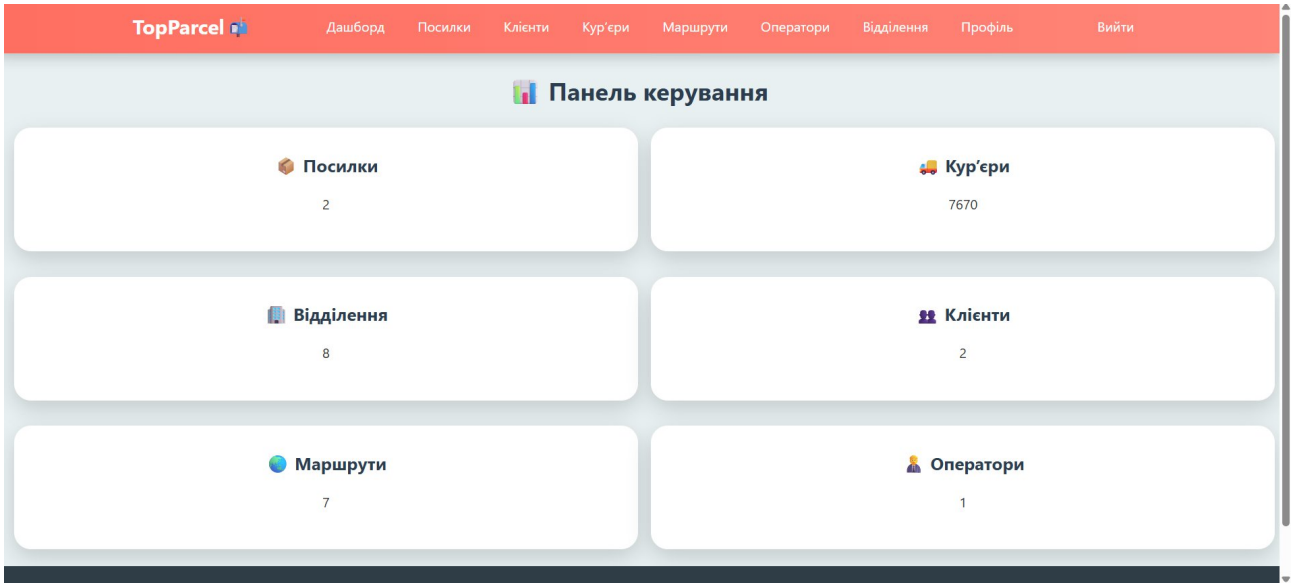


Рисунок 2.5 – Сторінка "Дашборд"

Сторінка "Профіль" (рис. 2.6, 2.7) надає користувачу можливість перегляду та редагування (тільки адміністратору) власних облікових даних. У межах цього розділу користувач може оновлювати персональну інформацію, що забезпечує актуальність даних та підвищує безпеку роботи з системою.

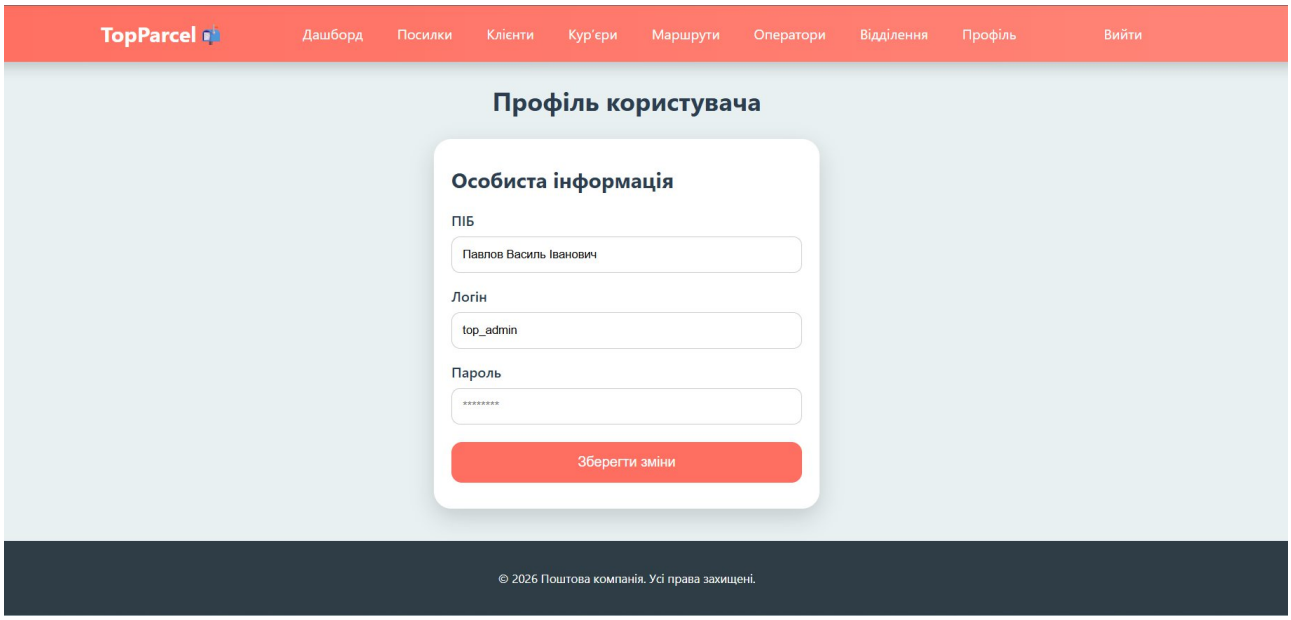


Рисунок 2.6 – Сторінка "Профіль"

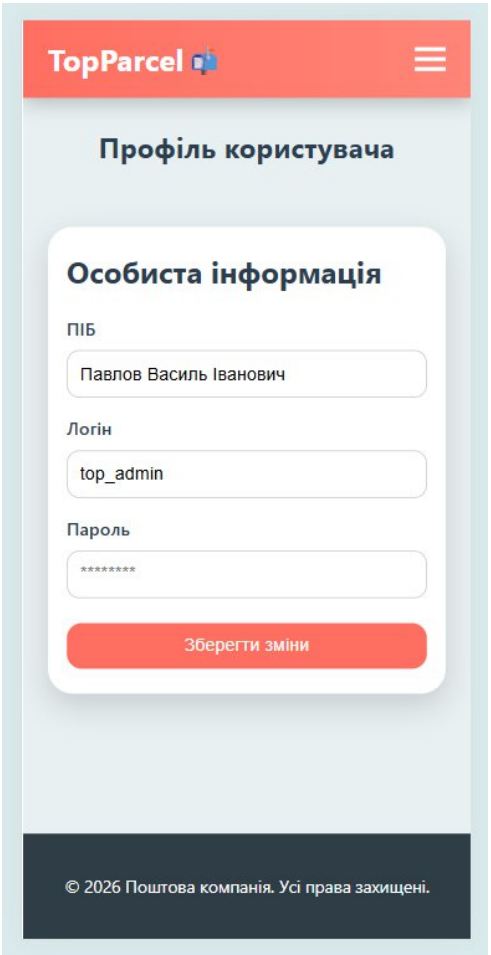


Рисунок 2.7 – Сторінка "Профіль" у мобільний адаптації

На сторінках "Посилки", "Клієнти", "Кур’єри", "Маршрути", "Оператори" та "Відділення" (рис. 2.8) реалізовано однаковий підхід до

взаємодії користувача з даними. Кожна сторінка надає можливість переглядати існуючі записи у вигляді таблиці, отримувати детальну інформацію про конкретний об’єкт, а також виконувати операції додавання, редагування та видалення даних залежно від прав доступу користувача. При ролі "адміністратор" користувач отримує повний доступ до всіх функцій керування, включаючи створення нових записів та зміну існуючих. При ролі "оператор" користувач має обмежений доступ: він може переглядати дані та оновлювати лише ті записи, до яких має доступ.

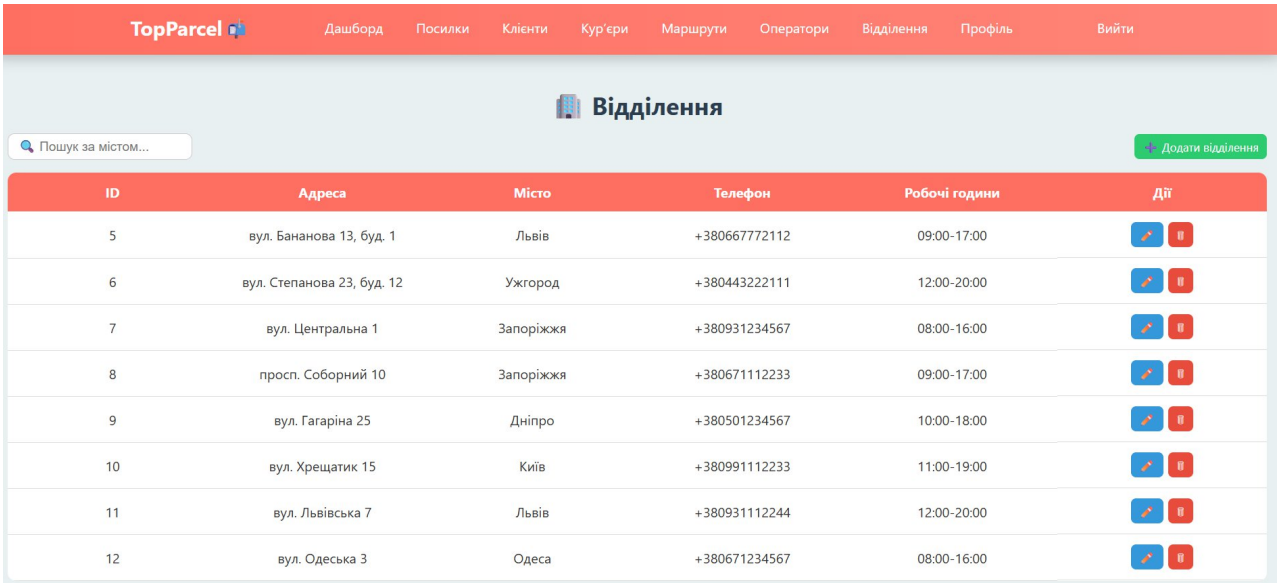


Рисунок 2.8 – Сторінка "Відділення"

Для операцій додавання та редагування передбачено окремі сторінки з однаковим дизайном для всіх розділів ("Посилки", "Клієнти", "Кур’єри", "Маршрути", "Оператори" та "Відділення"). Інтерфейс цих сторінок оформлений у вигляді форм із полями для введення необхідної інформації, що забезпечує уніфікований та зрозумілий користувацький досвід. Користувач може вводити або змінювати дані (рис. 2.9), підтверджувати свої дії відповідною кнопкою та автоматично повертатися до таблиці записів без необхідності повторного переходу через навігаційне меню.

Для видалення запису на будь-якій сторінці достатньо натиснути червону кнопку у відповідному рядку таблиці.

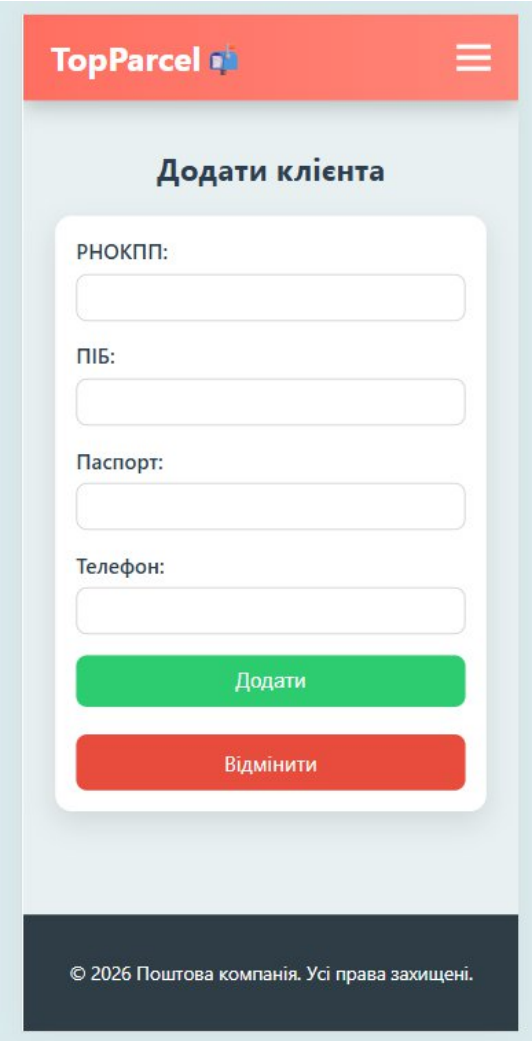


Рисунок 2.9 – Сторінка "Додати клієнта" у мобільній адаптації

2.6 Висновки до розділу

У результаті проведеного проєктування та реалізації web-додатку на Python з використанням MySQL було створено систему, що відповідає всім визначеним функціональним та нефункціональним вимогам. Додаток забезпечує виконання базових операцій з даними (CRUD), реалізує автентифікацію та авторизацію користувачів із різними рівнями доступу, а також підтримує безпечну взаємодію із базою даних, що дозволяє мінімізувати ризики SQL-ін’єкцій та несанкціонованого доступу. Вибір Python як серверної мови та використання фреймворку Flask забезпечили гнучку організацію

серверної логіки, ефективну маршрутизацію HTTP-запитів та можливість точного контролю процесів обробки даних, тоді як використання стандартного набору вебтехнологій (HTML, CSS, JS) дозволило створити адаптивний і зручний інтерфейс для користувачів як на десктопних, так і на мобільних пристроях.

Структура бази даних була ретельно спроєктована з урахуванням нормалізації, принципів цілісності даних та оптимізації продуктивності. Було визначено основні сутності предметної області – користувачів, посилки, відділення, кур'єрів, маршрути та клієнтів – та встановлено зв'язки між ними за допомогою первинних і зовнішніх ключів. Використання ERD і EER-діаграм дозволило наочно перевірити відповідність логічної та фізичної моделі даних, забезпечити ефективне зберігання інформації та спростити подальший супровід бази даних.

Реалізація клієнтської та серверної частин додатку забезпечує зручну взаємодію користувачів із системою та централізоване управління даними. Серверна логіка відповідає за обробку запитів, управління сесіями, контроль прав доступу, виконання операцій із базою даних і реалізацію бізнес-процесів. Клієнтська частина надає уніфіковані інтерфейси для перегляду, додавання, редагування та видалення даних з урахуванням ролей користувачів, а також дозволяє відображати аналітичну інформацію у вигляді зручних для сприйняття дашбордів.

Взаємодія з базою даних у розробленому web-додатку реалізована через окремий клас DBConnection та ORM-підхід із використанням SQLAlchemy, що дозволяє гнучко керувати підключеннями, виконувати як параметризовані SQL-запити, так і роботу з об'єктами Python, а також ефективно проводити порівняльний аналіз продуктивності різних методів доступу до бази даних. Завдяки реалізованим механізмам безпеки – автентифікації, авторизації, валідації даних та захисту конфіденційної інформації через змінні середовища – забезпечується надійний контроль доступу та стабільна робота системи у реальних умовах експлуатації.

Таким чином, розроблений web-додаток підходить і є готовим для проведення аналізу продуктивності при доступі до інформації бази даних MySQL. Він представляє собою масштабовану систему, що дозволяє комплексно оцінювати вплив архітектурних рішень, методів доступу до БД та механізмів безпеки на ефективність роботи вебсистеми, одночасно забезпечуючи зручний та адаптивний інтерфейс для користувачів із різними ролями.

3 АНАЛІЗ ПРОДУКТИВНОСТІ ТА ТЕСТУВАННЯ WEB-ДОДАТКУ

3.1 Методика проведення аналізу продуктивності

Продуктивність web-додатку є одним із ключових показників якості його реалізації, особливо для систем, що працюють з базами даних та обслуговують велику кількість користувачів. Низька швидкодія або неефективна робота з БД може призводити до затримок відповіді, перевантаження серверних ресурсів та зниження стабільності системи в цілому. Тому аналіз продуктивності є обов’язковим етапом розробки та впровадження web-додатків.

Обрана методика аналізу продуктивності web-додатку при доступі до інформації бази даних MySQL полягає у поетапному вимірюванні та порівнянні ключових показників швидкодії серверної частини додатку за різних підходів реалізації доступу до даних. Для проведення дослідження було сформовано набір типових серверних запитів, які відображають найбільш поширені сценарії взаємодії web-додатку з базою даних. До них належать операції авторизації користувачів, додавання нових записів, вибірка даних, оновлення та видалення інформації, виконання складних SQL-запитів із використанням об’єднань таблиць (JOIN), агрегатних функцій, а також сценарій з проблемою N+1 запитів [54].

Згідно методики, кожен із зазначених типів запитів реалізується трьома різними способами доступу до даних: шляхом виконання сирих SQL-запитів, за допомогою використання параметризованих SQL-запитів та із застосуванням ORM-підходу на основі бібліотеки SQLAlchemy. Для забезпечення об’єктивності результатів усі реалізації базуються на однаковій бізнес-логіці та працюють з ідентичними наборами даних.

Перший етап методики передбачає багаторазове послідовне виконання кожного запиту. Для кожного способу доступу до даних запит виконується фіксовану кількість разів (1000 ітерацій), що дозволяє мінімізувати вплив

випадкових факторів та отримати усереднені значення показників продуктивності. У процесі виконання вимірюється час обробки запиту, пропускна здатність, затримка відповіді, а також рівень використання процесорних ресурсів і оперативної пам’яті сервером додатку, після чого відбувається порівняння показників для кожного методу.

Другий етап методики полягає в імітації реального користувацького навантаження за допомогою інструменту для навантажувального тестування Locust. Під час тестування для кожного методу доступу до БД моделюється одночасний запит до web-додатку від різної кількості віртуальних користувачів у діапазоні від 10 до 1000. Такий підхід дозволяє оцінити поведінку серверної частини додатку в умовах підвищеного навантаження, визначити стабільність часу відповіді та перевірити здатність системи до масштабування.

Результати навантажувального тестування аналізуються методом порівняння на основі статистичних показників, що надаються інструментом Locust, зокрема кількості виконаних запитів, середнього, мінімального та максимального часу відповіді, кількості помилок, показників RPS, а також процентильного розподілу часу відповіді. Додатково аналізуються графіки інтенсивності запитів, часу відповіді та кількості активних користувачів, що дозволяє наочно оцінити динаміку роботи web-додатку під навантаженням.

Застосування такої комплексної методики забезпечує можливість всебічного аналізу продуктивності захищеного Python web-додатку при роботі з базою даних MySQL та дозволяє обґрунтовано порівняти ефективність різних підходів доступу до даних у практичних умовах.

3.2 Показники продуктивності

Для оцінки ефективності роботи вебдодатку використовуються показники продуктивності, які дозволяють комплексно оцінити швидкодію та ресурсоємність системи. Показники вимірюються окремо для двох типів тестів: багаторазового послідовного виконання запитів (за допомогою time, psutil) та навантажувального тестування (за допомогою Locust).

3.2.1 Вимірювані показники продуктивності при багаторазовому послідовному виконанні запитів

При багаторазовому послідовному виконанні запитів для кожного запиту та способу доступу до даних (RAW, PARAM, ORM) вимірюються наступні показники:

- середній час виконання запиту (Avg Time, с):

$$Avg\ Time = \frac{1}{N}\sum t_i^N t_i\,, \tag{3.1}$$

де t_i – час виконання і-го запиту, N – кількість ітерацій (наприклад, 1000), i – індекс запиту;

- пропускна здатність (Transactions per second, TPS) – показує, скільки запитів система здатна обробити за одну секунду в середньому:

$$TPS = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{t_i}\,, \tag{3.2}$$

де t_i – час виконання і-го запиту, N – загальна кількість запитів, що тестуються, i – індекс запиту;

~ середня затримка відповіді (Latency, с) – відображає час затримки від моменту надсилання запиту до отримання першого результату:

$$Latency = \frac{1}{N}\sum_{i=1}^{N}(t_{first\ byte,\ i} - t_{start,i}), \quad (3.3)$$

де $t_{start,i}$ – час початку і-го запиту, $t_{firstbyte,i}$ – час надходження першого байту відповіді, N – загальна кількість запитів, що тестуються, i – індекс запиту;

~ середнє використання процесора (CPU%) – показує, скільки відсотків процесорного часу в середньому витрачається на обробку одного запиту:

$$CPU\% = \frac{1}{N}\sum_{i=1}^{N}(CPU_{before,i} - CPU_{after,i}), \quad (3.4)$$

де $CPU_{before,i}$ та $CPU_{after,i}$ – показники завантаження процесора до і після виконання і-го запиту, N – загальна кількість запитів, що тестуються, i – індекс запиту;

~ середнє використання оперативної пам’яті (MEM, bytes) – показує середнє споживання пам’яті сервером під час обробки одного запиту:

$$MEM = \frac{1}{N}\sum_{i=1}^{N}(RSS_{after,i} - RSS_{before,i}), \quad (3.5)$$

де $RSS_{before,i}$ та $RSS_{after,i}$ – обсяг використовуваної оперативної пам’яті до і після виконання і-го запиту, N – загальна кількість запитів, що тестуються, i – індекс запиту;

Отримані показники дозволяють всебічно оцінити продуктивність вебдодатку та порівняти ефективність різних способів доступу до даних БД.

Вони слугують основою для виявлення "вузьких місць" у роботі системи та оптимізації її ресурсоспоживання.

3.2.2 Вимірювані показники продуктивності при моделюванні користувацького навантаження

При моделюванні користувацького навантаження оцінюється поведінка вебдодатку в умовах одночасного доступу до нього (зокрема його БД) великої кількості користувачів. Для цього використовується інструмент для навантажувального тестування, а саме Locust, що дозволяє симулювати одночасну роботу десятків, сотень або тисяч користувачів та збирати детальні статистичні дані.

Основними показниками продуктивності під час такого тестування є:

- кількість виконаних запитів (# Requests) – загальна кількість запитів, оброблених сервером під час тесту;
- кількість помилок (# Fails) – кількість запитів, що завершилися з помилкою або не були оброблені сервером;
- середній час відповіді (Average, ms) – середній час обробки одного запиту сервером:

$$Average = \frac{1}{R} \sum_{j=1}^{R} t_j , \tag{3.6}$$

де t_j – час відповіді j-го запиту, R – загальна кількість запитів, j – індекс конкретного запиту;

- мінімальний та максимальний час відповіді (Min, Max, ms) – дозволяє оцінити розкид часу обробки запитів;
- середній розмір відповіді (Average size, bytes) – середній обсяг даних, повернутих сервером за один запит;

˜ пропускна здатність (RPS – Requests per second) – середня кількість запитів, оброблених сервером за секунду:

$$RPS = \frac{\#Requests}{Total\ Test\ Duration}, \tag{3.7}$$

˜ де Total # Requests – загальна кількість запитів, які були виконані під час тесту, Total Test Duration – загальна тривалість тесту (у секундах);

˜ кількість помилок за секунду (Failures/s) – індикатор стабільності системи під навантаженням.

Для детального аналізу часу відповіді також використовуються процентильні показники: 50%ile, 60%ile, 70%ile, 80%ile, 90%ile, 95%ile, 99%ile, 100%ile (ms). Вони показують, скільки часу займає обробка певної частки запитів. Наприклад, 90%ile означає, що 90% запитів виконані за час, менший або рівний цьому значенню.

Таке детальне вимірювання дозволяє оцінити стабільність вебдодатку, виявити критичні ділянки продуктивності системи при високому навантаженні та порівняти ефективність різних способів доступу до даних БД у практичних умовах.

3.3 Проведення експериментів

Для оцінки продуктивності вебдодатку проведено серію комплексних експериментів, які дозволили не лише виміряти швидкодію системи, а й порівняти ефективність різних підходів доступу до БД MySQL (за допомогою "сирих" SQL-запитів, параметризованих запитів та ORM). Експерименти для групи запитів кожного типу виконувалися на однаковому наборі даних та з ідентичною бізнес-логікою, що забезпечило об’єктивність і коректність отриманих результатів. Такий підхід дозволив зменшити вплив випадкових

факторів та отримати усереднені показники продуктивності для кожного способу доступу до інформації БД.

Перед проведенням експериментів було підготовлено БД: всі таблиці, що брали участь у тестах, були заповнені даними. Експериментальна частина включала два основні підходи до тестування продуктивності. Перший – багаторазове послідовне виконання запитів, у ході якого вимірювався час обробки кожного з них, середнє використання ресурсів сервером під час серії стандартних операцій та інші показники. Другий підхід – моделювання реального користувацького навантаження за допомогою інструменту Locust, що дало змогу оцінити поведінку системи при одночасній роботі великої кількості користувачів з інформацією БД, перевірити стабільність часу відповіді, пропускну здатність та масштабованість вебдодатку.

3.3.1 Проведення багаторазових послідовних запитів

З метою проведення подальшого аналізу продуктивності вебдодатку при доступі до інформації БД MySQL за допомогою багаторазового послідовного виконання запитів було проведено серію експериментів, спрямованих на вимірювання ключових показників швидкодії серверної частини системи. Їх суть полягала в наступному: кожен тип серверного запиту (авторизація користувачів, INSERT, SELECT, UPDATE, DELETE, JOIN-запит, агрегація та сценарій N+1) виконувався послідовно багаторазово трьома методами доступу до даних – із використанням сирих SQL-запитів, із використанням параметризованих SQL-запитів та за допомогою технологій ORM – фіксовану кількість разів (у більшості експериментів проведено 1000 ітерацій для кожного методу).

Для вимірювання показників продуктивності використовувалися Python-бібліотеки time та psutil. Час початку і завершення кожного запиту

фіксувався за допомогою time.perf_counter(), що дозволяло точно визначити Avg Time (середній час виконання запиту) та Latency (час до отримання першого байту відповіді). Використання процесора (CPU%) та оперативної пам’яті (MEM, bytes) відстежувалося через об’єкт процесу psutil.Process(os.getpid()): для цього зберігалися значення process.cpu_percent(interval=0.01) для оцінки середнього завантаження CPU та process.memory_info().rss для оцінки обсягу споживаної оперативної пам’яті до і після виконання запиту. Пропускна здатність (TPS) обчислювалася як обернене значення часу виконання запиту та усереднювалась для всіх ітерацій.

Для кожного методу доступу та взаємодії з даними після завершення серії ітерацій розраховувалися усереднені значення цих показників. Подібний підхід дав змогу об’єктивно порівняти ефективність різних підходів роботи з базою даних та зменшити ризик флуктуацій.

Слід зазначити, що показники використання процесора (CPU%) та оперативної пам’яті (MEM, bytes) відображали лише локальне споживання ресурсів безпосередньо вебпроцесом web-додатку та не враховували загальне системне навантаження або активність інших процесів операційної системи. Результати виконаних тестів у вигляді зведених таблиць виводилися безпосередньо в консоль, що забезпечувало наочний аналіз та подальше узагальнення отриманих показників.

Приклади скриптів, що виконували ці виміри, наведено у Додатку Д.

3.3.2 Проведення навантажувального тестування

Для оцінки продуктивності web-додатку при доступі до інформації бази даних MySQL під реальним користувацьким навантаженням було проведено

серію експериментів із використанням інструменту Locust. Метою такого тестування було визначити поведінку системи при одночасній роботі великої кількості користувачів, оцінити стабільність часу відповіді, пропускну здатність та продуктивність різних методів доступу до БД під високим навантаженням.

Перед проведенням навантажувальних тестів БД була підготовлена: усі таблиці, що використовувалися у сценаріях тестування, були заповнені актуальними даними. Тестування здійснювалося шляхом моделювання одночасної роботи великої кількості користувачів (від 10 до 1000 залежно від сценарію та типу запиту), які виконували стандартні дії. До цих дій належать авторизація користувачів, додавання нових записів (INSERT), отримання даних (SELECT), редагування записів (UPDATE), видалення записів (DELETE), виконання об’єднань таблиць (JOIN), агрегованих запитів (AGGREGATION) та сценаріїв N+1.

Під час виконання тестів у коді вебдодатку було реалізовано можливість вибору методу доступу до бази даних безпосередньо через роут (за допомогою базової конструкції if-else): для кожного запиту перевірявся параметр method, і залежно від його значення виконувався відповідний метод – сирий SQL (raw), параметризований SQL (param) або ORM (orm). Якщо параметр не відповідав жодному з варіантів, сервер повертав помилку (рис. 3.1). Це дозволяло Locust під час навантажувальних тестів автоматично викликати різні методи доступу до БД, фіксуючи показники продуктивності для кожного сценарію та забезпечуючи порівняння ефективності різних підходів у реальному навантаженні.

```
if method == "raw":
    result = raw_method(min_courier_id)
elif method == "param":
    result = param_method(min_courier_id)
elif method == "orm":
    result = orm_method(min_courier_id)
else:
    return jsonify({"error": "Invalid method"}), 400
```

Рисунок 3.1 – Код, що реалізує виконання обраного методу доступу до БД

Під час виконання тестів кожен користувач у Locust створював окремий об’єкт класу HttpUser, який запускав відповідні завдання (@task) з певною затримкою між запитами (wait_time). Для кожного запиту здійснювалася HTTP-взаємодія із сервером, і Locust автоматично відстежував час відповіді, статус коди та обсяг даних. Всі запити оброблялися циклічно для різних користувачів, що дозволяло моделювати реальне одночасне навантаження на БД та вебдодаток, а також отримувати детальну статистику продуктивності по всіх сценаріях.

У ході проведення тестів збиралися детальні показники продуктивності, у тому числі середній час обробки запиту, мінімальний і максимальний час відповіді, перцентилі часу відповіді, загальна кількість запитів, кількість помилок, пропускна здатність у запитах за секунду (RPS), середній розмір відповіді тощо. Для візуального аналізу також формувалися графіки, що відображали загальну кількість запитів за секунду, час відповіді та кількість одночасних користувачів. Для реалізації та збору цих показників використовувалася бібліотека Locust, яка дозволяє моделювати реальне навантаження і відстежувати статистику продуктивності вебдодатку під різними сценаріями.

Скрипти для навантажувального тестування наведено у Додатку Е.

3.4 Аналіз результатів багаторазового послідовного виконання запитів

У ході оцінки продуктивності web-додатку при доступі до інформації бази даних MySQL методом багаторазового послідовного виконання запитів було проведено тестування набору ключових SQL-запитів, що охоплюють основні операції з БД: авторизацію, вставку, вибірку, оновлення, видалення, об’єднання таблиць, агрегатні функції та сценарії, що імітують проблему N+1. Кожен запит виконували трьома методами доступу до бази даних: сирий SQL, параметризований SQL та ORM.

Фінальні результати тестування представлені у таблиці нижче (табл. 3.1).

Таблиця 3.1 – Порівняння продуктивності методів доступу до БД при багаторазовому послідовному виконанні SQL-запитів

Метод	**Запит**	**Avg Time (c)**	**TPS**	**Latency (c)**	**CPU (%)**	**MEM (байт)**
1	2	3	4	5	6	7
RAW	SELECT (автентифікація)	0.001208	896.46	0.001198	2.12	94.208
PARAM	SELECT (автентифікація)	0.001261	858.80	0.001251	1.50	49.152
ORM	SELECT (автентифікація)	0.001363	803.23	0.001351	0.80	42.192
RAW	INSERT	0.005748	185.77	0.001526	1.49	20.48
PARAM	INSERT	0.006003	174.47	0.001658	1.41	8.192

Продовження табл. 3.1

1	2	3	4	5	6	7
ORM	INSERT	0.005886	179.77	0.001631	1.90	4.096
PARAM	SELECT	0.063711	17.93	0.001676	5.61	34160.64
ORM	SELECT	0.154521	7.07	0.003865	5.48	18388.608
RAW	UPDATE	0.001681	673.28	0.001315	1.80	24.576
PARAM	UPDATE	0.001908	582.17	0.001504	1.51	20.48
ORM	UPDATE	0.007420	144.08	0.007384	1.19	49.008
RAW	DELETE	0.002338	549.67	0.002324	2.02	204.8
PARAM	DELETE	0.005461	192.23	0.005442	1.04	180.0
ORM	DELETE	0.010841	104.17	0.010809	0.98	279.04
RAW	JOIN	0.079007	18.58	0.002942	13.02	49176.576
PARAM	JOIN	0.075905	20.09	0.003274	13.09	15593.472
ORM	JOIN	0.151019	9.10	0.133506	19.36	25989.12
RAW	Агрегація	0.029328	35.16	0.029045	3.54	110.808
PARAM	Агрегація	0.030352	33.56	0.030103	4.04	94.208
ORM	Агрегація	0.030599	33.33	0.030552	4.09	81.92
RAW	N+1	0.071824	14.43	0.071808	1.95	33136.64
PARAM	N+1	0.111866	9.09	0.111843	3.00	33300.00
ORM	N+1	0.304199	3.35	0.303158	1.04	78979.84

Проаналізувавши отримані результати можна зробити висновок, що для запитів на автентифікацію користувачів спостерігається висока ефективність усіх методів. Середній час виконання SELECT для авторизації коливається від 0.0012 с у RAW до 0.00136 с у ORM. При цьому TPS для RAW складає 896, PARAM – 859, ORM – 803, що свідчить про дуже високу пропускну здатність при невеликій кількості даних. Латентність у всіх трьох методів низька і практично співпадає з часом виконання, а завантаження процесора та пам’яті незначне. Це демонструє, що для простих операцій будь-який метод забезпечує швидку обробку і мінімальне навантаження на систему.

Для запитів INSERT результати показують, що середній час виконання варіюється від 0.0057 с у RAW до 0.0060 с у PARAM, при цьому TPS для RAW – 186, PARAM – 174, ORM – 180. Латентність усіх методів залишається низькою, а використання пам’яті у ORM мінімальне, що відображає ефективне створення окремих об’єктів без значного споживання ресурсів. RAW і PARAM виконують вставку швидше, тоді як ORM демонструє трохи більшу варіабельність, але при цьому він є зручнішим для роботи з об’єктною моделлю.

Для загальних SELECT-запитів, які вибирають великі обсяги даних, різниця між методами стає помітною. RAW SQL виконує запит за 0.0546 с із TPS 18.91, PARAM трохи повільніше – 0.0637 с із TPS 17.93, а ORM істотно поступається – 0.1545 с із TPS 7.07. Це вказує на накладні витрати ORM при створенні об’єктів для кожного рядка, особливо при великих наборах даних, хоча навантаження на CPU незначне завдяки ефективній оптимізації обробки на стороні інтерпретатора. Використання пам’яті для RAW і PARAM істотно перевищує ORM, що обумовлено тим, що RAW і PARAM повертають дані у вигляді словників чи списків, тоді як ORM створює менше об’єктів, але з великим внутрішнім накладним кодом.

Запити UPDATE демонструють, що RAW обробляє операцію за 0.00168 с із TPS 673, PARAM трохи повільніше – 0.0019 с із TPS 582, а ORM – значно довше 0.00742 с із TPS 144. Це свідчить про те, що для оновлення даних ORM

накладає додаткові витрати через створення і синхронізацію об’єктів у сесії, що знижує швидкість виконання порівняно із сирим SQL.

Запити DELETE показують подібну тенденцію: RAW SQL обробляє видалення за 0.00234 с із TPS 549, PARAM – 0.00546 с із TPS 192, а ORM – 0.01084 с із TPS 104. Використання пам’яті у ORM при цьому максимальне, що пояснюється створенням об’єктів для кожного видаленого запису та необхідністю відстеження змін у сесії.

Для JOIN-запитів різниця між методами стає ще більш вираженою. RAW SQL виконує об’єднання таблиць за 0.079 с із TPS 18.58, PARAM – 0.0759 с із TPS 20.09, а ORM потребує 0.151 с із TPS лише 9.1. Латентність ORM зростає до 0.1335 с, а завантаження CPU до 19 %, що відображає накладні витрати на побудову об’єктної структури та обробку результатів об’єднання.

Аналіз агрегатних запитів показує, що усі три методи працюють приблизно однаково: RAW SQL – 0.0293 с із TPS 35.16, PARAM – 0.0303 с із TPS 33.56, ORM – 0.0306 с із TPS 33.33. Латентність, CPU і використання пам’яті також подібні, що свідчить про ефективність обчислення агрегатних функцій і мінімальний вплив ORM у таких сценаріях.

Найбільш показовим є тест N+1, де для кожного маршруту виконується окремий запит до таблиці кур’єрів. RAW SQL обробляє N+1 сценарій за 0.0718 с із TPS 14.43, PARAM – за 0.1118 с із TPS 9.09, а ORM – за 0.3041 с із TPS лише 3.35. Латентність ORM складає 0.3031 с, а використання пам’яті досягає 78 979 байт, що наочно демонструє класичну проблему N+1: кількість запитів зростає пропорційно до кількості маршрутів, а накладні витрати ORM на створення об’єктів значно уповільнюють обробку. RAW і PARAM показують кращу продуктивність завдяки меншій кількості накладних операцій та обробці даних у вигляді простих структур.

У цілому, результати тестування демонструють, що метод доступу до бази даних має суттєвий вплив на продуктивність web-додатку. RAW SQL забезпечує найвищу швидкість виконання для більшості типів запитів, включно з простими SELECT, INSERT, UPDATE, DELETE, а також

складними JOIN-запитами. Це пояснюється мінімізацією накладних витрат на створення додаткових об’єктів та обробку даних у пам’яті, що дозволяє серверу виконувати запити максимально швидко. Однак при цьому RAW SQL має серйозний недолік – відсутність захисту від SQL-ін’єкцій, оскільки запити формуються без параметризації. Це робить додаток вразливим до потенційно небезпечних введень користувача і потребує додаткових заходів безпеки при використанні такого підходу.

PARAM SQL показує трохи довший час виконання порівняно з RAW, однак забезпечує підвищену безпеку за рахунок параметризації запитів та ефективніше використовує пам’ять, особливо при обробці великих обсягів даних. Цей метод зберігає високу продуктивність для більшості запитів і є компромісом між швидкістю та безпекою, що робить його більш придатним для реальних додатків, де важлива надійність системи.

ORM надає максимальну зручність для розробника завдяки роботі з об’єктною моделлю та спрощенню підтримки коду, проте демонструє суттєво нижчу продуктивність для складних вибірок, JOIN-запитів та сценаріїв N+1. У цих випадках час виконання запитів зростає, TPS падає, а використання пам’яті збільшується через необхідність створення об’єктів для кожного рядка та синхронізації змін у сесії. Найбільш показовим є сценарій N+1, де ORM потребує набагато більше часу на виконання запиту порівняно з RAW і PARAM, що підкреслює ризик значного падіння продуктивності без оптимізації через попереднє завантаження пов’язаних об’єктів (eager loading) або об’єднання запитів на стороні БД.

Цей експеримент підтверджує, що вибір методу доступу до бази даних повинен базуватися на балансі між швидкістю виконання, ефективністю використання пам’яті, безпекою та зручністю розробки. RAW SQL доцільно застосовувати там, де критична максимальна швидкість, але захист даних не відіграє ролі, PARAM SQL забезпечує хороший компроміс між продуктивністю та безпекою, а ORM слід використовувати для підвищення

зручності роботи з об’єктами, але із обов’язковою оптимізацією складних запитів для уникнення проблем з N+1 та надмірним використанням ресурсів.

3.5 Аналіз результатів навантажувального тестування

У рамках оцінки продуктивності web-додатку при доступі до інформації бази даних MySQL також було проведено навантажувальне тестування з використанням Locust для визначення здатності системи обробляти велику кількість одночасних запитів. Було протестовано наступні операції з БД: авторизацію, вставку, вибірку, оновлення, видалення, об’єднання таблиць, агрегатні функції та сценарії, що імітують проблему N+1. Як і у першому тесті, кожен запит виконувався трьома методами доступу до БД: сирим SQL, параметризованим SQL та за допомогою ORM.

Результати навантажувального тестування при авторизації користувачів наведені на рис. 3.2 – 3.4.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	Login ORM	1134	0	4115.52	17	25601	971	30.44	0
POST	Login PARAM	1082	0	4105.68	14	25694	971	29.04	0
POST	Login RAW	1132	0	3838.52	16	24060	971	30.39	0
	Aggregated	3348	0	4018.68	14	25694	971	89.87	0

Рисунок 3.2 – Статистика виконання запитів під навантаженням для сценарію авторизації користувачів різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	AGGREGATION N+1 ORM	50000	51000	52000	52000	54000	55000	56000	56000
GET	AGGREGATION N+1 PARAM	53000	54000	54000	55000	56000	57000	58000	58000
GET	AGGREGATION N+1 RAW	53000	54000	55000	56000	56000	57000	58000	59000
	Aggregated	52000	53000	54000	55000	56000	56000	58000	59000

Рисунок 3.3 – Розподіл часу відповіді (percentiles) для сценарію авторизації користувачів різними методами доступу до БД

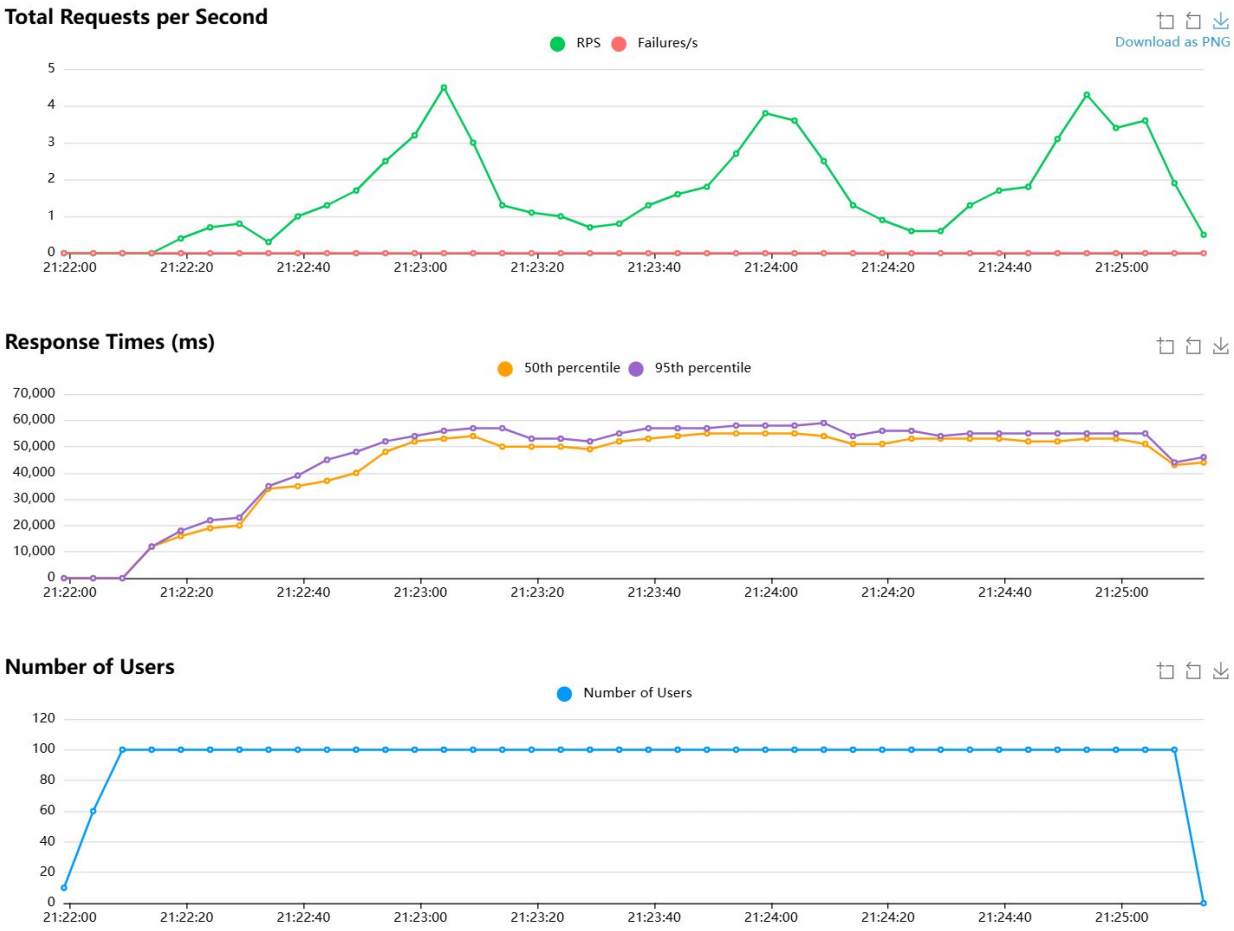


Рисунок 3.4 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію вставки даних наведені на рис. 3.5 – 3.7.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	Add Courier ORM	623	0	141.58	8	604	40	24.32	0
POST	Add Courier PARAM	653	0	74.7	12	559	42	25.49	0
POST	Add Courier RAW	637	0	75.31	12	541	40	24.86	0
	Aggregated	1913	0	96.69	8	604	40.68	74.67	0

Рисунок 3.5 – Статистика виконання запитів під навантаженням для сценарію вставки даних різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	Add Courier ORM	89	120	180	250	360	410	500	600
POST	Add Courier PARAM	63	73	86	100	130	150	250	560
POST	Add Courier RAW	65	78	90	100	130	150	240	540
	Aggregated	69	83	99	130	200	310	450	600

Рисунок 3.6 – Розподіл часу відповіді (percentiles) для сценарію вставки даних різними методами доступу до БД

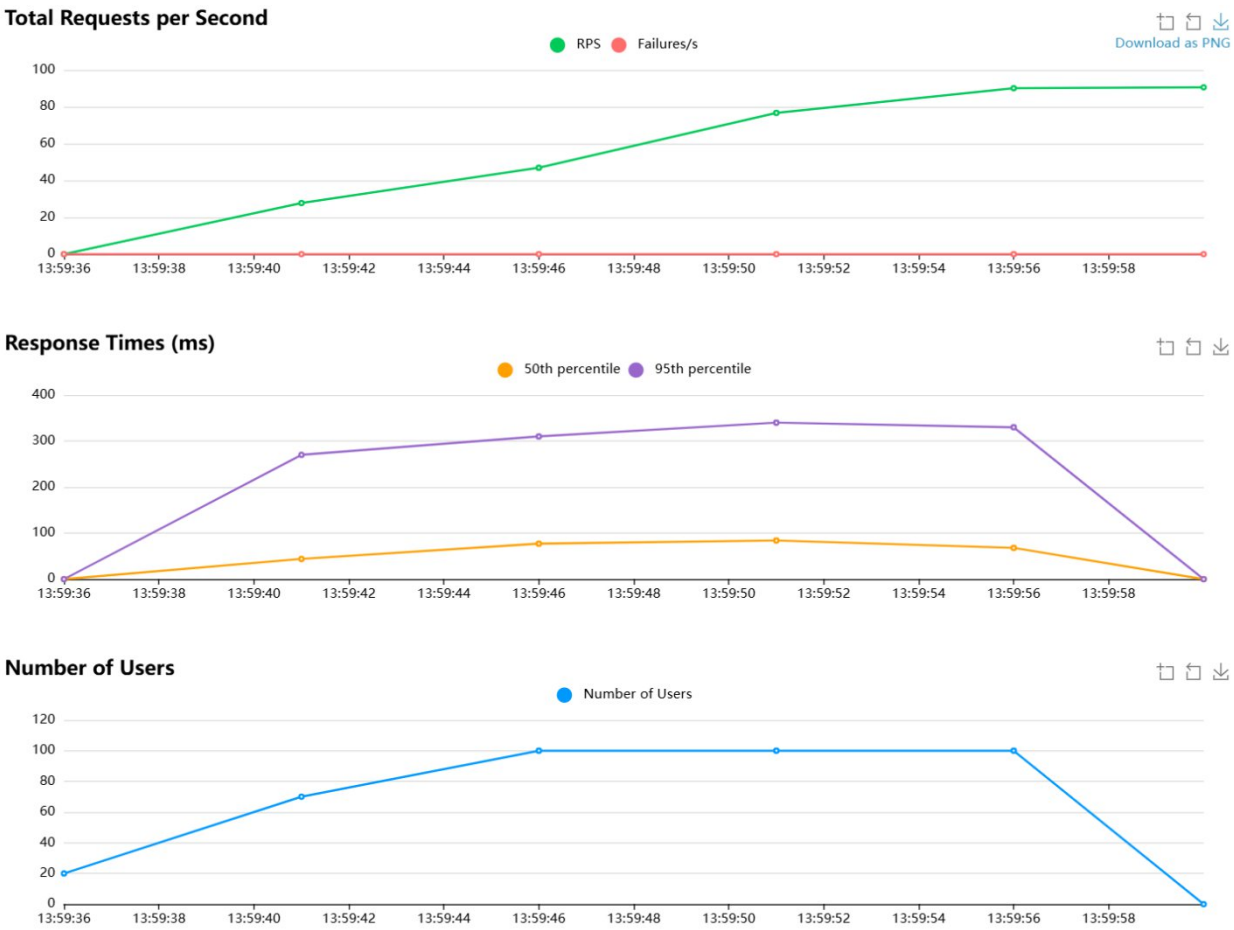


Рисунок 3.7 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію вибірки даних наведені на рис. 3.8 – 3.10.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	Get Couriers ORM	63	0	31395.68	4162	93607	5210891.57	0.61	0
GET	Get Couriers PARAM	29	0	43184.52	11907	93568	12622425	0.28	0
GET	Get Couriers RAW	30	0	40758.37	9177	78934	12622425	0.29	0
	Aggregated	122	0	36500.24	4162	93607	8795157.74	1.19	0

Рисунок 3.8 – Статистика виконання запитів під навантаженням для сценарію вибірки даних різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	Get Couriers ORM	30000	30000	30000	30000	57000	77000	94000	94000
GET	Get Couriers PARAM	39000	47000	56000	63000	89000	91000	94000	94000
GET	Get Couriers RAW	37000	50000	58000	64000	71000	79000	79000	79000
	Aggregated	30000	30000	41000	56000	70000	79000	94000	94000

Рисунок 3.9 – Розподіл часу відповіді (percentiles) для сценарію вибірки даних різними методами доступу до БД

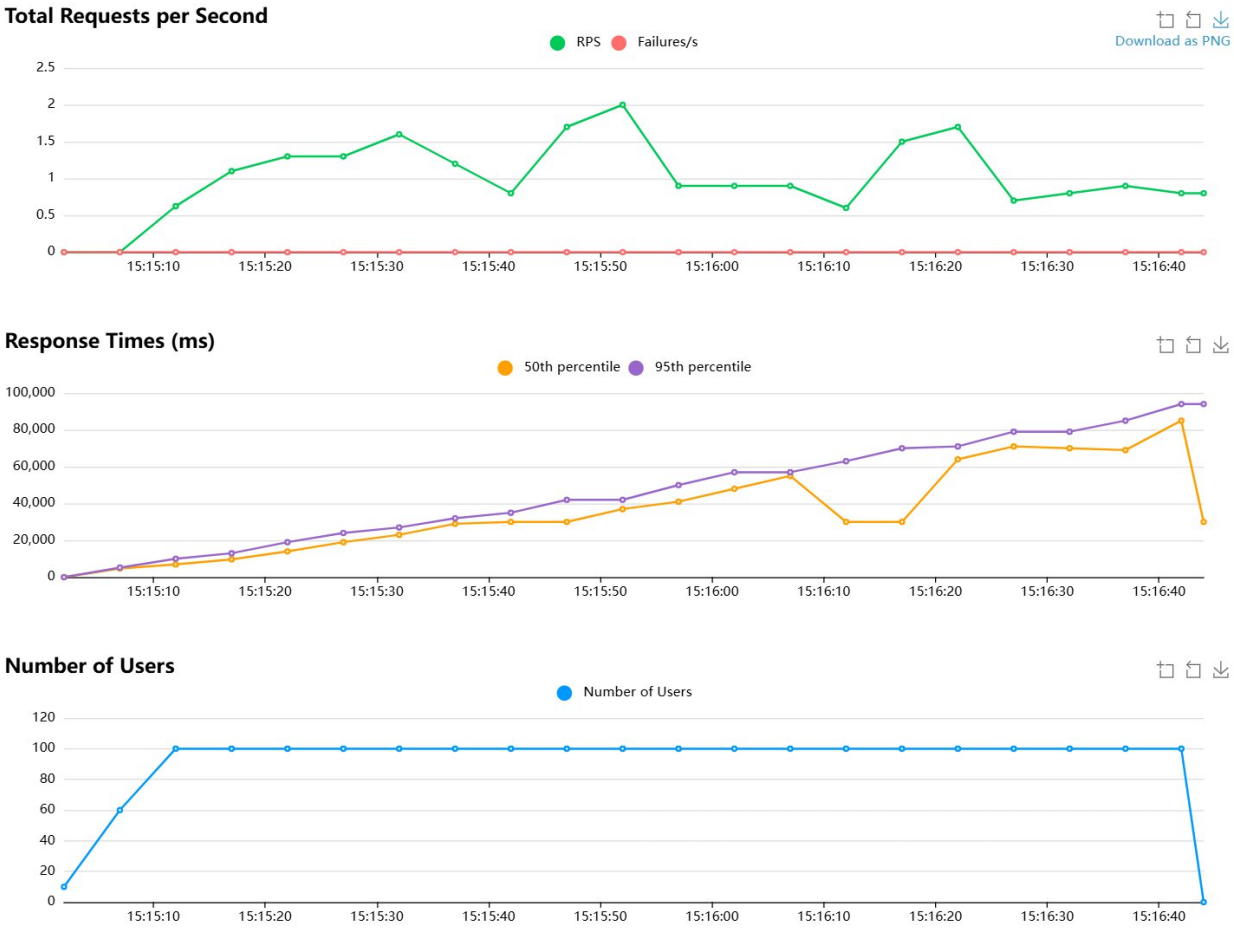


Рисунок 3.10 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію оновлення даних наведені на рис. 3.11 – 3.13.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	Update Courier ORM	42	0	80219.17	6894	197705	12622426	0.2	0
POST	Update Courier PARAM	39	0	85185.73	13132	197072	12622426	0.19	0
POST	Update Courier RAW	46	0	93638.78	9125	197286	12622426	0.22	0
	Aggregated	127	0	86604.98	6894	197705	12622426	0.61	0

Рисунок 3.11 – Статистика виконання запитів під навантаженням для сценарію оновлення даних різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	Update Courier ORM	71000	80000	95000	121000	137000	172000	198000	198000
POST	Update Courier PARAM	78000	85000	106000	119000	178000	187000	197000	197000
POST	Update Courier RAW	87000	102000	122000	140000	153000	164000	197000	197000
	Aggregated	79000	86000	106000	132000	153000	179000	197000	198000

Рисунок 3.12 – Розподіл часу відповіді (percentiles) для сценарію оновлення даних різними методами доступу до БД

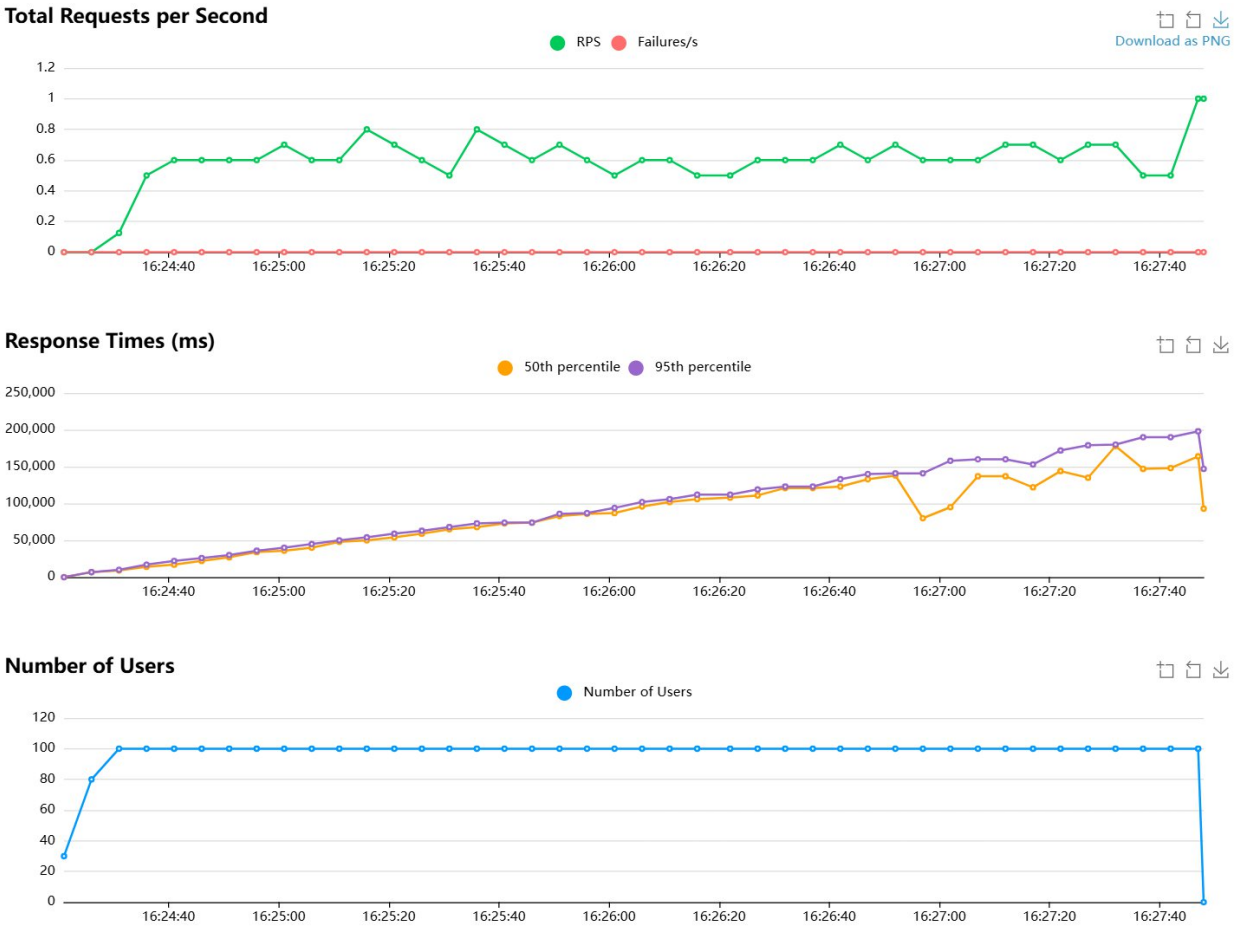


Рисунок 3.13 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію видалення даних наведені на рис. 3.14 – 31.6.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	Delete Courier ORM	38	0	52168.18	4723	127266	12328176	0.26	0
POST	Delete Courier PARAM	39	0	58626.43	3964	123490	12328176	0.27	0
POST	Delete Courier RAW	43	0	57934.92	10126	124245	12328176	0.3	0
	Aggregated	120	0	56333.53	3964	127266	12328176	0.82	0

Рисунок 3.14 – Статистика виконання запитів під навантаженням для сценарію видалення даних різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	Delete Courier ORM	53000	55000	62000	71000	82000	127000	127000	127000
POST	Delete Courier PARAM	58000	65000	72000	85000	96000	116000	123000	123000
POST	Delete Courier RAW	58000	62000	66000	75000	84000	96000	124000	124000
	Aggregated	55000	62000	68000	77000	91000	114000	127000	127000

Рисунок 3.15 – Розподіл часу відповіді (percentiles) для сценарію видалення даних різними методами доступу до БД

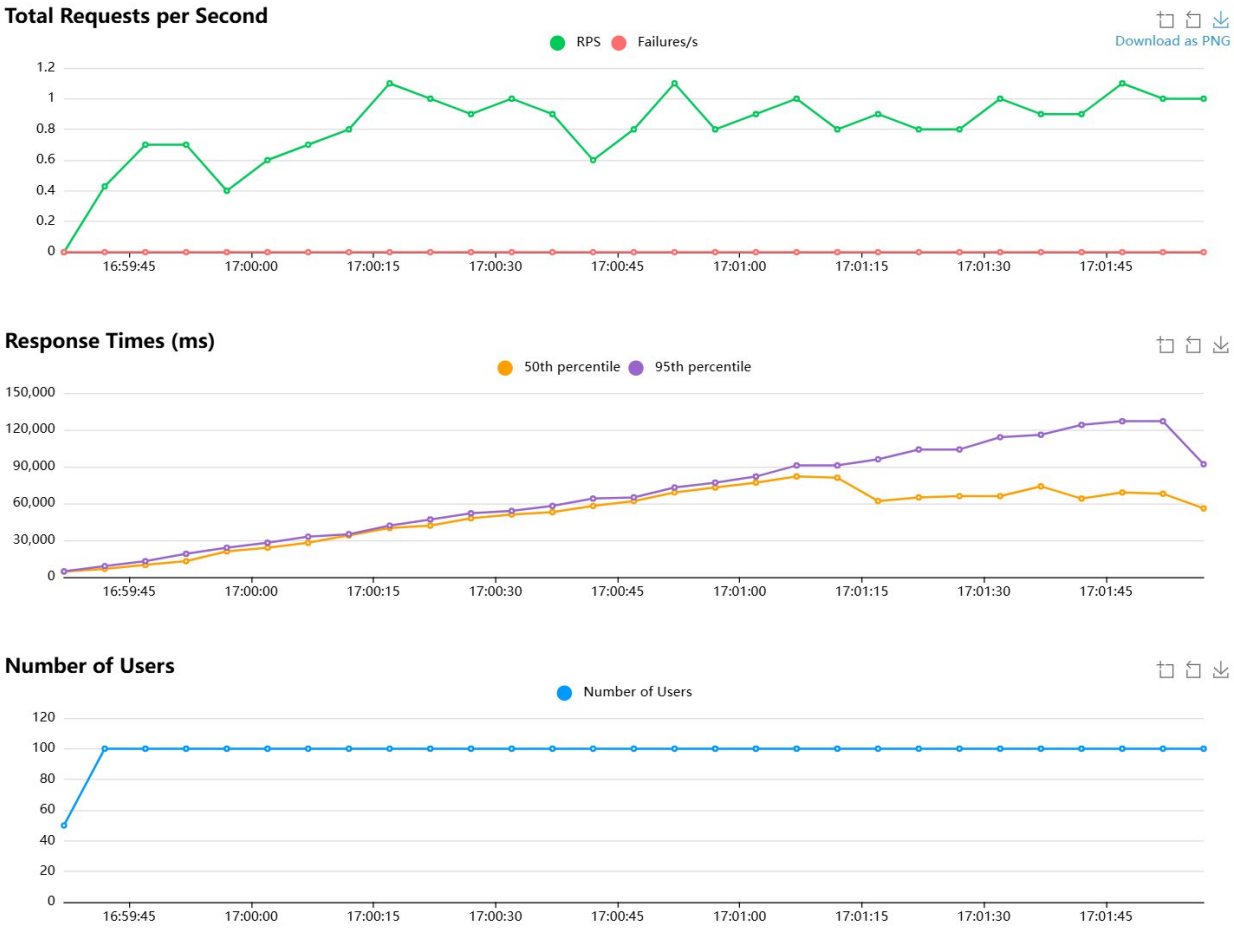


Рисунок 3.16 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію об’єднання таблиць (JOIN) наведені на рис. 3.17 – 3.19.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	SQL JOIN ORM	24	0	9524.56	2976	28902	2003901.17	0.73	0
GET	SQL JOIN PARAM	21	0	18391.7	4714	27628	2005098.62	0.64	0
GET	SQL JOIN RAW	18	0	17457.19	6029	27795	2006107	0.55	0
	Aggregated	63	0	14746.74	2976	28902	2004930.56	1.91	0

Рисунок 3.17 – Статистика виконання запитів під навантаженням для сценарію об’єднання таблиць різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	SQL JOIN ORM	8400	9100	9700	16000	19000	22000	29000	29000
GET	SQL JOIN PARAM	19000	20000	25000	26000	26000	27000	28000	28000
GET	SQL JOIN RAW	18000	20000	22000	24000	25000	28000	28000	28000
	Aggregated	14000	17000	20000	23000	26000	27000	29000	29000

Рисунок 3.18 – Розподіл часу відповіді (percentiles) для сценарію об’єднання таблиць різними методами доступу до БД

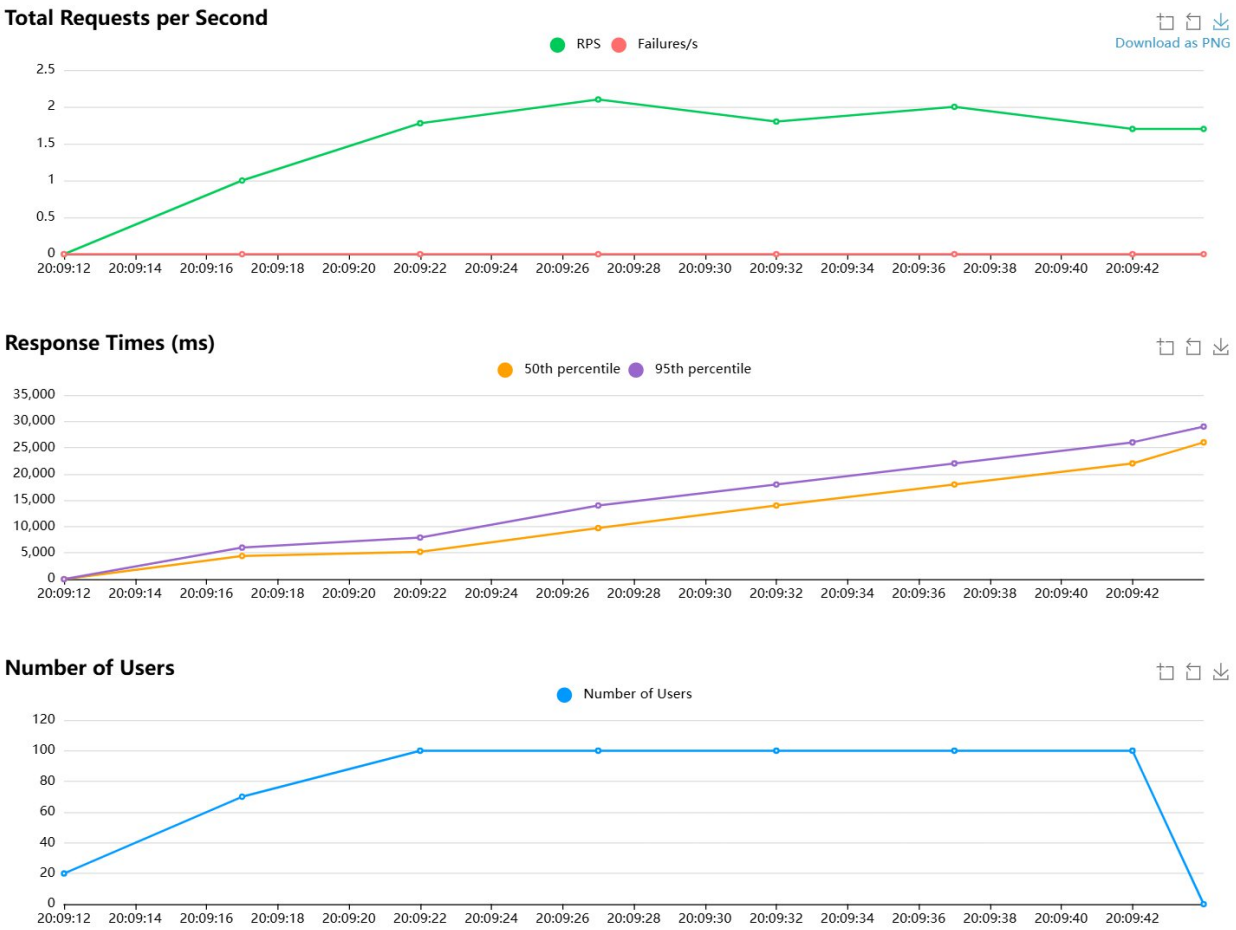


Рисунок 3.19 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію агрегування даних наведені на рис. 3.20 – 3.22.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	AGGREGATION ORM	1065	0	505.8	32	1773	3581	20.54	0
GET	AGGREGATION PARAM	1191	0	297.54	37	1354	3739	22.97	0
GET	AGGREGATION RAW	1192	0	292.83	38	1211	3739	22.99	0
	Aggregated	3448	0	360.24	32	1773	3690.2	66.49	0

Рисунок 3.20 – Статистика виконання запитів під навантаженням для сценарію агрегування даних різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	AGGREGATION ORM	140	710	850	1000	1200	1400	1600	1800
GET	AGGREGATION PARAM	360	420	460	500	550	600	930	1400
GET	AGGREGATION RAW	360	420	450	490	540	580	720	1200
	Aggregated	340	430	480	550	850	1100	1500	1800

Рисунок 3.21– Розподіл часу відповіді (percentiles) для сценарію агрегування даних різними методами доступу до БД

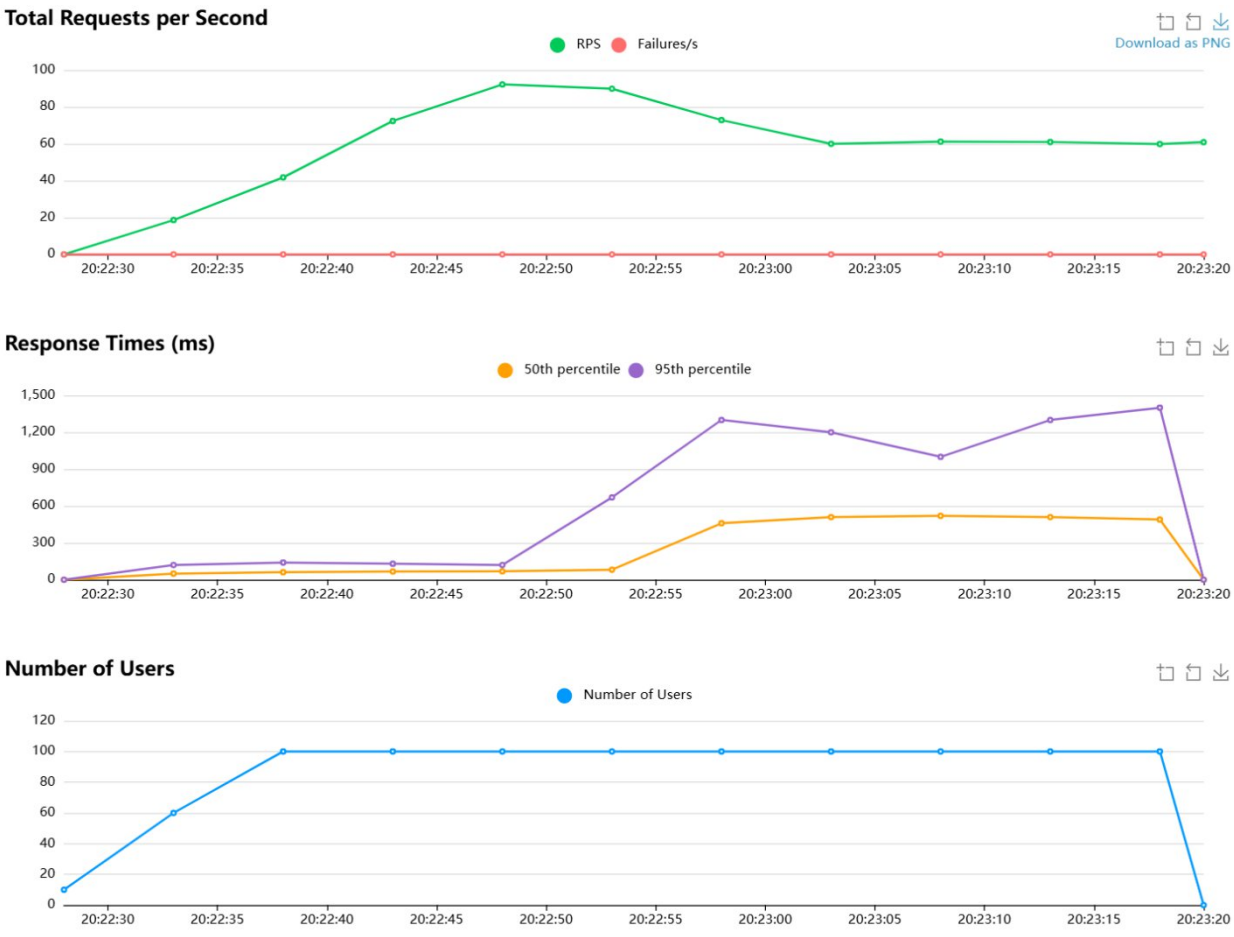


Рисунок 3.22 – Графіки продуктивності системи під навантаженням

Результати навантажувального тестування при виконанні сценарію N+1 наведені на рис. 3.23 – 3.25.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	AGGREGATION N+1 ORM	130	0	42804.4	10960	56167	5235.77	0.66	0
GET	AGGREGATION N+1 PARAM	103	0	51992.2	26175	57914	6134	0.53	0
GET	AGGREGATION N+1 RAW	102	0	52703.62	35297	58850	6134	0.52	0
	Aggregated	335	0	48643.4	10960	58850	5785.43	1.71	0

Рисунок 3.23 – Статистика виконання запитів під навантаженням при виконанні сценарію N+1 різними методами доступу до БД

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	AGGREGATION N+1 ORM	50000	51000	52000	52000	54000	55000	56000	56000
GET	AGGREGATION N+1 PARAM	53000	54000	54000	55000	56000	57000	58000	58000
GET	AGGREGATION N+1 RAW	53000	54000	55000	56000	56000	57000	58000	59000
	Aggregated	52000	53000	54000	55000	56000	56000	58000	59000

Рисунок 3.24– Розподіл часу відповіді (percentiles при виконанні сценарію N+1 різними методами доступу до БД

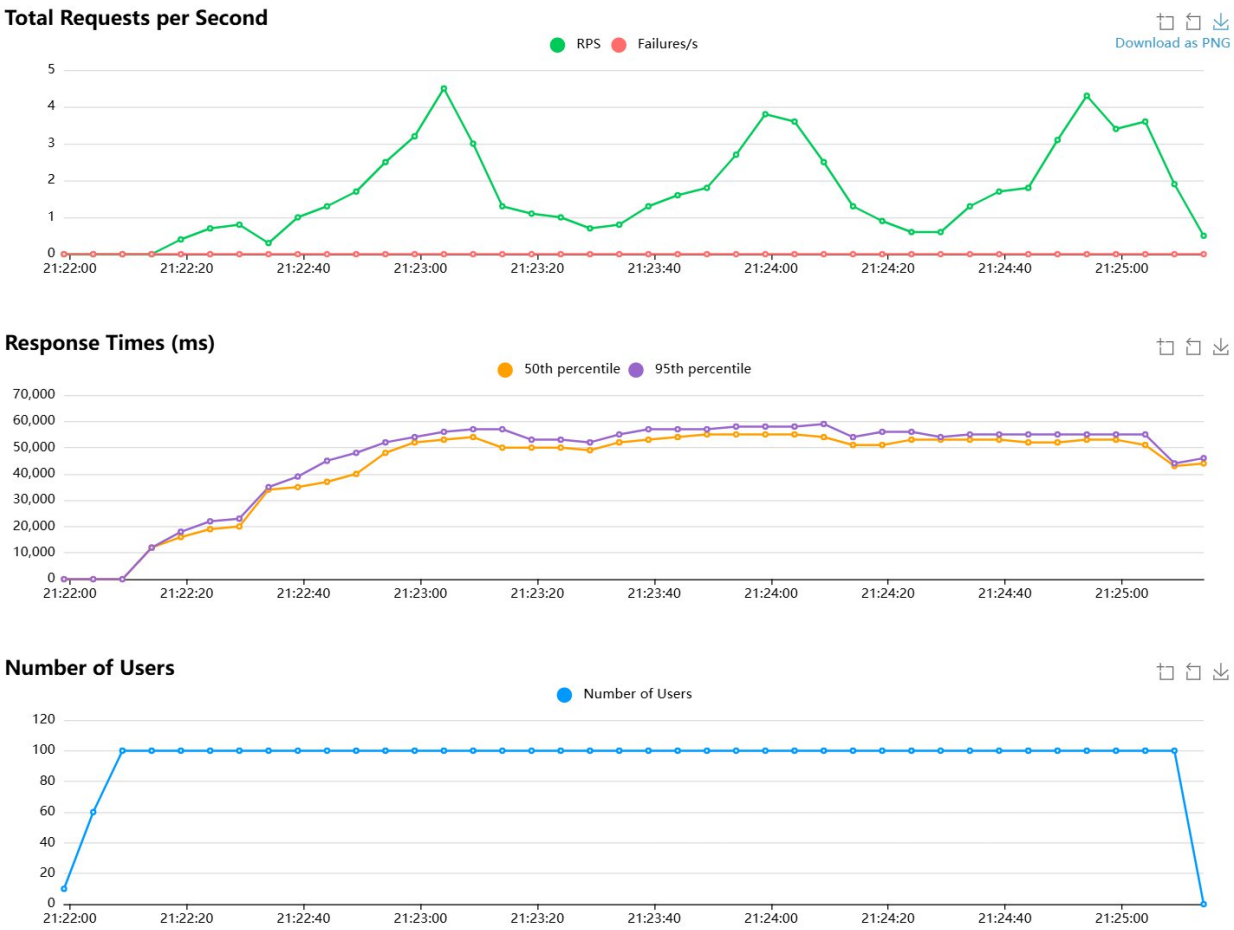


Рисунок 3.25 – Графіки продуктивності системи під навантаженням

У ході навантажувального тестування результати демонструють помітні відмінності у продуктивності залежно від методу доступу до БД. Для сценарію авторизації користувачів усі методи виконують запити швидко та стабільно: середній час відповіді для RAW і PARAM коливається близько 3-4 секунд, тоді як ORM трохи повільніший через накладні витрати на роботу з об’єктною моделлю. Розподіл часу відповіді (percentiles) показує, що при пікових навантаженнях час відповіді зростає для ORM, тоді як RAW і PARAM залишаються трохи швидшими. Це свідчить про більшу ефективність параметризованих та сирих запитів для сценаріїв з великою кількістю одночасних підключень.

Сценарій вставки даних демонструє, що PARAM і RAW методи мають майже однаковий середній час відповіді – близько 75 мс, тоді як ORM трохи повільніше (141 мс). Percentiles показують, що пікові затримки у ORM

досягають 600 мс, тоді як RAW і PARAM залишаються у межах 540-560 мс. Пропускна здатність (RPS) для RAW і PARAM трохи вища, що підтверджує їх ефективність при масових операціях вставки.

Сценарій вибірки великої кількості даних (приблизно 50 000 записів) демонструє істотне зростання часу відповіді, особливо для ORM: середній час відповіді ORM досягає понад 31 секунди, PARAM – близько 43 секунд, а RAW – приблизно 40 секунд. Це пов’язано зі складністю запитів та обсягом даних, що повертаються, а також накладними витратами на десеріалізацію об’єктів у ORM. Percentiles показують, що при піковому навантаженні час відповіді досягає понад 94 секунд, що підкреслює необхідність оптимізації складних SELECT-запитів та обмеження обсягу повернутих даних.

Сценарій оновлення даних показав іншу тенденцію: ORM виявився найшвидшим серед трьох методів із середнім часом ≈80 секунд, PARAM трохи повільніший (≈85 секунд), а RAW – найповільніший (≈94 секунди). Percentiles демонструють пікові затримки до 198 секунд. Сценарій оновлення даних показав трохи порушену тенденцію: ORM виявився швидшим за RAW і PARAM. Це пояснюється тим, що ORM оптимізує оновлення через сесію та пакетну синхронізацію об’єктів, тоді як RAW і PARAM обробляють кожен запит окремо, що збільшує накладні витрати при паралельному доступі.

Сценарій видалення даних показав, що всі методи демонструють приблизно однакову продуктивність. ORM має трохи нижче максимальні percentiles, що свідчить про ефективну обробку транзакцій у сесії, тоді як RAW і PARAM, скоріш за все, іноді зіштовхуються з блокуваннями рядків.

Сценарій об’єднання таблиць (JOIN) продемонстрував відхилення від загальної тенденції: середній час відповіді у ORM менший, але кількість запитів більша. Це може бути через те, що ORM розбиває операцію на частіші, але легші запити завдяки оптимізації об’єктної моделі, тоді як RAW і PARAM виконують менше, проте більш “важких” запитів.

Сценарій агрегації даних продемонстрував наступні результати: середній час відповіді для ORM ≈0,5 секунди, PARAM ≈0,3 секунди, RAW

≈0,29 секунди. Percentiles підтверджують невелике розширення часу відповіді навіть при пікових навантаженнях. Це свідчить, що операції агрегації добре оптимізовані на стороні бази даних і не створюють суттєвих затримок.

Сценарій N+1 показав критичні відмінності: ORM забезпечує середній час відповіді ≈42,8 секунди, PARAM ≈51,9 секунди, RAW ≈52,7 секунди, а максимальні percentiles досягають 56-59 секунд. Це чітко демонструє проблему N+1 при необережному використанні ORM або багатьох послідовних запитів. RAW і PARAM виявляються більш затратними за часом через багаторазове виконання запитів, тоді як ORM відносно стабільний завдяки кешуванню об’єктів у сесії.

У цілому, навантажувальне тестування підтвердило, що при паралельних запитах RAW SQL забезпечує високу пропускну здатність для простих операцій, проте при складних вибірках, JOIN-запитах та N+1 сценаріях спостерігаються значні пікові затримки. PARAM демонструє дещо меншу продуктивність, але забезпечує більшу стабільність виконання та додатковий захист запитів від ін’єкцій, що робить його більш безпечним для роботи під навантаженням. ORM забезпечує зручне управління об’єктною моделлю та транзакціями, демонструє більш стабільні результати при складних оновленнях і сценаріях N+1, проте поступається у швидкості масових вибірок і об’єднань таблиць без використання попереднього завантаження даних або оптимізації запитів. Графіки RPS, response time та кількості користувачів підтвердили ці тенденції: система стабільно обробляє навантаження до певного піку, після якого час відповіді різко зростає, особливо для RAW і PARAM при складних операціях, що вказує на необхідність оптимізації запитів і використання параметризованого підходу або ORM для масштабованих сценаріїв.

3.6 Висновки до розділу

Таким чином, при доступі до інформації бази даних MySQL продуктивність web-додатку суттєво залежить від обраного методу взаємодії з даними. RAW SQL забезпечує найвищу швидкість виконання більшості запитів, особливо для простих SELECT, INSERT, UPDATE та DELETE операцій, завдяки мінімальним накладним витратам на обробку даних. Однак такий підхід не гарантує безпеку від SQL-ін’єкцій.

PARAM SQL демонструє трохи більший час виконання порівняно з RAW, але забезпечує стабільність роботи та безпеку за рахунок параметризації запитів. Цей метод ефективно використовує пам’ять при обробці великих обсягів даних і є компромісом між продуктивністю та надійністю системи.

ORM показує нижчу продуктивність при складних вибірках, JOIN-запитах та сценаріях N+1 через накладні витрати на створення та синхронізацію об’єктів у сесії. Водночас він забезпечує зручність роботи з об’єктною моделлю та стабільне управління транзакціями, що робить його оптимальним для складної бізнес-логіки та підтримки коду.

Навантажувальні експерименти показали, що для простих операцій RAW і PARAM забезпечують високу швидкість, але при складних запитах і пікових навантаженнях час відповіді може зростати. ORM показує відносно стабільні результати в сценаріях N+1 завдяки оптимізації роботи з сесіями, проте поступається за швидкістю масових SELECT та JOIN-запитів. Вибір методу доступу до БД повинен базуватися на балансі між швидкістю виконання, ефективністю використання ресурсів, безпекою та зручністю розробки.

ВИСНОВКИ

У ході дипломної роботи було здійснено аналіз продуктивності захищеного web-додатку при доступі до інформації бази даних MySQL з використанням різних підходів до взаємодії з нею, а саме: сирого SQL, параметризованих SQL-запитів та ORM-технологій.

У межах роботи було розроблено ER-діаграму бази даних, спроєктовано та реалізовано функціональний web-додаток із механізмами автентифікації та авторизації, реалізацією базових операцій роботи з даними (додавання, вибірки, оновлення та видалення записів), під'єднаною базою даних MySQL і базовими засобами захисту, що дозволило наблизити умови експериментів до реальних сценаріїв експлуатації системи.

Під час дослідження було виконано послідовне та навантажувальне тестування ключових операцій доступу до даних, включно з SELECT, INSERT, UPDATE, DELETE, JOIN-запитами, агрегатними функціями та сценаріями, що імітують проблему N+1. Отримані результати показали, що метод доступу до інформації має суттєвий вплив на швидкодію, стабільність та споживання ресурсів web-додатку при доступі до інформації бази даних MySQL.

Було встановлено, що використання сирого SQL забезпечує найвищу продуктивність для більшості типових запитів завдяки мінімальним накладним витратам, однак цей підхід не забезпечує захисту від SQL-ін'єкцій. Параметризовані SQL-запити демонструють дещо нижчу швидкість виконання, однак суттєво підвищують рівень захищеності web-додатку та залишаються ефективними в умовах навантаження. ORM-підхід забезпечує зручність розробки й кращу підтримуваність коду, проте за наявності складних вибірок, об'єднання таблиць та недостатньо оптимізованих сценаріїв доступу до даних може призводити до зниження продуктивності.

Таким чином, результати проведеного дослідження підтверджують, що при розробці захищених web-додатків вибір методу доступу до бази даних MySQL повинен здійснюватися з урахуванням характеру запитів, вимог до продуктивності, рівня безпеки та масштабованості системи. Отримані висновки можуть бути використані під час проєктування та оптимізації web-додатків, орієнтованих на роботу з великими обсягами даних і високим навантаженням.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Shklar L., Rosen R. Web Application Architecture: Principles, Protocols and Practices [Електронний ресурс]. – с. 201-202. – Режим доступу: http://ndl.ethernet.edu.et/bitstream/123456789/39394/1/18.pdf
2. Singh A. Categories of Web Applications and Characteristics: A Review [Електронний ресурс] // Assistant Professor in Computer Science. – с. 176(3). – Режим доступу: https://urr.shodhsagar.com/index.php/j/article/view/745/728
3. An introduction to web applications architecture [Електронний ресурс]. – The Open University. – Режим доступу: https://www.open.edu/openlearn/science-maths-technology/an-introduction-web-applications-architecture/content-section-1.1
4. Web-Application Architecture [Електронний ресурс] / Hillel IT School Blog. – Режим доступу: https://blog.ithillel.ua/articles/web-application-architecture
5. Web Application Architecture: Client-Server and Other Models [Електронний ресурс] / Positiwise. – Режим доступу: https://positiwise.com/blog/web-application-architecture#1_Client-Server_Architecture
6. What is MySQL? [Електронний ресурс] / Oracle. – Режим доступу: https://www.oracle.com/mysql/what-is-mysql/
7. mysqli – MySQL Improved Extension [Електронний ресурс] / PHP Manual. – Режим доступу: https://www.php.net/manual/en/book.mysqli.php
8. PDO – PHP Data Objects [Електронний ресурс] / PHP Manual (укр.). – Режим доступу: https://www.php.net/manual/uk/intro.pdo.php
9. MySQLi Quickstart: Connections [Електронний ресурс] / PHP Manual. – Режим доступу: https://www.php.net/manual/en/mysqli.quickstart.connections.php
10. Eloquent ORM [Електронний ресурс] / Laravel Documentation, версія 12.x. – Режим доступу: https://laravel.com/docs/12.x/eloquent

11. Laravel – The PHP Framework for Web Artisans [Електронний ресурс] / Laravel. – Режим доступу: https://laravel.com/
12. Doctrine ORM [Електронний ресурс] / The Doctrine Project. – Режим доступу: https://www.doctrine-project.org/projects/orm.html
13. Propel ORM [Електронний ресурс] / Propel ORM. – Режим доступу: https://propelorm.org/
14. Doctrine ORM 3.6: Configuration [Електронний ресурс] / The Doctrine Project. – Режим доступу: https://www.doctrine-project.org/projects/doctrine-orm/en/3.6/reference/configuration.html
15. MySQL Workbench Tutorial: Plugins with PHP and PDO [Електронний ресурс] / MySQL Documentation. – Режим доступу: https://dev.mysql.com/doc/workbench/en/wb-tutorial-plugins-php-pdo.html
16. PEP 249 – Python Database API Specification v2.0 [Електронний ресурс] / Python Enhancement Proposals. – Режим доступу: https://peps.python.org/pep-0249/
17. mysql-connector-python [Електронний ресурс] / Python Package Index (PyPI). – Режим доступу: https://pypi.org/project/mysql-connector-python/
18. PyMySQL Documentation [Електронний ресурс]. – Режим доступу: https://pymysql.readthedocs.io/en/latest/
19. mysqlclient User Guide [Електронний ресурс] / Read the Docs. – Режим доступу: https://mysqlclient.readthedocs.io/user_guide.html
20. MySQL Connector/Python Developer Guide [Електронний ресурс] / Oracle. – с. 13-15. – Режим доступу: https://downloads.mysql.com/docs/connector-python-en.a4.pdf
21. What is ORM in Programming? [Електронний ресурс] / YouStable. – Режим доступу: https://www.youstable.com/uk/blog/what-is-orm-in-programming/
22. SQLAlchemy [Електронний ресурс] / Python Package Index (PyPI). – Режим доступу: https://pypi.org/project/SQLAlchemy/

23. Django – Database Queries [Електронний ресурс] / Django Documentation. – Режим доступу: https://docs.djangoproject.com/en/6.0/topics/db/queries/
24. Peewee ORM Documentation [Електронний ресурс] / Peewee ORM. – Режим доступу: https://docs.peewee-orm.com/en/latest/
25. Flask – Web Development, One Drop at a Time [Електронний ресурс] / Pallets Projects. – Режим доступу: https://flask.palletsprojects.com/en/stable/
26. Django Documentation [Електронний ресурс] / Django Documentation. – Режим доступу: https://docs.djangoproject.com/en/6.0/
27. FastAPI – Modern, Fast (high-performance), Web Framework [Електронний ресурс] / FastAPI. – Режим доступу: https://fastapi.tiangolo.com/
28. Flask-SQLAlchemy Documentation [Електронний ресурс] / Flask-SQLAlchemy. – Режим доступу: https://flask-sqlalchemy.readthedocs.io/en/stable/
29. SQLModel – SQL and Pydantic Models for Python [Електронний ресурс] / Tiangolo. – Режим доступу: https://sqlmodel.tiangolo.com/
30. aiomysql Documentation [Електронний ресурс] – с. 15. – Режим доступу: https://aiomysql.readthedocs.io/_/downloads/en/v0.0.21/pdf/
31. Tortoise ORM – Clean, Familiar Python Interface [Електронний ресурс] / Tortoise ORM. – Режим доступу: https://tortoise.github.io/#clean-familiar-python-interface
32. SQLAlchemy ORM – AsyncIO Extension [Електронний ресурс] / SQLAlchemy Documentation. – Режим доступу: https://docs.sqlalchemy.org/en/20/orm/extensions/asyncio.html
33. Hamidi A., Hamraz A. R., Rahmani K. Database Security Mechanisms in MySQL [Електронний ресурс] // Afghanistan Research Journal – Natural Science. – 2022. – с. 3 (використано сторінку 3). – Режим доступу: https://www.arj.af/index.php/arj/article/view/66
34. Nawrocki M., Kołodziej J. Vulnerabilities of Web Applications: Good Practices and New Trends [Електронний ресурс] / Applied Cybersecurity & Internet

Governance (ACIG), Vol. 3, No. 2, 2024. – С. 126-127, 136. – Режим доступу: https://www.acigjournal.com/pdf-199521-124464?filename=Vulnerabilities-of-Web-Ap.pdf

35. Preventing MySQL Injection Attacks: Best Practices [Електронний ресурс] / Cybersecurity Decoder – Режим доступу: https://cybersecuritydecoder.com/threats/sql-injection/preventing-mysql-injection-attacks-best-practices-1179
36. Xdebug – The Debugger and Profiler Tool for PHP [Електронний ресурс] / Xdebug. – Режим доступу: https://xdebug.org/
37. Blackfire – Continuous Performance Management [Електронний ресурс] / Blackfire. – Режим доступу: https://www.blackfire.io/
38. Tideways – PHP Monitoring & Profiler [Електронний ресурс] / Tideways. – Режим доступу: https://tideways.com/
39. Doctrine DBAL Documentation [Електронний ресурс] / Doctrine Project. – Режим доступу: https://www.doctrine-project.org/projects/doctrine-dbal/en/4.4/index.html
40. Laravel Debugbar [Електронний ресурс] / Laravel Debugbar. – Режим доступу: https://laraveldebugbar.com/
41. Python – profile: Python Documentation [Електронний ресурс] / Python Documentation. – Режим доступу: https://docs.python.org/3/library/profile.html
42. Kernprof Documentation [Електронний ресурс] / Read the Docs. – Режим доступу: https://kernprof.readthedocs.io/en/latest/
43. memory-profiler [Електронний ресурс] / Python Package Index (PyPI). – Режим доступу: https://pypi.org/project/memory-profiler/
44. Python – logging: Logging facility for Python [Електронний ресурс] / Python Documentation. – Режим доступу: https://docs.python.org/3/library/logging.html
45. Python – time: Time access and conversions [Електронний ресурс] / Python Documentation. – Режим доступу: https://docs.python.org/uk/3.13/library/time.html

46. Python – timeit: Measure execution time of small code snippets [Електронний ресурс] / Python Documentation. – Режим доступу: https://docs.python.org/uk/3.13/library/timeit.html
47. psutil Documentation [Електронний ресурс] / Read the Docs. – Режим доступу: https://psutil.readthedocs.io/en/latest/
48. Locust – Scalable User Load Testing Tool [Електронний ресурс] / Locust. – Режим доступу: https://locust.io/
49. pytest-benchmark [Електронний ресурс] / Python Package Index (PyPI). – Режим доступу: https://pypi.org/project/pytest-benchmark/
50. aiohttp Documentation [Електронний ресурс] / aiohttp. – Режим доступу: https://docs.aiohttp.org/en/stable/
51. Sentry – Python Platform [Електронний ресурс] / Sentry Documentation. – Режим доступу: https://docs.sentry.io/platforms/python/
52. Werkzeug – General Helpers [Електронний ресурс] / Pallets Projects. – Режим доступу: https://werkzeug.palletsprojects.com/en/stable/utils/#general-helpers
53. MySQL Workbench [Електронний ресурс] / Oracle. – Режим доступу: https://www.mysql.com/products/workbench/
54. The N+1 Query Problem – The Silent Performance Killer [Електронний ресурс] / dev.to. – Режим доступу: https://dev.to/lovestaco/the-n1-query-problem-the-silent-performance-killer-2b1c

ДОДАТОК А

SQL-скрипт створення бази даних і таблиць web-додатку

```
CREATE DATABASE IF NOT EXISTS parcel_system;
USE parcel_system;

/* ========================
  АДМІНІСТРАТОРИ
  ======================== */
CREATE TABLE admins (
    admin_id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    login VARCHAR(50) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL
);

/* ========================
  ВІДДІЛЕННЯ
  ======================== */
CREATE TABLE branches (
    branch_id INT AUTO_INCREMENT PRIMARY KEY,
    address VARCHAR(255) NOT NULL,
    city VARCHAR(100) NOT NULL,
    phone CHAR(13) NOT NULL CHECK (phone LIKE '+380_________'),
    work_hours ENUM(
        '08:00-16:00',
        '09:00-17:00',
        '10:00-18:00',
        '11:00-19:00',
        '12:00-20:00'
    ) NOT NULL
);

/* ========================
  МАРШРУТИ
  ======================== */
CREATE TABLE routes (
    route_id INT AUTO_INCREMENT PRIMARY KEY,
    route_name VARCHAR(100) NOT NULL,
    length_km DECIMAL(6,2) NOT NULL CHECK (length_km > 0),
    start_branch_id INT NOT NULL,
    end_branch_id INT NOT NULL,

    CONSTRAINT fk_route_start_branch
        FOREIGN KEY (start_branch_id) REFERENCES branches(branch_id),

    CONSTRAINT fk_route_end_branch
        FOREIGN KEY (end_branch_id) REFERENCES branches(branch_id)
);
```

```
/* ============================
   КУР'ЄРИ
   ============================ */
CREATE TABLE couriers (
    courier_id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(100) NOT NULL,
    route_id INT NOT NULL,
    passport VARCHAR(20),
    phone CHAR(13) NOT NULL CHECK (phone LIKE '+380_________'),

    CONSTRAINT fk_courier_route
        FOREIGN KEY (route_id) REFERENCES routes(route_id)
);

/* ============================
   КЛІЄНТИ
   ============================ */
CREATE TABLE clients (
    client_id INT AUTO_INCREMENT PRIMARY KEY,
    rnokpp CHAR(10) NOT NULL UNIQUE CHECK (rnokpp REGEXP '^[0-9]{10}$'),
    full_name VARCHAR(100) NOT NULL,
    passport VARCHAR(20),
    phone CHAR(13) NOT NULL CHECK (phone LIKE '+380_________')
);

/* ============================
   ПОСИЛКИ
   ============================ */
CREATE TABLE parcels (
    parcel_id INT AUTO_INCREMENT PRIMARY KEY,
    weight_kg DECIMAL(6,2) NOT NULL CHECK (weight_kg > 0),
    parcel_type ENUM(
        'documents',
        'small',
        'medium',
        'large',
        'fragile'
    ) NOT NULL,
    status ENUM(
        'created',
        'in_transit',
        'at_branch',
        'delivered',
        'cancelled'
    ) NOT NULL DEFAULT 'created',
    delivery_cost DECIMAL(8,2) NOT NULL CHECK (delivery_cost >= 0),

    courier_id INT NOT NULL,
    sender_id INT NOT NULL,
    receiver_id INT NOT NULL,
    start_branch_id INT NOT NULL,
```

```
    end_branch_id INT NOT NULL,

    CONSTRAINT fk_parcel_courier
        FOREIGN KEY (courier_id) REFERENCES couriers(courier_id),

    CONSTRAINT fk_parcel_sender
        FOREIGN KEY (sender_id) REFERENCES clients(client_id),

    CONSTRAINT fk_parcel_receiver
        FOREIGN KEY (receiver_id) REFERENCES clients(client_id),

    CONSTRAINT fk_parcel_start_branch
        FOREIGN KEY (start_branch_id) REFERENCES branches(branch_id),

    CONSTRAINT fk_parcel_end_branch
        FOREIGN KEY (end_branch_id) REFERENCES branches(branch_id)
);

/* ========================
   ОПЕРАТОРИ
   ======================== */
CREATE TABLE operators (
    operator_id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(150) NOT NULL,
    phone CHAR(13) NOT NULL CHECK (phone LIKE '+380_________'),
    branch_id INT NOT NULL,
    login VARCHAR(50) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,

    CONSTRAINT fk_operator_branch
        FOREIGN KEY (branch_id) REFERENCES branches(branch_id)
);
```

ДОДАТОК Б

Сторінка авторизації (index.html):

```
<!DOCTYPE html>
<html lang="uk">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Login - WebApp</title>
   <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
   <header>
      <h1>Вхід у систему</h1>
   </header>

   <div class="container">
      <div class="card">
         <form id="loginForm" method="POST" action="{{ url_for('login') }}">
            <label for="login">Логін</label>
            <input type="text" id="login" name="login" required>

            <label for="password">Пароль</label>
            <input type="password" id="password" name="password" required>

            <button type="submit">Увійти</button>
         </form>
      </div>
   </div>

   <footer>
      <p>© 2026 Поштова компанія. Усі права захищені.</p>
   </footer>

</body>
</html>
```

Сторінка профілю (profile.html):

```
<!DOCTYPE html>
<html lang="uk">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Профіль</title>
   <link rel="stylesheet" href="{{ url_for('static', filename='css/profile.css') }}">
</head>
```

```
<body>

<header>
  <div class="header-container">
    <div class="logo"><h2>TopParcel📯</h2></div>

    <div class="burger" id="burger">
      <div></div>
      <div></div>
      <div></div>
    </div>

    <nav class="nav" id="nav">
      <a href="{{ url_for('dashboard') }}">Дашборд</a>
      <a href="{{ url_for('parcels') }}">Посилки</a>
      <a href="{{ url_for('clients') }}">Клієнти</a>
      <a href="{{ url_for('couriers') }}">Кур’єри</a>
      <a href="{{ url_for('routes') }}">Маршрути</a>
      <a href="{{ url_for('operators') }}">Оператори</a>
      <a href="{{ url_for('branches') }}">Відділення</a>
      <a href="{{ url_for('profile') }}">Профіль</a>
    </nav>

    <a href="{{ url_for('logout') }}" class="logout">Вийти</a>
  </div>
</header>

<main>
  <h1 class="page-title">Профіль користувача</h1>
  <div class="container center">
    <div class="card">
      <h2>Особиста інформація</h2>
      <form id="profileForm" method="POST" action="{{ url_for('profile') }}">

        <label for="full_name">ПІБ</label>
        <input type="text" id="full_name" name="full_name"
value="{{ user.full_name }}"
          {% if role != 'admin' %}readonly{% endif %}>

        <label for="login">Логін</label>
        <input type="text" id="login" name="login" value="{{ user.login }}"
          {% if role != 'admin' %}readonly{% endif %}>

        {% if role == 'operator' %}
        <label for="phone">Телефон</label>
        <input type="text" id="phone" name="phone" value="{{ user.phone }}"
readonly>

        <label for="branch">Відділення</label>
        <input type="text" id="branch" name="branch" value="{{ user.branch_id }}"
readonly>
        {% endif %}
```

```
                {% if role == 'admin' %}
                <label for="password">Пароль</label>
                <input type="password" id="password" name="password"
placeholder="********">
                {% endif %}

                {% if role == 'admin' %}
                <button type="submit" class="btn save">Зберегти зміни</button>
                {% endif %}
            </form>
        </div>
    </div>
</main>

<footer>
    <p>© 2026 Поштова компанія. Усі права захищені.</p>
</footer>

<script src="{{ url_for('static', filename='js/burger.js') }}"></script>
</body>
</html>
```

Сторінка дашборду (dashboard.html):

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dashboard</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/dashboard.css') }}">
</head>
<body>

<header>
    <div class="header-container">
        <div class="logo"><h2>TopParcel📬</h2></div>

        <div class="burger" id="burger">
            <div></div>
            <div></div>
            <div></div>
        </div>

        <nav class="nav" id="nav">
            <a href="{{ url_for('dashboard') }}">Дашборд</a>
            <a href="{{ url_for('parcels') }}">Посилки</a>
            <a href="{{ url_for('clients') }}">Клієнти</a>
            <a href="{{ url_for('couriers') }}">Кур’єри</a>
            <a href="{{ url_for('routes') }}">Маршрути</a>
```

```
            <a href="{{ url_for('operators') }}">Оператори</a>
            <a href="{{ url_for('branches') }}">Відділення</a>
            <a href="{{ url_for('profile') }}">Профіль</a>
        </nav>

        <a href="{{ url_for('logout') }}" class="logout">Вийти</a>
    </div>
</header>

<main>
    <h1 class="page-title">📊 Панель керування</h1>
    <div class="container dashboard">
        <div class="card stat">
            <h3>📦 Посилки</h3>
            <p>{{ parcels_count }}</p>
        </div>
        <div class="card stat">
            <h3>🚚 Кур’єри</h3>
            <p>{{ couriers_count }}</p>
        </div>
        <div class="card stat">
            <h3>🏢 Відділення</h3>
            <p>{{ branches_count }}</p>
        </div>
        <div class="card stat">
            <h3>👥 Клієнти</h3>
            <p>{{ clients_count }}</p>
        </div>
        <div class="card stat">
            <h3>🌍 Маршрути</h3>
            <p>{{ routes_count }}</p>
        </div>
        <div class="card stat">
            <h3>👧📋 Оператори</h3>
            <p>{{ operators_count }}</p>
        </div>
    </div>
</main>

<footer>
    <p>© 2026 Поштова компанія. Усі права захищені.</p>
</footer>

<script src="{{ url_for('static', filename='js/burger.js') }}"></script>
</body>
</html>
```

Сторінка перегляду інформації про клієнтів (clients.html):

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Клієнти</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='css/clients.css') }}">
</head>
<body>

<header>
  <div class="header-container">
    <div class="logo"><h2>TopParcel📯</h2></div>

    <div class="burger" id="burger">
      <div></div>
      <div></div>
      <div></div>
    </div>

    <nav class="nav" id="nav">
      <a href="{{ url_for('dashboard') }}">Дашборд</a>
      <a href="{{ url_for('parcels') }}">Посилки</a>
      <a href="{{ url_for('clients') }}">Клієнти</a>
      <a href="{{ url_for('couriers') }}">Кур’єри</a>
      <a href="{{ url_for('routes') }}">Маршрути</a>
      <a href="{{ url_for('operators') }}">Оператори</a>
      <a href="{{ url_for('branches') }}">Відділення</a>
      <a href="{{ url_for('profile') }}">Профіль</a>
    </nav>

    <a href="{{ url_for('logout') }}" class="logout">Вийти</a>
  </div>
</header>

<main class="content">
  <h1 class="page-title">👤 Клієнти</h1>

  {% if role == 'admin' %}
  <div class="filters-search">
    <div class="filters">
      <input type="text" id="phoneSearch" placeholder="🔍 Пошук по телефону..."
onkeyup="filterByPhone()">
    </div>
    <div class="search-container">
      <a href="{{ url_for('add_client') }}" class="btn add">+ Додати клієнта</a>
    </div>
  </div>
  {% endif %}

  <div class="table-wrapper">
    <table class="parcels-table" id="clientsTable">
```

```
                <thead>
                  <tr>
                    <th>ID</th>
                    <th>РНОКПП</th>
                    <th>ПІБ</th>
                    <th>Паспорт</th>
                    <th>Телефон</th>
                    <th>Дії</th>
                  </tr>
                </thead>
                <tbody>
                {% for c in clients %}
                  <tr>
                    <td>{{ c['client_id'] }}</td>
                    <td>{{ c['rnokpp'] }}</td>
                    <td>{{ c['full_name'] }}</td>
                    <td>{{ c['passport'] }}</td>
                    <td>{{ c['phone'] }}</td>
                    <td class="actions">
                      <a href="{{ url_for('edit_client', cl_id=c['client_id']) }}" class="btn
edit">✏</a>

                      {% if role == 'admin' %}
                      <a href="{{ url_for('delete_client', cl_id=c['client_id']) }}"
                      onclick="return confirm('Видалити клієнта?')"
                      class="btn delete">🗑</a>
                      {% endif %}
                    </td>
                  </tr>
                {% endfor %}
                </tbody>
              </table>
            </div>
          </main>

          <footer>
            <p>© 2026 Поштова компанія. Усі права захищені.</p>
          </footer>

          <script src="{{ url_for('static', filename='js/burger.js') }}"></script>
          <script>
          function filterByPhone() {
            const input = document.getElementById('phoneSearch');
            const filter = input.value.toLowerCase();
            const table = document.getElementById('clientsTable');
            const rows = table.getElementsByTagName('tr');

            for (let i = 1; i < rows.length; i++) {
              const td = rows[i].getElementsByTagName('td')[4]; // стовпець Телефон
              if (td) {
                const txtValue = td.textContent || td.innerText;
                rows[i].style.display = txtValue.toLowerCase().includes(filter) ? " : 'none';
```

```
      }
    }
}
</script>

</body>
</html>
```

Сторінка форми для редагування або додавання клієнтів (client_form.html):

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>{{ 'Додати' if action=='add' else 'Редагувати' }} клієнта</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='css/form.css') }}">
</head>
<body>

<header>
  <div class="header-container">
    <div class="logo"><h2>TopParcel📯</h2></div>

    <div class="burger" id="burger">
      <div></div>
      <div></div>
      <div></div>
    </div>

    <nav class="nav" id="nav">
      <a href="{{ url_for('dashboard') }}">Дашборд</a>
      <a href="{{ url_for('parcels') }}">Посилки</a>
      <a href="{{ url_for('clients') }}">Клієнти</a>
      <a href="{{ url_for('couriers') }}">Кур’єри</a>
      <a href="{{ url_for('routes') }}">Маршрути</a>
      <a href="{{ url_for('operators') }}">Оператори</a>
      <a href="{{ url_for('branches') }}">Відділення</a>
      <a href="{{ url_for('profile') }}">Профіль</a>
    </nav>

    <a href="{{ url_for('logout') }}" class="logout">Вийти</a>
  </div>
</header>

<main class="content">
  <h1 class="page-title">{{ 'Додати' if action=='add' else 'Редагувати' }} клієнта</h1>

  <form method="post" class="parcel-form">
```

```
            <label>РНОКПП:
              <input type="text" name="rnokpp" required
                value="{{ client['rnokpp'] if client else '' }}">
            </label>

            <label>ПІБ:
              <input type="text" name="full_name" required
               value="{{ client['full_name'] if client else '' }}">
            </label>

            <label>Паспорт:
              <input type="text" name="passport"
              value="{{ client['passport'] if client else '' }}">
            </label>

            <label>Телефон:
              <input type="text" name="phone" required
              value="{{ client['phone'] if client else '' }}">
            </label>

            <button type="submit" class="btn add">{{ 'Додати' if action=='add' else
'Зберегти' }}</button>
            <a href="{{ url_for('clients') }}" class="btn delete">Відмінити</a>
          </form>
        </main>

        <footer>
          <p>© 2026 Поштова компанія. Усі права захищені.</p>
        </footer>

        <script src="{{ url_for('static', filename='js/burger.js') }}"></script>
        </body>
        </html>
```

Загальний файл стилів (style.css):

```
        body {
          font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
          margin: 0;
          padding: 0;
          background-color: #e8f0f2;
          color: #333;
          min-height: 100vh;
          display: flex;
          flex-direction: column;
        }

        header {
          background: linear-gradient(90deg, #ff6f61, #ff8578);
          color: #fff;
          padding: 25px 15px;
```

```
    text-align: center;
    box-shadow: 0 5px 15px rgba(0,0,0,0.2);
}

header h1 {
    margin: 0;
    font-size: 2em;
    letter-spacing: 1px;
}

.page-title {
    text-align: center;
    font-size: 1.8em;
    color: #2c3e50;
    margin: 25px 0 15px 0;
}

.container {
    flex: 1;
    display: flex;
    justify-content: center;
    align-items: center;
    padding: 20px;
    width: 100%;
    box-sizing: border-box;
}

.card {
    background-color: #ffffff;
    padding: 25px 20px;
    border-radius: 20px;
    box-shadow: 0 10px 25px rgba(0,0,0,0.15);
    width: 100%;
    max-width: 400px;
    min-width: 200px;
    margin: 0 auto;
    transition: transform 0.3s ease, box-shadow 0.3s ease;
}

.card:hover {
    transform: translateY(-3px);
    box-shadow: 0 12px 28px rgba(0,0,0,0.2);
}

.card h2 {
    margin-top: 0;
    color: #2c3e50;
}

label {
    display: block;
    margin-top: 15px;
```

```
    font-weight: 600;
    color: #2c3e50;
}

input, select {
    padding: 12px;
    width: 100%;
    margin-top: 8px;
    border-radius: 10px;
    border: 1px solid #ccc;
    box-sizing: border-box;
    transition: border-color 0.3s, box-shadow 0.3s;
}

input:focus {
    border-color: #ff6f61;
    box-shadow: 0 0 5px rgba(255,111,97,0.5);
    outline: none;
}

button {
    background-color: #ff6f61;
    color: white;
    padding: 12px;
    margin-top: 20px;
    border: none;
    border-radius: 12px;
    width: 100%;
    font-size: 1em;
    cursor: pointer;
    transition: background-color 0.3s ease, transform 0.2s ease;
}

button:hover {
    background-color: #e55b50;
    transform: scale(1.03);
}

footer {
    background-color: #2f3e46;
    color: #ffffff;
    text-align: center;
    padding: 20px 0;
    font-size: 0.85em;
    margin-top: auto;
}

@media (max-width: 800px) {
    .card {
        width: 90%;
        padding: 20px 15px;
    }
```

```
    .page-title {
        font-size: 1.6em;
        margin: 20px 0 12px 0;
    }
}

@media (max-width: 480px) {
    .container {
        padding: 10px;
    }

    .card {
        width: 100%;
        padding: 15px 10px;
        min-width: 0;
    }

    header h1 {
        font-size: 1.5em;
    }

    label {
        font-size: 0.9em;
    }

    input, button {
        font-size: 0.9em;
        padding: 10px;
    }

    .page-title {
        font-size: 1.4em;
        margin: 15px 0 10px 0;
    }
}
```

Файл стилів для форм (form.css):

```
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #e8f0f2;
    color: #333;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
}

header {
```

```
    background: linear-gradient(90deg, #ff6f61, #ff8578);
    color: #fff;
    padding: 15px 20px;
    box-shadow: 0 5px 15px rgba(0,0,0,0.2);
}

.header-container {
    display: flex;
    align-items: center;
    justify-content: space-between;
    max-width: 1200px;
    margin: 0 auto;
    position: relative;
}

.logo h2 { margin: 0; font-size: 1.5em; }

.nav {
    display: flex;
    gap: 15px;
    flex-wrap: wrap;
    justify-content: center;
    flex: 2;
}

.nav a {
    color: #fff;
    text-decoration: none;
    padding: 6px 12px;
    border-radius: 8px;
    transition: background 0.3s;
}

.nav a:hover { background-color: rgba(255,255,255,0.3); }

.logout {
    margin-left: auto;
    color: #fff;
    text-decoration: none;
    padding: 6px 12px;
    border-radius: 8px;
    transition: background 0.3s;
}

.logout:hover { background-color: rgba(255,255,255,0.3); }

.page-title {
    text-align: center;
    font-size: 1.8em;
    color: #2c3e50;
    margin: 25px 0 15px 0;
}
```

```
.burger {
   display: none;
   flex-direction: column;
   cursor: pointer;
   gap: 5px;
}

.burger div {
   width: 25px;
   height: 3px;
   background-color: #fff;
}

.content {
   flex: 1;
   width: 100%;
   padding: 15px;
   display: flex;
   flex-direction: column;
   box-sizing: border-box;
}

.parcel-form {
   max-width: 600px;
   margin: 0 auto;
   background: #fff;
   padding: 25px 30px;
   border-radius: 12px;
   box-shadow: 0 8px 20px rgba(0,0,0,0.1);
   display: flex;
   flex-direction: column;
   gap: 15px;
}

.parcel-form label {
   display: flex;
   flex-direction: column;
   font-weight: 500;
   font-size: 0.95em;
   color: #2c3e50;
}

.parcel-form input,
.parcel-form select {
   margin-top: 6px;
   padding: 8px 12px;
   border: 1px solid #ccc;
   border-radius: 8px;
   font-size: 0.95em;
   transition: border 0.3s, box-shadow 0.3s;
}
```

```
.parcel-form input:focus,
.parcel-form select:focus {
   outline: none;
   border-color: #ff6f61;
   box-shadow: 0 0 5px rgba(255,111,97,0.5);
}

.parcel-form button,
.parcel-form a.btn {
   padding: 10px 18px;
   border-radius: 8px;
   font-size: 0.95em;
   text-decoration: none;
   border: none;
   cursor: pointer;
   text-align: center;
   transition: all 0.3s;
}

.parcel-form button.add {
   background-color: #2ecc71;
   color: #fff;
}

.parcel-form button.add:hover {
   background-color: #27ae60;
}

.parcel-form a.btn.delete {
   background-color: #e74c3c;
   color: #fff;
   display: inline-block;
   margin-top: 5px;
}

.parcel-form a.btn.delete:hover {
   background-color: #c0392b;
}

footer {
   background-color: #2f3e46;
   color: #fff;
   text-align: center;
   padding: 20px 0;
   font-size: 0.85em;
   margin-top: auto;
}

@media (max-width: 960px) {
   .burger { display: flex; }
```

```
    .nav {
        position: absolute;
        top: 65px;
        left: 0;
        width: 100%;
        flex-direction: column;
        background: linear-gradient(90deg, #ff6f61, #ff8578);
        display: none;
        z-index: 1000;
        padding: 10px 0;
    }

    .nav.active { display: flex; }

    .nav a { padding: 12px 20px; border-bottom: 1px solid rgba(255,255,255,0.3); }
    .logout { padding: 12px 20px; margin: 0; border-radius: 0; }

    .parcel-form {
        width: 90%;
        padding: 20px;
        margin-left: auto;
        margin-right: auto;
        box-sizing: border-box;
    }


}

@media (max-width: 480px) {
    .page-title { font-size: 1.4em; margin: 20px 0; }

    .parcel-form {
        width: 95%;
        padding: 15px;
        margin-left: auto;
        margin-right: auto;
        box-sizing: border-box;
    }
}
```

Файл стилів для клієнтів (clients.css):

```
body {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    margin: 0;
    padding: 0;
    background-color: #e8f0f2;
    color: #333;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
```

```
}

header {
    background: linear-gradient(90deg, #ff6f61, #ff8578);
    color: #fff;
    padding: 15px 20px;
    box-shadow: 0 5px 15px rgba(0,0,0,0.2);
}

.header-container {
    display: flex;
    align-items: center;
    justify-content: space-between;
    max-width: 1200px;
    margin: 0 auto;
    position: relative;
}

.logo h2 { margin: 0; font-size: 1.5em; }

.nav {
    display: flex;
    gap: 15px;
    flex-wrap: wrap;
    justify-content: center;
    flex: 2;
}

.nav a {
    color: #fff;
    text-decoration: none;
    padding: 6px 12px;
    border-radius: 8px;
    transition: background 0.3s;
}

.nav a:hover { background-color: rgba(255,255,255,0.3); }

.logout {
    margin-left: auto;
    color: #fff;
    text-decoration: none;
    padding: 6px 12px;
    border-radius: 8px;
    transition: background 0.3s;
}

.logout:hover { background-color: rgba(255,255,255,0.3); }

.page-title {
    text-align: center;
    font-size: 1.8em;
```

```
    color: #2c3e50;
    margin: 25px 0 15px 0;
}

.burger {
    display: none;
    flex-direction: column;
    cursor: pointer;
    gap: 5px;
}

.burger div {
    width: 25px;
    height: 3px;
    background-color: #fff;
}

.content {
    flex: 1;
    width: 100%;
    padding: 15px;
    display: flex;
    flex-direction: column;
    align-items: center;
    box-sizing: border-box;
}

.filters-search {
    display: flex;
    justify-content: space-between;
    align-items: center;
    width: 100%;
    margin-bottom: 15px;
    flex-wrap: wrap;
    gap: 10px;
}

.filters-search .filters {
    flex: 1 1 auto;
    display: flex;
    justify-content: flex-start;
}

.filters-search .search-container {
    flex: 1 1 auto;
    display: flex;
    justify-content: flex-end;
}

.filters-search input {
    padding: 6px 12px;
    border-radius: 8px;
```

```
    border: 1px solid #ccc;
    font-size: 0.95em;
}

.table-wrapper {
    flex: 1;
    width: 100%;
    overflow-x: auto;
    overflow-y: auto;
}

.parcels-table {
    width: 100%;
    min-width: 900px;
    border-collapse: collapse;
    background: #fff;
    border-radius: 10px;
    overflow: hidden;
    table-layout: fixed;
}

.parcels-table th,
.parcels-table td {
    padding: 12px;
    border-bottom: 1px solid #ddd;
    text-align: center;
    word-wrap: break-word;
}

.parcels-table th {
    background: #ff6f61;
    color: #fff;
    position: sticky;
    top: 0;
    z-index: 1;
}

.actions {
    display: flex;
    justify-content: center;
    gap: 6px;
}

.btn {
    padding: 6px 10px;
    border-radius: 6px;
    text-decoration: none;
    color: #fff;
    font-size: 0.85em;
}

.btn.add { background: #2ecc71; }
```

```
.btn.edit { background: #3498db; }
.btn.delete { background: #e74c3c; }

footer {
   background-color: #2f3e46;
   color: #fff;
   text-align: center;
   padding: 20px 0;
   font-size: 0.85em;
}

@media (max-width: 960px) {
   .burger { display: flex; }

   .nav {
      position: absolute;
      top: 65px;
      left: 0;
      width: 100%;
      flex-direction: column;
      background: linear-gradient(90deg, #ff6f61, #ff8578);
      display: none;
      z-index: 1000;
      padding: 10px 0;
   }

   .nav.active { display: flex; }

   .nav a { padding: 12px 20px; border-bottom: 1px solid rgba(255,255,255,0.3); }
   .logout { padding: 12px 20px; margin: 0; border-radius: 0; }
}

@media (max-width: 480px) {
   .filters-search {
      flex-direction: column;
      gap: 10px;
   }

   .filters-search .filters,
   .filters-search .search-container {
      width: 100%;
      justify-content: flex-start;
   }

   .filters-search input {
      width: 100%;
   }

   .parcels-table { min-width: 600px; }
}
```

Скрипт для бургер-меню (burger.js):

```
document.addEventListener('DOMContentLoaded', () => {
    const burger = document.getElementById('burger');
    const nav = document.getElementById('nav');
    const logout = document.querySelector('.logout');

    const logoutMobile = document.createElement('a');
    logoutMobile.href = "index.html";
    logoutMobile.className = "logout-mobile";
    logoutMobile.textContent = "Вийти";

    function updateLogoutPlacement() {
        if (window.innerWidth < 960) {
            logout.style.display = 'none';
            if (!nav.contains(logoutMobile)) {
                nav.appendChild(logoutMobile);
            }
        } else {
            logout.style.display = 'block';
            if (nav.contains(logoutMobile)) {
                nav.removeChild(logoutMobile);
            }
        }
    }

    updateLogoutPlacement();
    window.addEventListener('resize', updateLogoutPlacement);

    burger.addEventListener('click', () => {
        nav.classList.toggle('active');
    });
});
```

ДОДАТОК В

Фрагмент реалізації серверної логіки для авторизації та автентифікації користувачів

```
@app.route("/", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        login_input = request.form["login"].strip()
        password_input = request.form["password"].strip()

        user = None
        role = None

        try:
            with db.conn.cursor(dictionary=True) as cursor:
            sql_admin = "SELECT admin_id AS user_id, full_name, password_hash FROM
admins WHERE login = %s"
                cursor.execute(sql_admin, (login_input,))
                user = cursor.fetchone()
                role = "admin"

                if not user:
                    sql_operator = "SELECT operator_id AS user_id, full_name, password_hash
FROM operators WHERE login = %s"
                    cursor.execute(sql_operator, (login_input,))
                    user = cursor.fetchone()
                    role = "operator"

        except Exception as e:
            flash("Помилка під час перевірки логіну. Спробуйте пізніше.", "error")
            print("Login DB error:", e)
            return redirect(url_for("login"))

        if user and check_password_hash(user["password_hash"], password_input):
            session.clear()
            session["user_id"] = user["user_id"]
            session["role"] = role
            session["full_name"] = user["full_name"]
            return redirect(url_for("dashboard"))

        flash("Невірний логін або пароль", "error")
        return redirect(url_for("login"))

    return render_template("index.html")
```

ДОДАТОК Г

Основний файл підключення до бази даних MySQL

```
import os
from dotenv import load_dotenv
import mysql.connector
from mysql.connector import Error

load_dotenv()  # завантажує .env

class DBConnection:
    def __init__(self):
        self.host = os.getenv("DB_HOST")
        self.user = os.getenv("DB_USER")
        self.password = os.getenv("DB_PASSWORD")
        self.database = os.getenv("DB_NAME")
        self.conn = None

    def connect(self):
        """Підключаємось до бази даних і перевіряємо з'єднання"""
        try:
            self.conn = mysql.connector.connect(
                host=self.host,
                user=self.user,
                password=self.password,
                database=self.database
            )
            if self.conn.is_connected():
                print(f"[INFO] Підключення до {self.database} успішне! MySQL версія: {self.conn.get_server_info()}")
            else:
                print("[WARN] Підключення встановити не вдалося.")
        except Error as e:
            print(f"[ERROR] Помилка підключення до MySQL: {e}")

    def close(self):
        """Закриваємо підключення"""
        if self.conn and self.conn.is_connected():
            self.conn.close()
            print("[INFO] Підключення до MySQL закрито.")
```

Приклад файлу підключення до бази даних за допомогою SQLAlchemy

```
from sqlalchemy import create_engine, Column, Integer, String, DECIMAL, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker
```

```
Base = declarative_base()

engine = create_engine(
    "mysql+pymysql://username:password@127.0.0.1/my_database",
    echo=True,    # Вивід SQL-запитів для дебагу
    future=True
)

SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False)

class Courier(Base):
    __tablename__ = "couriers"

    courier_id = Column(Integer, primary_key=True, autoincrement=True)
    full_name = Column(String(100), nullable=False)
    route_id = Column(Integer, ForeignKey("routes.route_id"), nullable=False)
    passport = Column(String(20))
    phone = Column(String(13), nullable=False)

class Route(Base):
    __tablename__ = "routes"

    route_id = Column(Integer, primary_key=True, autoincrement=True)
    route_name = Column(String(100), nullable=False)
    length_km = Column(DECIMAL(6,2), nullable=False)
    start_branch_id = Column(Integer, nullable=False)
    end_branch_id = Column(Integer, nullable=False)


if __name__ == "__main__":
    Base.metadata.create_all(bind=engine)
    print("Таблиці створені або вже існують")
```

ДОДАТОК Д

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції SELECT (автентифікація користувача)

```
@app.route("/", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        login_input = request.form["login"].strip()
        password_input = request.form["password"].strip()

        NUM_RUNS = 1000  # для точних вимірів CPU/MEM
        process = psutil.Process(os.getpid())

        def measure_performance(func, login_input):
            elapsed_times = []
            latencies = []
            tps_list = []
            cpu_used_list = []
            mem_used_list = []

            for _ in range(NUM_RUNS):
                mem_before = process.memory_info().rss
                cpu_before = process.cpu_percent(interval=0.01)

                start = time.perf_counter()
                first_byte_time = [None]

                user, role = func(login_input, first_byte_time)
                end = time.perf_counter()

                mem_after = process.memory_info().rss
                cpu_after = process.cpu_percent(interval=0.01)

                elapsed = end - start
                tps = 1 / elapsed if elapsed > 0 else float('inf')
                latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
                cpu_used = max(cpu_after - cpu_before, 0)
                mem_used = max(mem_after - mem_before, 0)

                elapsed_times.append(elapsed)
                latencies.append(latency)
                tps_list.append(tps)
                cpu_used_list.append(cpu_used)
                mem_used_list.append(mem_used)

            avg_elapsed = sum(elapsed_times)/NUM_RUNS
```

```
            avg_latency = sum(latencies)/NUM_RUNS
            avg_tps = sum(tps_list)/NUM_RUNS
            avg_cpu = sum(cpu_used_list)/NUM_RUNS
            avg_mem = sum(mem_used_list)/NUM_RUNS

            return user, role, avg_elapsed, avg_tps, avg_latency, avg_cpu, avg_mem


        def raw_login(login_input, first_byte_time):
            user = None
            role = None
            with db.conn.cursor(dictionary=True) as cursor:
            sql_admin = f"SELECT admin_id AS user_id, full_name, password_hash FROM
admins WHERE login = '{login_input}'"
                cursor.execute(sql_admin)
                row = cursor.fetchone()
                if first_byte_time[0] is None and row:
                    first_byte_time[0] = time.perf_counter()
                if row:
                    user = row
                    role = "admin"
                else:
                sql_operator = f"SELECT operator_id AS user_id, full_name, password_hash
FROM operators WHERE login = '{login_input}'"
                    cursor.execute(sql_operator)
                    row = cursor.fetchone()
                    if first_byte_time[0] is None and row:
                        first_byte_time[0] = time.perf_counter()
                    if row:
                        user = row
                        role = "operator"
            return user, role


        def param_login(login_input, first_byte_time):
            user = None
            role = None
            with db.conn.cursor(dictionary=True) as cursor:
            sql_admin = "SELECT admin_id AS user_id, full_name, password_hash FROM
admins WHERE login = %s"
                cursor.execute(sql_admin, (login_input,))
                row = cursor.fetchone()
                if first_byte_time[0] is None and row:
                    first_byte_time[0] = time.perf_counter()
                if row:
                    user = row
                    role = "admin"
                else:
                    sql_operator = "SELECT operator_id AS user_id, full_name, password_hash
FROM operators WHERE login = %s"
                    cursor.execute(sql_operator, (login_input,))
                    row = cursor.fetchone()
```

```
                    if first_byte_time[0] is None and row:
                        first_byte_time[0] = time.perf_counter()
                    if row:
                        user = row
                        role = "operator"
                return user, role


            def orm_login(login_input, first_byte_time):
                user = None
                role = None
                with db.conn.cursor(dictionary=True) as cursor:
                sql_admin = "SELECT admin_id AS user_id, full_name, password_hash FROM
admins WHERE login = %s"
                    cursor.execute(sql_admin, (login_input,))
                    row = cursor.fetchone()
                    if first_byte_time[0] is None and row:
                        first_byte_time[0] = time.perf_counter()
                    if row:
                        user = Admin(row)
                        role = "admin"
                    else:
                        sql_operator = "SELECT operator_id AS user_id, full_name, password_hash
FROM operators WHERE login = %s"
                        cursor.execute(sql_operator, (login_input,))
                        row = cursor.fetchone()
                        if first_byte_time[0] is None and row:
                            first_byte_time[0] = time.perf_counter()
                        if row:
                            user = Operator(row)
                            role = "operator"
                return user, role

            # ======================== Замір ========================
            methods = [("RAW SQL", raw_login), ("PARAM SQL", param_login), ("ORM",
orm_login)]
            results = []

            for name, func in methods:
                user, role, avg_time, avg_tps, avg_latency, avg_cpu, avg_mem =
measure_performance(func, login_input)
                results.append({
                    "method": name,
                    "user": user,
                    "role": role,
                    "avg_time": avg_time,
                    "avg_tps": avg_tps,
                    "latency": avg_latency,
                    "cpu": avg_cpu,
                    "mem": avg_mem
                })
```

```
            # ====================== Друк ======================
            print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} | {'CPU%':<6} | {'MEM(bytes)':<10}")
            print("-"*75)
            for r in results:
                print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} | {r['latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

            actual_user = results[1]["user"]  # PARAM як основний
            actual_role = results[1]["role"]
            password_hash = getattr(actual_user, "password_hash", None) or actual_user["password_hash"]

            if actual_user and check_password_hash(password_hash, password_input):
                session.clear()
                session["user_id"] = getattr(actual_user, "user_id", None) or actual_user["user_id"]
                session["role"] = actual_role
                session["full_name"] = getattr(actual_user, "full_name", None) or actual_user["full_name"]
                return redirect(url_for("dashboard"))

            flash("Невірний логін або пароль", "error")
            return redirect(url_for("login"))

        return render_template("index.html")
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції INSERT (додавання кур'єра)

```
@app.route("/couriers/add", methods=["GET", "POST"])
def add_courier():
    if session.get("role") != "admin":
        return redirect(url_for("couriers"))

    db = DBConnection()
    db.connect()

    if request.method == "POST":
        # Дані з форми
        full_name = request.form["full_name"]
        route_id = request.form["route_id"]
        passport = request.form.get("passport", "")
        phone = request.form["phone"]

        courier_data = {
            "full_name": full_name,
            "route_id": route_id,
            "passport": passport,
            "phone": phone
```

```
}

NUM_RUNS = 1000
process = psutil.Process(os.getpid())

def measure_performance(func, courier_data):
    elapsed_times = []
    latencies = []
    tps_list = []
    sql_counts = []
    cpu_used_list = []
    mem_used_list = []

    for _ in range(NUM_RUNS):
        mem_before = process.memory_info().rss
        cpu_before = process.cpu_percent(interval=0.01)

        start = time.perf_counter()
        sql_count_ref = [0]
        first_byte_time = [None]

        func(courier_data, sql_count_ref, first_byte_time)

        end = time.perf_counter()
        mem_after = process.memory_info().rss
        cpu_after = process.cpu_percent(interval=0.01)

        elapsed = end - start
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        latencies.append(latency)
        tps_list.append(tps)
        sql_counts.append(sql_count_ref[0])
        cpu_used_list.append(cpu_used)
        mem_used_list.append(mem_used)

    return {
        "avg_time": sum(elapsed_times)/NUM_RUNS,
        "avg_latency": sum(latencies)/NUM_RUNS,
        "avg_tps": sum(tps_list)/NUM_RUNS,
        "avg_cpu": sum(cpu_used_list)/NUM_RUNS,
        "avg_mem": sum(mem_used_list)/NUM_RUNS,
        "avg_sql_count": sum(sql_counts)//NUM_RUNS
    }


def raw_insert(courier, sql_count_ref, first_byte_time):
    with db.conn.cursor() as cursor:
```

```
                sql = f"""
                    INSERT INTO couriers (full_name, route_id, passport, phone)
                VALUES ('{courier['full_name']}', {courier['route_id']}, '{courier['passport']}',
'{courier['phone']}')
                """
                cursor.execute(sql)
                sql_count_ref[0] += 1
                if first_byte_time[0] is None:
                    first_byte_time[0] = time.perf_counter()
                db.conn.commit()


            def param_insert(courier, sql_count_ref, first_byte_time):
                with db.conn.cursor() as cursor:
                    sql = """
                        INSERT INTO couriers (full_name, route_id, passport, phone)
                        VALUES (%s, %s, %s, %s)
                    """
                    cursor.execute(sql, (courier['full_name'], courier['route_id'], courier['passport'],
courier['phone']))
                    sql_count_ref[0] += 1
                    if first_byte_time[0] is None:
                        first_byte_time[0] = time.perf_counter()
                    db.conn.commit()


            class Courier:
                def __init__(self, full_name, route_id, passport, phone):
                    self.full_name = full_name
                    self.route_id = route_id
                    self.passport = passport
                    self.phone = phone

            def orm_insert(courier, sql_count_ref, first_byte_time):
                c = Courier(courier['full_name'], courier['route_id'], courier['passport'],
courier['phone'])
                with db.conn.cursor() as cursor:
                    sql = """
                        INSERT INTO couriers (full_name, route_id, passport, phone)
                        VALUES (%s, %s, %s, %s)
                    """
                    cursor.execute(sql, (c.full_name, c.route_id, c.passport, c.phone))
                    sql_count_ref[0] += 1
                    if first_byte_time[0] is None:
                        first_byte_time[0] = time.perf_counter()
                    db.conn.commit()

            methods = [("RAW SQL", raw_insert), ("PARAM SQL", param_insert), ("ORM",
orm_insert)]
            results = {}

            for name, func in methods:
```

```
            results[name] = measure_performance(func, courier_data)


        print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} | {'CPU%':<6} | {'MEM(bytes)':<10} | {'SQL#':<5}")
        print("-"*95)
        for method, data in results.items():
            print(f"{method:<10} | {data['avg_time']:<12.6f} | {data['avg_tps']:<8.2f} | "
              f"{data['avg_latency']:<12.6f} | {data['avg_cpu']:<6.2f} | {data['avg_mem']:<10} | {data['avg_sql_count']:<5}")

        db.close()
        return redirect(url_for("couriers"))

    try:
        with db.conn.cursor(dictionary=True) as cursor:
            cursor.execute("SELECT route_id, route_name FROM routes")
            routes = cursor.fetchall()
    finally:
        db.close()

    return render_template("courier_form.html", action="add", courier=None, routes=routes, role="admin")
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції SELECT (отримання списку кур’єрів)

```
@app.route("/couriers")
def couriers():
    db = DBConnection()
    db.connect()
    role = session.get("role", "admin")
    couriers_list = []
    NUM_RUNS = 1000  # кількість запусків для замірів
    process = psutil.Process(os.getpid())

    def measure_performance(func):
        elapsed_times = []
        tps_list = []
        latencies = []
        cpu_used_list = []
        mem_used_list = []

        result_data = []

        for _ in range(NUM_RUNS):
            mem_before = process.memory_info().rss
            cpu_before = process.cpu_percent(interval=0.01)
```

```
        start = time.perf_counter()
        first_byte_time = [None]

        data = func(first_byte_time)  # для latency
        end = time.perf_counter()

        mem_after = process.memory_info().rss
        cpu_after = process.cpu_percent(interval=0.01)

        elapsed = end - start
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        tps_list.append(tps)
        latencies.append(latency)
        cpu_used_list.append(cpu_used)
        mem_used_list.append(mem_used)

        if not result_data:
            result_data = data

    avg_elapsed = sum(elapsed_times) / NUM_RUNS
    avg_tps = sum(tps_list) / NUM_RUNS
    avg_latency = sum(latencies) / NUM_RUNS
    avg_cpu = sum(cpu_used_list) / NUM_RUNS
    avg_mem = sum(mem_used_list) / NUM_RUNS

    return result_data, avg_elapsed, avg_tps, avg_latency, avg_cpu, avg_mem


def raw_method(first_byte_time):
    with db.conn.cursor(dictionary=True) as cursor:
        cursor.execute("""
            SELECT c.courier_id, c.full_name, r.route_name, c.passport, c.phone
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
        """)
        if first_byte_time[0] is None:
            first_byte_time[0] = time.perf_counter()
        return cursor.fetchall()


def param_method(first_byte_time):
    with db.conn.cursor(dictionary=True) as cursor:
        sql = """
            SELECT c.courier_id, c.full_name, r.route_name, c.passport, c.phone
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
```

```
            """
            cursor.execute(sql)
            if first_byte_time[0] is None:
                first_byte_time[0] = time.perf_counter()
            return cursor.fetchall()


        class CourierORM:
            def __init__(self, row):
                self.courier_id = row["courier_id"]
                self.full_name = row["full_name"]
                self.route_name = row["route_name"]
                self.passport = row["passport"]
                self.phone = row["phone"]

        def orm_method(first_byte_time):
            with db.conn.cursor(dictionary=True) as cursor:
                sql = """
                    SELECT c.courier_id, c.full_name, r.route_name, c.passport, c.phone
                    FROM couriers c
                    JOIN routes r ON c.route_id = r.route_id
                """
                cursor.execute(sql)
                rows = cursor.fetchall()
                if first_byte_time[0] is None:
                    first_byte_time[0] = time.perf_counter()
                return [CourierORM(r) for r in rows]

        # ======================== Замір ========================
        methods = [("RAW SQL", raw_method), ("PARAM SQL", param_method), ("ORM",
orm_method)]
        measurement_results = []

        for name, func in methods:
                    data, avg_time, avg_tps, avg_latency, avg_cpu, avg_mem =
measure_performance(func)
            measurement_results.append({
                "method": name,
                "avg_time": avg_time,
                "avg_tps": avg_tps,
                "avg_latency": avg_latency,
                "cpu": avg_cpu,
                "mem": avg_mem
            })

            if name == "PARAM SQL":
                couriers_list = data

        # ======================== Вивід у консоль ========================
          print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} |
{'CPU%':<6} | {'MEM(bytes)':<10}")
        print("-"*80)
```

```
        for r in measurement_results:
                print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} | {r['avg_latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

        db.close()
            return render_template("couriers.html", couriers=couriers_list, role=role, measurements=measurement_results)
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції UPDATE (редагування даних кур'єра)

```
@app.route("/couriers/edit/<int:courier_id>", methods=["GET", "POST"])
def edit_courier(courier_id):
    if session.get("role") != "admin":
        return redirect(url_for("couriers"))

    db = DBConnection()
    db.connect()

    if request.method == "POST":
        full_name = request.form["full_name"]
        route_id = request.form["route_id"]
        passport = request.form.get("passport", "")
        phone = request.form["phone"]

        courier_data = {
            "courier_id": courier_id,
            "full_name": full_name,
            "route_id": route_id,
            "passport": passport,
            "phone": phone
        }

        NUM_RUNS = 1000
        process = psutil.Process(os.getpid())

        def measure_performance(func, courier_data):
            elapsed_times = []
            latencies = []
            tps_list = []
            sql_counts = []
            cpu_used_list = []
            mem_used_list = []

            for _ in range(NUM_RUNS):
                mem_before = process.memory_info().rss
                cpu_before = process.cpu_percent(interval=0.01)
```

```
        start = time.perf_counter()
        sql_count_ref = [0]
        first_byte_time = [None]

        func(courier_data, sql_count_ref, first_byte_time)

        end = time.perf_counter()
        mem_after = process.memory_info().rss
        cpu_after = process.cpu_percent(interval=0.01)

        elapsed = end - start
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        latencies.append(latency)
        tps_list.append(tps)
        sql_counts.append(sql_count_ref[0])
        cpu_used_list.append(cpu_used)
        mem_used_list.append(mem_used)

    return {
        "avg_time": sum(elapsed_times)/NUM_RUNS,
        "avg_latency": sum(latencies)/NUM_RUNS,
        "avg_tps": sum(tps_list)/NUM_RUNS,
        "avg_cpu": sum(cpu_used_list)/NUM_RUNS,
        "avg_mem": sum(mem_used_list)/NUM_RUNS,
        "avg_sql_count": sum(sql_counts)//NUM_RUNS
    }


def raw_update(courier, sql_count_ref, first_byte_time):
    with db.conn.cursor() as cursor:
        sql = f"""
            UPDATE couriers
            SET full_name='{courier['full_name']}', route_id={courier['route_id']},
                passport='{courier['passport']}', phone='{courier['phone']}'
            WHERE courier_id={courier['courier_id']}
        """
        cursor.execute(sql)
        sql_count_ref[0] += 1
        if first_byte_time[0] is None:
            first_byte_time[0] = time.perf_counter()
        db.conn.commit()


def param_update(courier, sql_count_ref, first_byte_time):
    with db.conn.cursor() as cursor:
        sql = """
            UPDATE couriers
```

```
                    SET full_name=%s, route_id=%s, passport=%s, phone=%s
                    WHERE courier_id=%s
                """
                cursor.execute(sql, (courier['full_name'], courier['route_id'], courier['passport'],
courier['phone'], courier['courier_id']))
                sql_count_ref[0] += 1
                if first_byte_time[0] is None:
                    first_byte_time[0] = time.perf_counter()
                db.conn.commit()


            def orm_update(courier, sql_count_ref, first_byte_time):
                session_db = SessionLocal()
                try:
                                                                                row =
session_db.query(Courier).filter_by(courier_id=courier['courier_id']).first()
                    if row:
                        row.full_name = courier['full_name']
                        row.route_id = courier['route_id']
                        row.passport = courier['passport']
                        row.phone = courier['phone']
                        session_db.commit()
                        sql_count_ref[0] += 1
                        if first_byte_time[0] is None:
                            first_byte_time[0] = time.perf_counter()
                    except Exception as e:
                        print("[ORM ERROR]", e)
                        session_db.rollback()
                    finally:
                        session_db.close()


            methods = [("RAW SQL", raw_update), ("PARAM SQL", param_update), ("ORM",
orm_update)]
            results = {}
            for name, func in methods:
                results[name] = measure_performance(func, courier_data)

            # ======================== Вивід таблиці ========================
              print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} |
{'CPU%':<6} | {'MEM(bytes)':<10} | {'SQL#':<5}")
            print("-"*95)
            for method, data in results.items():
                print(f"{method:<10} | {data['avg_time']:<12.6f} | {data['avg_tps']:<8.2f} | "
                                f"{data['avg_latency']:<12.6f}  |  {data['avg_cpu']:<6.2f}  |
{data['avg_mem']:<10} | {data['avg_sql_count']:<5}")

            db.close()
            return redirect(url_for("couriers"))

        # ======================== GET ========================
        try:
```

```
            with db.conn.cursor(dictionary=True) as cursor:
                cursor.execute("SELECT route_id, route_name FROM routes")
                routes = cursor.fetchall()

               cursor.execute("SELECT * FROM couriers WHERE courier_id=%s", (courier_id,))
                courier = cursor.fetchone()
        finally:
            db.close()

            return render_template("courier_form.html", action="edit", courier=courier,
routes=routes, role="admin")
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції DELETE (видалення кур'єра)

```
@app.route("/couriers/delete/<int:courier_id>")
def delete_courier(courier_id):
    if session.get("role") != "admin":
        return redirect(url_for("couriers"))

    db = DBConnection()
    db.connect()

    START_ID = 18633
    NUM_PER_METHOD = 50
    process = psutil.Process(os.getpid())


    def measure_performance(func, ids):
        elapsed_times = []
        tps_list = []
        latencies = []
        cpu_used_list = []
        mem_used_list = []

        for cid in ids:
            mem_before = process.memory_info().rss
            cpu_before = process.cpu_percent(interval=0.01)

            start = time.perf_counter()
            first_byte_time = [None]

            func(cid, first_byte_time)

            end = time.perf_counter()
            mem_after = process.memory_info().rss
            cpu_after = process.cpu_percent(interval=0.01)

            elapsed = end - start
```

```
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        tps_list.append(tps)
        latencies.append(latency)
        cpu_used_list.append(cpu_used)
        mem_used_list.append(mem_used)

    avg_elapsed = sum(elapsed_times) / len(ids)
    avg_tps = sum(tps_list) / len(ids)
    avg_latency = sum(latencies) / len(ids)
    avg_cpu = sum(cpu_used_list) / len(ids)
    avg_mem = sum(mem_used_list) / len(ids)

    return avg_elapsed, avg_tps, avg_latency, avg_cpu, avg_mem


def raw_delete(courier_id, first_byte_time):
    with db.conn.cursor() as cursor:
        cursor.execute(f"DELETE FROM couriers WHERE courier_id={courier_id}")
        db.conn.commit()
        if first_byte_time[0] is None:
            first_byte_time[0] = time.perf_counter()


def param_delete(courier_id, first_byte_time):
    with db.conn.cursor() as cursor:
      cursor.execute("DELETE FROM couriers WHERE courier_id=%s", (courier_id,))
        db.conn.commit()
        if first_byte_time[0] is None:
            first_byte_time[0] = time.perf_counter()


def orm_delete(courier_id, first_byte_time):
    session_db = SessionLocal()
    try:
      courier = session_db.query(Courier).filter(Courier.courier_id == courier_id).first()
        if courier:
            session_db.delete(courier)
            session_db.commit()
        if first_byte_time[0] is None:
            first_byte_time[0] = time.perf_counter()
    except Exception as e:
        print("[ORM ERROR]", e)
        session_db.rollback()
    finally:
        session_db.close()

raw_ids = list(range(START_ID, START_ID + NUM_PER_METHOD))
```

```
            param_ids = list(range(START_ID + NUM_PER_METHOD, START_ID + NUM_PER_METHOD*2))
            orm_ids = list(range(START_ID + NUM_PER_METHOD*2, START_ID + NUM_PER_METHOD*3))

        methods = [
            ("RAW SQL", raw_delete, raw_ids),
            ("PARAM SQL", param_delete, param_ids),
            ("ORM", orm_delete, orm_ids)
        ]

        measurement_results = []

        for name, func, ids in methods:
            avg_time, avg_tps, avg_latency, avg_cpu, avg_mem = measure_performance(func, ids)
            measurement_results.append({
                "method": name,
                "avg_time": avg_time,
                "avg_tps": avg_tps,
                "avg_latency": avg_latency,
                "cpu": avg_cpu,
                "mem": avg_mem
            })


          print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} | {'CPU%':<6} | {'MEM(bytes)':<10}")
        print("-"*80)
        for r in measurement_results:
                print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} | {r['avg_latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

        db.close()
        return redirect(url_for("couriers"))
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції SELECT із JOIN (отримання кур’єрів із маршрутами)

```
@app.route("/sql-test/join/<int:min_courier_id>")
def sql_test_join(min_courier_id):
    db = DBConnection()
    db.connect()
    NUM_RUNS = 1000
    process = psutil.Process(os.getpid())

    def measure(func, name):
        elapsed_times, tps_list, latencies, cpu_list, mem_list = [], [], [], [], []
```

```
    result_data = None

    for _ in range(NUM_RUNS):
        mem_before = process.memory_info().rss
        cpu_before = process.cpu_percent(interval=0.01)
        start = time.perf_counter()
        first_byte = [None]

        data = func(min_courier_id, first_byte)

        end = time.perf_counter()
        mem_after = process.memory_info().rss
        cpu_after = process.cpu_percent(interval=0.01)

        elapsed = end - start
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte[0] - start if first_byte[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        tps_list.append(tps)
        latencies.append(latency)
        cpu_list.append(cpu_used)
        mem_list.append(mem_used)

        if result_data is None:
            result_data = data

    return result_data, {
        "method": name,
        "avg_time": sum(elapsed_times)/NUM_RUNS,
        "avg_tps": sum(tps_list)/NUM_RUNS,
        "latency": sum(latencies)/NUM_RUNS,
        "cpu": sum(cpu_list)/NUM_RUNS,
        "mem": sum(mem_list)/NUM_RUNS
    }


def raw(min_id, fb):
    with db.conn.cursor(dictionary=True) as c:
        sql = f"""
            SELECT c.full_name, r.route_name
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
            WHERE c.courier_id > {min_id}
        """
        c.execute(sql)
        fb[0] = time.perf_counter()
        return c.fetchall()

def param(min_id, fb):
```

```
            with db.conn.cursor(dictionary=True) as c:
                sql = """
                    SELECT c.full_name, r.route_name
                    FROM couriers c
                    JOIN routes r ON c.route_id = r.route_id
                    WHERE c.courier_id > %s
                """
                c.execute(sql, (min_id,))
                fb[0] = time.perf_counter()
                return c.fetchall()


        class CourierORM:
            def __init__(self, row):
                self.name = row["full_name"]
                self.route = row["route_name"]

        def orm(min_id, fb):
            with db.conn.cursor(dictionary=True) as c:
                sql = """
                    SELECT c.full_name, r.route_name
                    FROM couriers c
                    JOIN routes r ON c.route_id = r.route_id
                    WHERE c.courier_id > %s
                """
                c.execute(sql, (min_id,))
                rows = c.fetchall()
                fb[0] = time.perf_counter()
                return [CourierORM(r) for r in rows]

        results = []
        raw_data, raw_res = measure(raw, "RAW")
        results.append(raw_res)
        param_data, param_res = measure(param, "PARAM")
        results.append(param_res)
        orm_data, orm_res = measure(orm, "ORM")
        results.append(orm_res)

        # ======================== Вивід у консоль ========================
          print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} |
{'CPU%':<6} | {'MEM(bytes)':<10}")
        print("-"*75)
        for r in results:
                print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} |
{r['latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

        db.close()
        return render_template("sql_test.html", title="JOIN", results=results)
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції SELECT із агрегацією (підрахунок кур'єрів та середньої довжини маршруту)

```
@app.route("/sql-test/aggregation/<int:min_courier_id>")
def sql_test_aggregation(min_courier_id):
    db = DBConnection()
    db.connect()
    NUM_RUNS = 1000
    process = psutil.Process(os.getpid())

    def measure(func, name):
        elapsed_times, tps_list, latencies, cpu_list, mem_list = [], [], [], [], []
        result_data = None

        for _ in range(NUM_RUNS):
            mem_before = process.memory_info().rss
            cpu_before = process.cpu_percent(interval=0.01)
            start = time.perf_counter()
            first_byte = [None]

            data = func(min_courier_id, first_byte)

            end = time.perf_counter()
            mem_after = process.memory_info().rss
            cpu_after = process.cpu_percent(interval=0.01)

            elapsed = end - start
            tps = 1 / elapsed if elapsed > 0 else float('inf')
            latency = first_byte[0] - start if first_byte[0] else elapsed
            cpu_used = max(cpu_after - cpu_before, 0)
            mem_used = max(mem_after - mem_before, 0)

            elapsed_times.append(elapsed)
            tps_list.append(tps)
            latencies.append(latency)
            cpu_list.append(cpu_used)
            mem_list.append(mem_used)

            if result_data is None:
                result_data = data

        return result_data, {
            "method": name,
            "avg_time": sum(elapsed_times)/NUM_RUNS,
            "avg_tps": sum(tps_list)/NUM_RUNS,
            "latency": sum(latencies)/NUM_RUNS,
            "cpu": sum(cpu_list)/NUM_RUNS,
            "mem": sum(mem_list)/NUM_RUNS
```

```
    }

def raw(min_id, fb):
    with db.conn.cursor(dictionary=True) as c:
        sql = f"""
            SELECT r.route_name, COUNT(c.courier_id) AS courier_count,
                AVG(r.length_km) AS avg_length
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
            WHERE c.courier_id > {min_id}
            GROUP BY r.route_name
        """
        c.execute(sql)
        fb[0] = time.perf_counter()
        return c.fetchall()
def param(min_id, fb):
    with db.conn.cursor(dictionary=True) as c:
        sql = """
            SELECT r.route_name, COUNT(c.courier_id) AS courier_count,
                AVG(r.length_km) AS avg_length
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
            WHERE c.courier_id > %s
            GROUP BY r.route_name
        """
        c.execute(sql, (min_id,))
        fb[0] = time.perf_counter()
        return c.fetchall()

class AggregationORM:
    def __init__(self, row):
        self.route_name = row["route_name"]
        self.courier_count = row["courier_count"]
        self.avg_length = float(row["avg_length"])

def orm(min_id, fb):
    with db.conn.cursor(dictionary=True) as c:
        sql = """
            SELECT r.route_name, COUNT(c.courier_id) AS courier_count,
                AVG(r.length_km) AS avg_length
            FROM couriers c
            JOIN routes r ON c.route_id = r.route_id
            WHERE c.courier_id > %s
            GROUP BY r.route_name
        """
        c.execute(sql, (min_id,))
        rows = c.fetchall()
        fb[0] = time.perf_counter()
        return [AggregationORM(r) for r in rows]

results = []
raw_data, raw_res = measure(raw, "RAW")
```

```
        results.append(raw_res)
        param_data, param_res = measure(param, "PARAM")
        results.append(param_res)
        orm_data, orm_res = measure(orm, "ORM")
        results.append(orm_res)

        print(f"{'Method':<10} | {'Avg Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} | {'CPU%':<6} | {'MEM(bytes)':<10}")
        print("-"*75)
        for r in results:
            print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} | {r['latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

        db.close()
        return render_template("sql_test.html", title="AGGREGATION", results=results)
```

Модуль експериментального порівняльного аналізу продуктивності методів доступу до БД при операції SELECT із агрегацією (підрахунок кур'єрів та середньої довжини маршруту)

```
@app.route("/sql-test/aggregation-nplus1/<int:min_courier_id>")
def sql_test_aggregation_nplus1(min_courier_id):
    db = DBConnection()
    db.connect()
    NUM_RUNS = 100
    process = psutil.Process(os.getpid())

    def measure(func, name):
        elapsed_times, tps_list, latencies, cpu_list, mem_list = [], [], [], [], []
        result_data = None

        for _ in range(NUM_RUNS):
            mem_before = process.memory_info().rss
            cpu_before = process.cpu_percent(interval=0.01)
            start = time.perf_counter()
            first_byte = [None]

            data = func(min_courier_id, first_byte)

            end = time.perf_counter()
            mem_after = process.memory_info().rss
            cpu_after = process.cpu_percent(interval=0.01)

            elapsed = end - start
            tps = 1 / elapsed if elapsed > 0 else float('inf')
            latency = first_byte[0] - start if first_byte[0] else elapsed
            cpu_used = max(cpu_after - cpu_before, 0)
            mem_used = max(mem_after - mem_before, 0)
```

```
                elapsed_times.append(elapsed)
                tps_list.append(tps)
                latencies.append(latency)
                cpu_list.append(cpu_used)
                mem_list.append(mem_used)

                if result_data is None:
                    result_data = data

            stats = {
                "method": name,
                "avg_time": sum(elapsed_times)/NUM_RUNS,
                "avg_tps": sum(tps_list)/NUM_RUNS,
                "latency": sum(latencies)/NUM_RUNS,
                "cpu": sum(cpu_list)/NUM_RUNS,
                "mem": sum(mem_list)/NUM_RUNS
            }
            return result_data, stats


        def raw_method(min_id, fb):
            with db.conn.cursor(dictionary=True) as c:
                c.execute("SELECT route_id, route_name, length_km FROM routes")
                routes = c.fetchall()
                results = []
                for r in routes:
                    sql = "SELECT courier_id, full_name FROM couriers WHERE courier_id > %s
AND route_id = %s"
                    c.execute(sql, (min_id, r["route_id"]))
                    couriers = c.fetchall()
                    results.append({
                        "route_name": r["route_name"],
                        "courier_count": len(couriers),
                        "avg_length": r["length_km"]
                    })
                fb[0] = time.perf_counter()
                return results

        def param_method(min_id, fb):
            with db.conn.cursor(dictionary=True) as c:
                c.execute("SELECT route_id, route_name, length_km FROM routes")
                routes = c.fetchall()
                results = []
                for r in routes:
                    sql = "SELECT courier_id, full_name FROM couriers WHERE courier_id > %s
AND route_id = %s"
                    c.execute(sql, (min_id, r["route_id"]))
                    couriers = c.fetchall()
                    results.append({
                        "route_name": r["route_name"],
                        "courier_count": len(couriers),
                        "avg_length": r["length_km"]
```

```
                })
            fb[0] = time.perf_counter()
            return results

        class AggregationORM:
            def __init__(self, route_name, courier_count, avg_length):
                self.route_name = route_name
                self.courier_count = courier_count
                self.avg_length = avg_length

        def orm_method(min_id, fb):
            session = SessionLocal()
            try:
                routes = session.query(Route).all()
                results = []
                for r in routes:
                    couriers = session.query(Courier).filter(
                        Courier.courier_id > min_id,
                        Courier.route_id == r.route_id
                    ).all()
                                    results.append(AggregationORM(r.route_name,    len(couriers),
float(r.length_km)))
                fb[0] = time.perf_counter()
                return results
            finally:
                session.close()

        funcs = {"RAW": raw_method, "PARAM": param_method, "ORM": orm_method}
        results_table = []

        for name, func in funcs.items():
            _, stats = measure(func, name)
            results_table.append(stats)

        db.close()

          print(f"{'Method':<10} | {'Avg  Time(s)':<12} | {'TPS':<8} | {'Latency(s)':<12} |
{'CPU%':<6} | {'MEM(bytes)':<10}")
        print("-"*75)
        for r in results_table:
                print(f"{r['method']:<10} | {r['avg_time']:<12.6f} | {r['avg_tps']:<8.2f} |
{r['latency']:<12.6f} | {r['cpu']:<6.2f} | {r['mem']:<10}")

            return      render_template("sql_test.html",      title="AGGREGATION      N+1",
results=results_table)
```

ДОДАТОК Е

Модуль експериментального навантажувального тестування web-додатку для операції автентифікації користувачів (SELECT) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between
USERS = [
    {"login": f"admin{i}", "password": "admin123"}
    for i in range(1, 101)
]

class LoginRawUser(HttpUser):
    wait_time = between(0.5, 1.5)
    user_index = 0

    @task
    def login_raw(self):
        if LoginRawUser.user_index >= 30:
            LoginRawUser.user_index = 0
        user = USERS[LoginRawUser.user_index]
        LoginRawUser.user_index += 1

        response = self.client.post(
            "/",
            data={
                "login": user["login"],
                "password": user["password"],
                "method": "raw_login",
                "test_mode": "1"
            },
            name="Login RAW",
            allow_redirects=True
        )

        if response.status_code not in (200, 302):
            print(f"[RAW ERROR] {response.status_code} {response.text}")

class LoginParamUser(HttpUser):
    wait_time = between(0.5, 1.5)
    user_index = 30

    @task
    def login_param(self):
        if LoginParamUser.user_index >= 60:
            LoginParamUser.user_index = 30
        user = USERS[LoginParamUser.user_index]
        LoginParamUser.user_index += 1
```

```
        response = self.client.post(
            "/",
            data={
                "login": user["login"],
                "password": user["password"],
                "method": "param_login",
                "test_mode": "1"
            },
            name="Login PARAM",
            allow_redirects=True
        )

        if response.status_code not in (200, 302):
            print(f"[PARAM ERROR] {response.status_code} {response.text}")

class LoginOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)
    user_index = 60

    @task
    def login_orm(self):
        if LoginOrmUser.user_index >= 90:
            LoginOrmUser.user_index = 60
        user = USERS[LoginOrmUser.user_index]
        LoginOrmUser.user_index += 1

        response = self.client.post(
            "/",
            data={
                "login": user["login"],
                "password": user["password"],
                "method": "orm_login",
                "test_mode": "1"
            },
            name="Login ORM",
            allow_redirects=True
        )

        if response.status_code not in (200, 302):
            print(f"[ORM ERROR] {response.status_code} {response.text}")
```

Модуль експериментального навантажувального тестування web-додатку для операції отримання списку кур’єрів (SELECT) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between

class GetCouriersRawUser(HttpUser):
```

```
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_raw(self):
        params = {"method": "raw", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers RAW")

class GetCouriersParamUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_param(self):
        params = {"method": "param", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers PARAM")

class GetCouriersOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_orm(self):
        params = {"method": "orm", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers ORM")
```

Модуль експериментального навантажувального тестування web-додатку для операції додавання кур'єра (INSERT) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between
import random

def random_courier_data():
    return {
        "full_name": f"Courier {random.randint(1,1000)}",
        "route_id": random.randint(36,40),
        "passport": f"AB{random.randint(100000,999999)}",
        "phone": f"+3809{random.randint(10000000,99999999)}"
    }

class AddCourierRawUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_raw(self):
        data = random_courier_data()
        data["method"] = "raw"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier RAW")

class AddCourierParamUser(HttpUser):
```

```
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_param(self):
        data = random_courier_data()
        data["method"] = "param"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier PARAM")

class AddCourierOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_orm(self):
        data = random_courier_data()
        data["method"] = "orm"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier ORM")
```

Модуль експериментального навантажувального тестування web-додатку для операції оновлення кур'єра (UPDATE) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between
import random

def random_courier_update_data(courier_id):
    return {
        "courier_id": courier_id,  # конкретний кур'єр
        "full_name": f"Courier Updated {random.randint(1,1000)}",
        "route_id": random.randint(36,40),
        "passport": f"AB{random.randint(100000,999999)}",
        "phone": f"+3809{random.randint(10000000,99999999)}"
    }

class UpdateCourierRawUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def update_courier_raw(self):
        courier_id = random.randint(18430, 18480)
        data = random_courier_update_data(courier_id)
        data["method"] = "raw"
        data["test_mode"] = "1"
            self.client.post(f"/couriers/edit/{courier_id}", data=data, name="Update Courier
RAW")

class UpdateCourierParamUser(HttpUser):
```

```
    wait_time = between(0.5, 1.5)

    @task
    def update_courier_param(self):
        courier_id = random.randint(18430, 18480)
        data = random_courier_update_data(courier_id)
        data["method"] = "param"
        data["test_mode"] = "1"
            self.client.post(f"/couriers/edit/{courier_id}", data=data, name="Update Courier
PARAM")

class UpdateCourierOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def update_courier_orm(self):
        courier_id = random.randint(18430, 18480)
        data = random_courier_update_data(courier_id)
        data["method"] = "orm"
        data["test_mode"] = "1"
            self.client.post(f"/couriers/edit/{courier_id}", data=data, name="Update Courier
ORM")
```

Модуль експериментального навантажувального тестування web-додатку для операції видалення кур’єра (DELETE) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between

class DeleteCourierRawUser(HttpUser):
    wait_time = between(0.5, 1.5)
    start_id = 18800
    end_id = 18900
    current_id = start_id

    @task
    def delete_courier_raw(self):
        if DeleteCourierRawUser.current_id > DeleteCourierRawUser.end_id:
            return
        courier_id = DeleteCourierRawUser.current_id
        DeleteCourierRawUser.current_id += 1
        self.client.post(
            f"/couriers/delete/{courier_id}",
            data={"test_mode": "1"},
            name="Delete Courier RAW"
        )

class DeleteCourierParamUser(HttpUser):
    wait_time = between(0.5, 1.5)
```

```
    start_id = 18901
    end_id = 19000
    current_id = start_id

    @task
    def delete_courier_param(self):
        if DeleteCourierParamUser.current_id > DeleteCourierParamUser.end_id:
            return
        courier_id = DeleteCourierParamUser.current_id
        DeleteCourierParamUser.current_id += 1
        self.client.post(
            f"/couriers/delete/{courier_id}",
            data={"test_mode": "1"},
            name="Delete Courier PARAM"
        )

class DeleteCourierOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)
    start_id = 19001
    end_id = 19100
    current_id = start_id

    @task
    def delete_courier_orm(self):
        if DeleteCourierOrmUser.current_id > DeleteCourierOrmUser.end_id:
            return
        courier_id = DeleteCourierOrmUser.current_id
        DeleteCourierOrmUser.current_id += 1
        self.client.post(
            f"/couriers/delete/{courier_id}",
            data={"test_mode": "1"},
            name="Delete Courier ORM"
        )
```

Модуль експериментального навантажувального тестування web-додатку для операції SQL JOIN (SELECT) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between
import random

def random_courier_data():
    return {
        "full_name": f"Courier {random.randint(1,1000)}",
        "route_id": random.randint(36,40),
        "passport": f"AB{random.randint(100000,999999)}",
        "phone": f"+3809{random.randint(10000000,99999999)}"
    }
```

```
class AddCourierRawUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_raw(self):
        data = random_courier_data()
        data["method"] = "raw"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier RAW")

class AddCourierParamUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_param(self):
        data = random_courier_data()
        data["method"] = "param"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier PARAM")

class AddCourierOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def add_courier_orm(self):
        data = random_courier_data()
        data["method"] = "orm"
        data["test_mode"] = "1"
        self.client.post("/couriers/add", data=data, name="Add Courier ORM")
```

Модуль експериментального навантажувального тестування web-додатку для операції SQL AGGREGATION (GROUP BY, COUNT, AVG) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM

```
import random
from locust import HttpUser, task, between

MIN_ID = 18833
MAX_ID = 19500

class AggregationRawUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def test_raw(self):
        min_id = random.randint(MIN_ID, MAX_ID)
                                self.client.get(f"/sql-test/aggregation/{min_id}?method=raw",
name="AGGREGATION RAW")
```

```
class AggregationParamUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def test_param(self):
        min_id = random.randint(MIN_ID, MAX_ID)
                                self.client.get(f"/sql-test/aggregation/{min_id}?method=param",
name="AGGREGATION PARAM")

class AggregationOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def test_orm(self):
        min_id = random.randint(MIN_ID, MAX_ID)
                                  self.client.get(f"/sql-test/aggregation/{min_id}?method=orm",
name="AGGREGATION ORM")
```

Модуль експериментального навантажувального тестування web-додатку для операції отримання агрегованих даних кур'єрів (SELECT, N+1) із порівнянням методів доступу до БД (RAW SQL, PARAM SQL, ORM)

```
from locust import HttpUser, task, between
import random

MIN_COURIER_ID_START = 18833
MIN_COURIER_ID_END = 19500

def random_courier_id():
    return random.randint(MIN_COURIER_ID_START, MIN_COURIER_ID_END)

class NPlus1RawUser(HttpUser):
    wait_time = between(1, 1.5)

    @task
    def test_raw(self):
        courier_id = random_courier_id()
        self.client.get(
            f"/sql-test/aggregation-nplus1/{courier_id}?method=raw",
            name="AGGREGATION N+1 RAW"
        )

class NPlus1ParamUser(HttpUser):
    wait_time = between(1, 1.5)

    @task
    def test_param(self):
        courier_id = random_courier_id()
```

```
        self.client.get(
            f"/sql-test/aggregation-nplus1/{courier_id}?method=param",
            name="AGGREGATION N+1 PARAM"
        )

class NPlus1OrmUser(HttpUser):
    wait_time = between(1, 1.5)

    @task
    def test_orm(self):
        courier_id = random_courier_id()
        self.client.get(
            f"/sql-test/aggregation-nplus1/{courier_id}?method=orm",
            name="AGGREGATION N+1 ORM"
        )
```

ДОДАТОК Є

Презентація

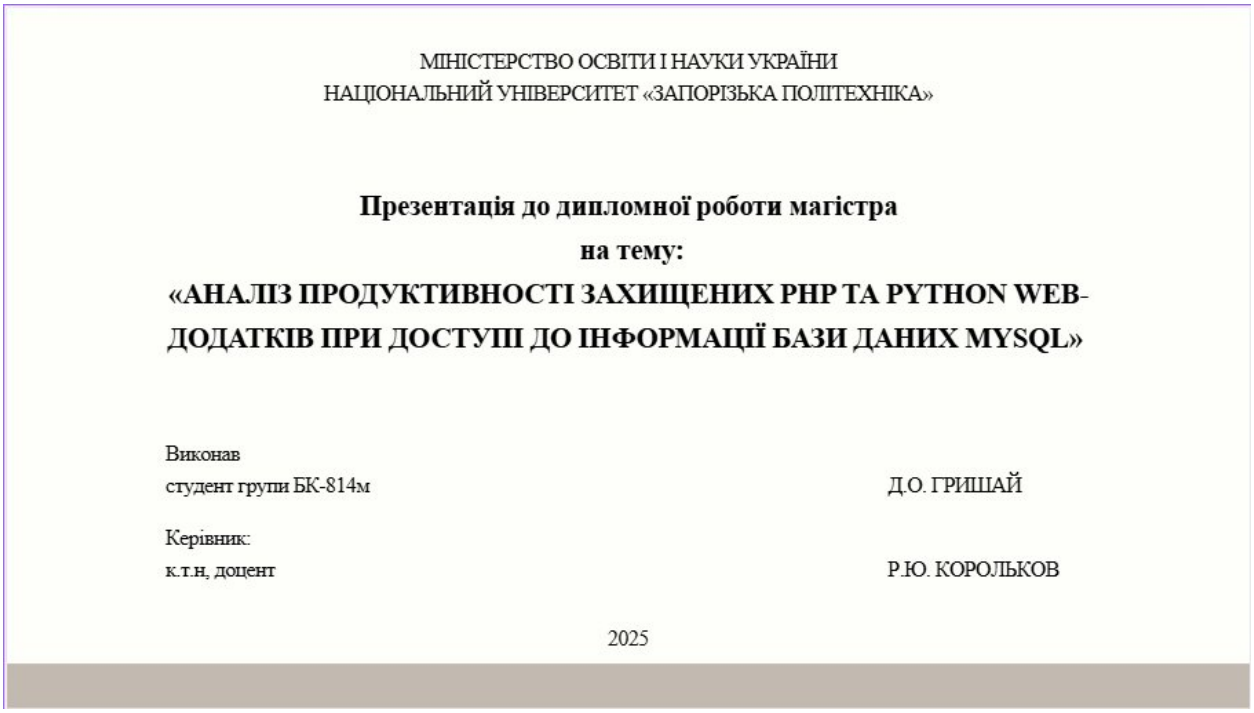


Рисунок Є.1 – Титульний слайд

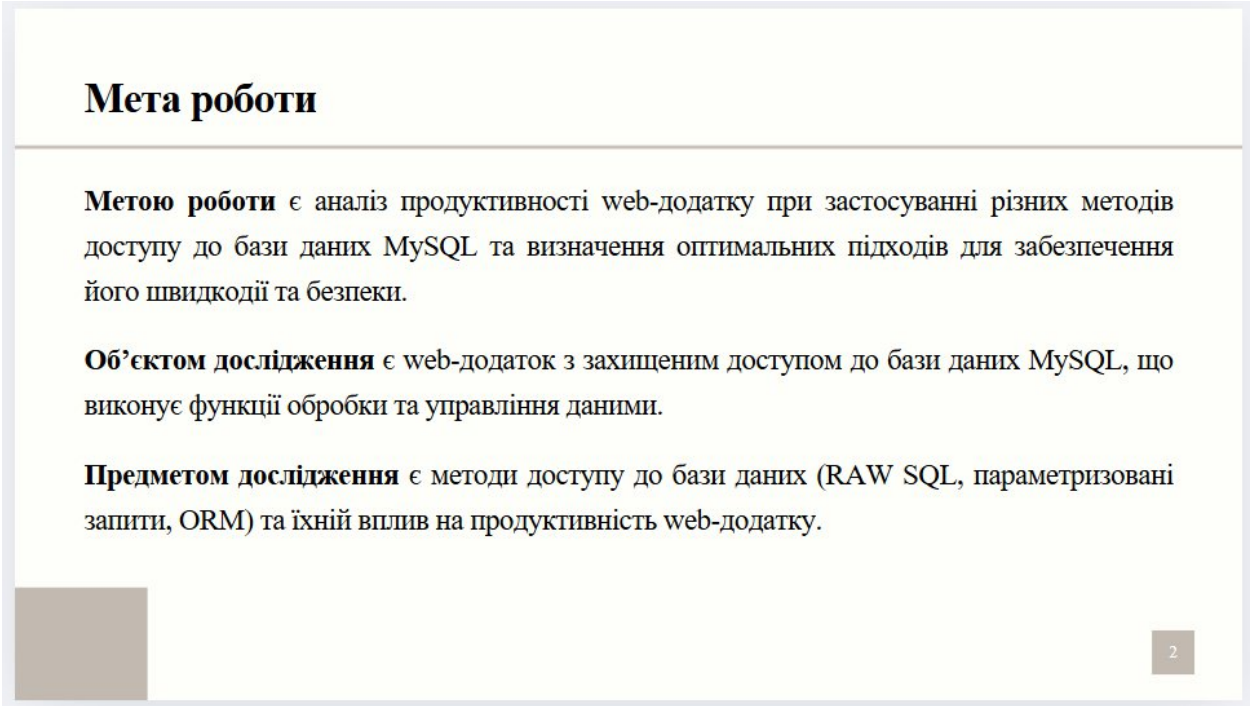


Рисунок Є.2 – Мета роботи

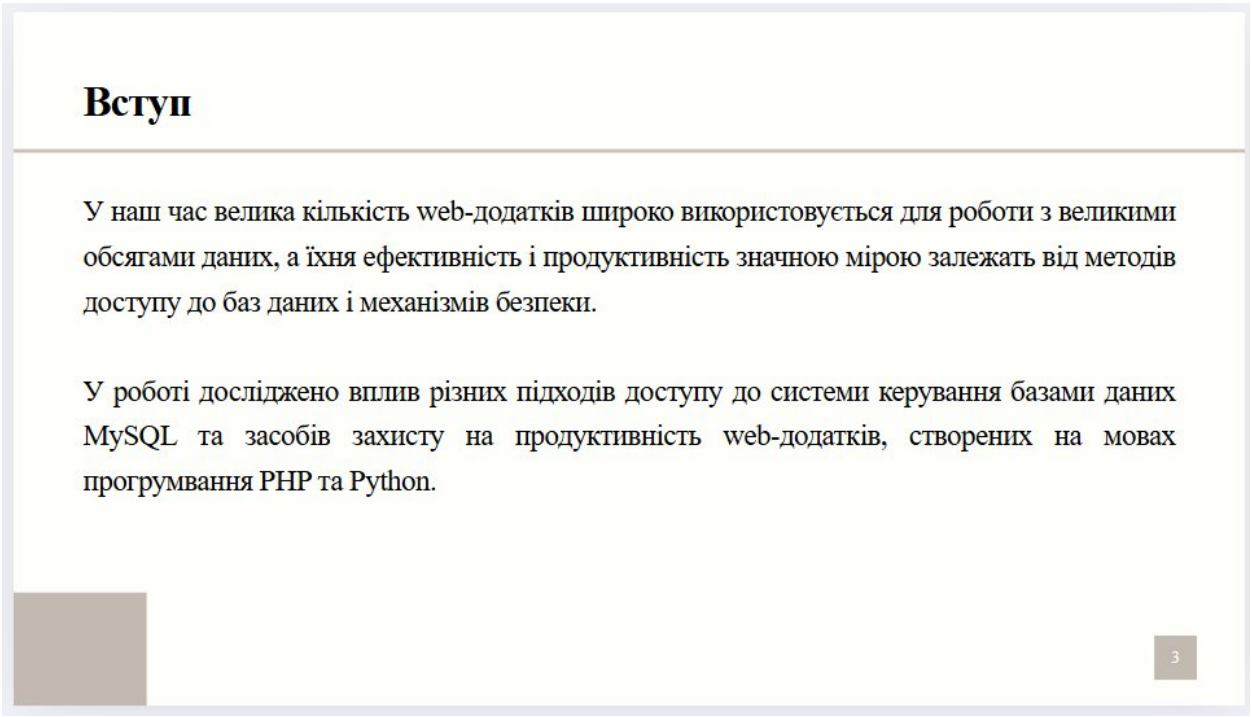


Рисунок Є.3 – Вступ

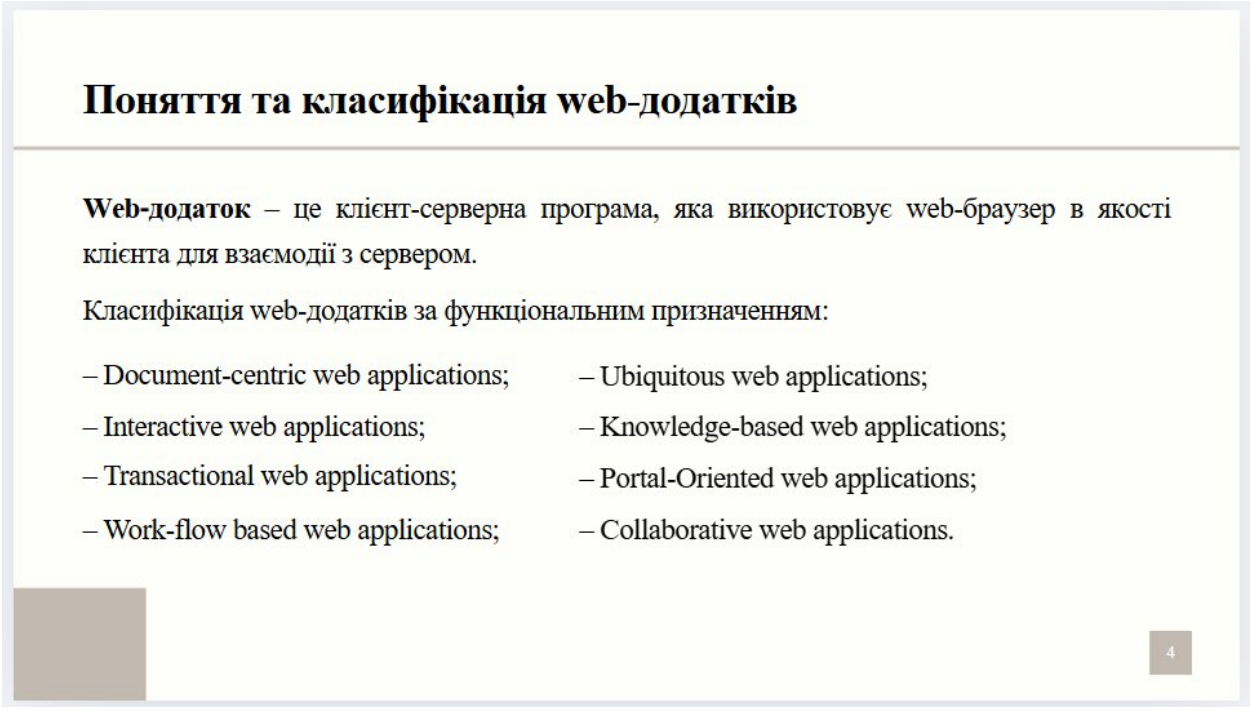


Рисунок Є.4 – Поняття та класифікація web-додатків

Архітектура web-додатків

Архітектура web-додатку визначає структуру системи, взаємодію її компонентів і роботу з базою даних, безпосередньо впливаючи на продуктивність, безпеку та можливість масштабування додатку.

Найпоширенішими архітектурними моделями є:

– монолітна;
– мікросервісна;
– сервісно-орієнтована;
– подієво-орієнтована.

У роботі буде розглянуто класичний **клієнт-серверний підхід**, бо він є фундаментальною основою більшості сучасних web-додатків.

5

Рисунок Є.5 – Архітектура web-додатків

База даних MySQL та методи доступу до неї у web-додатках

MySQL – це відкрита система керування реляційними базами даних, яка використовується для зберігання, обробки та доступу до структурованих даних.

Методи доступу до MySQL у PHP-додатках

розширення MySQLi | PHP Data Objects | ORM

Методи доступу до MySQL у Python-додатках

драйвери (mysql.connector..) | плагіни (Flask-SQLAlchemy..) | ORM

6

Рисунок Є.6 – База даних MySQL та методи доступу до неї у web-додатках

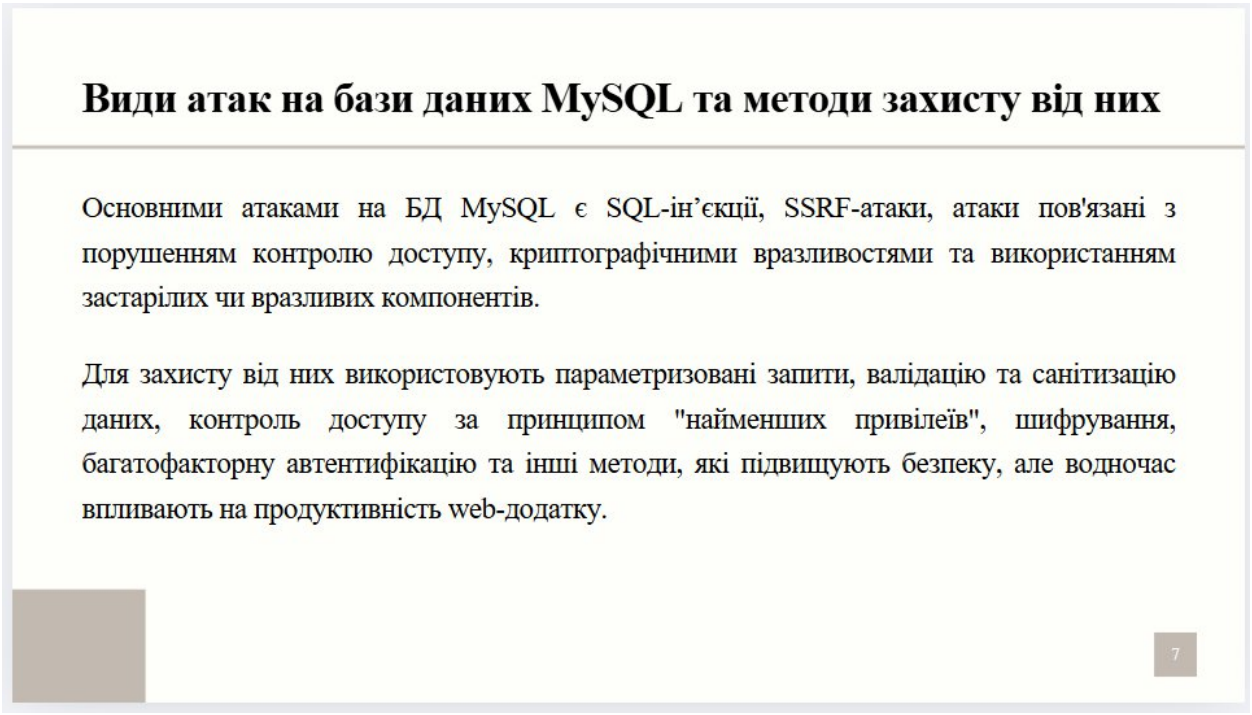


Рисунок Є.7 – Види атак на бази даних MySQL та методи захисту від них

Рисунок Є.8 – Огляд інструментів аналізу продуктивності

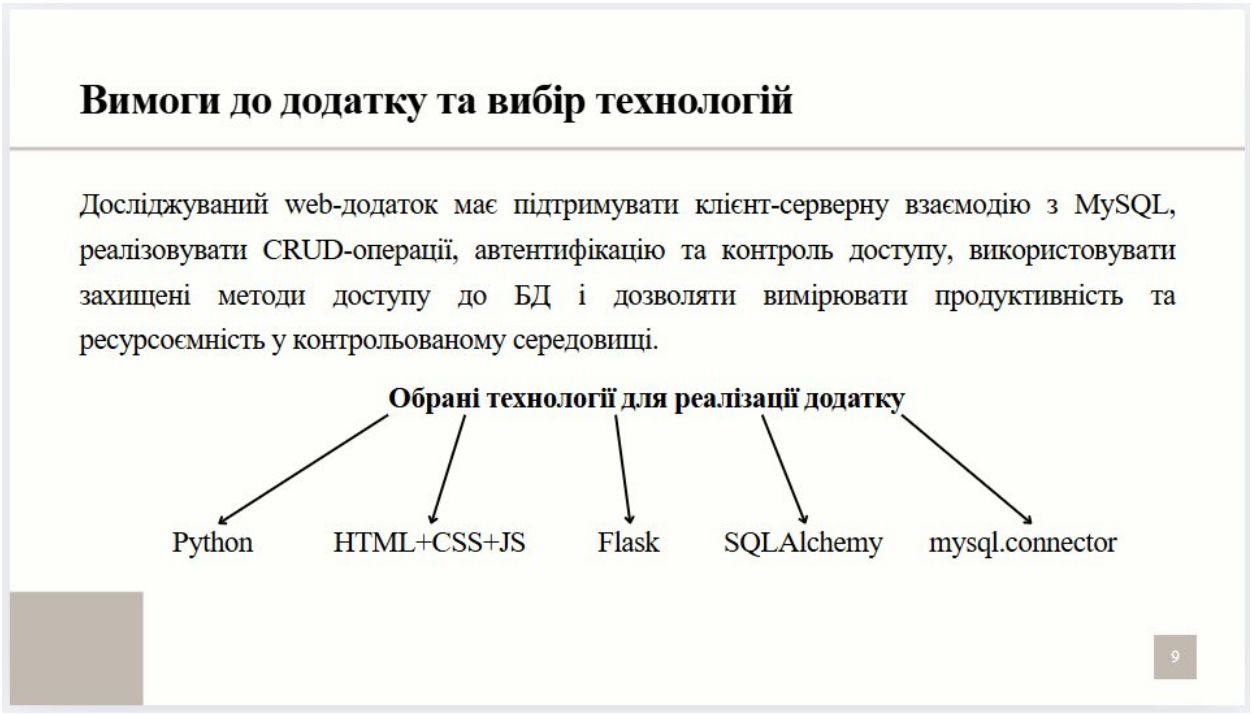


Рисунок Є.9 – Вимоги до додатку та вибір технологій

Рисунок Є.10 – Проєктування структури бази даних MySQL

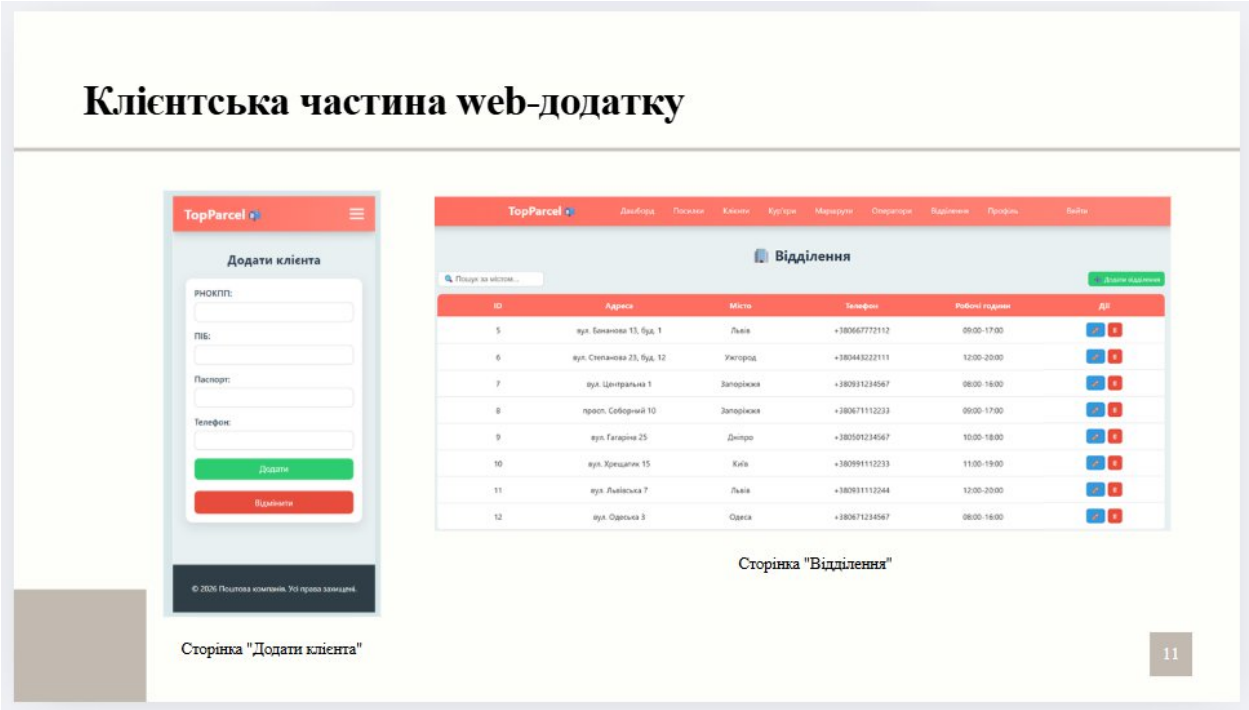


Рисунок Є.11 – Клієнтська частина web-додатку

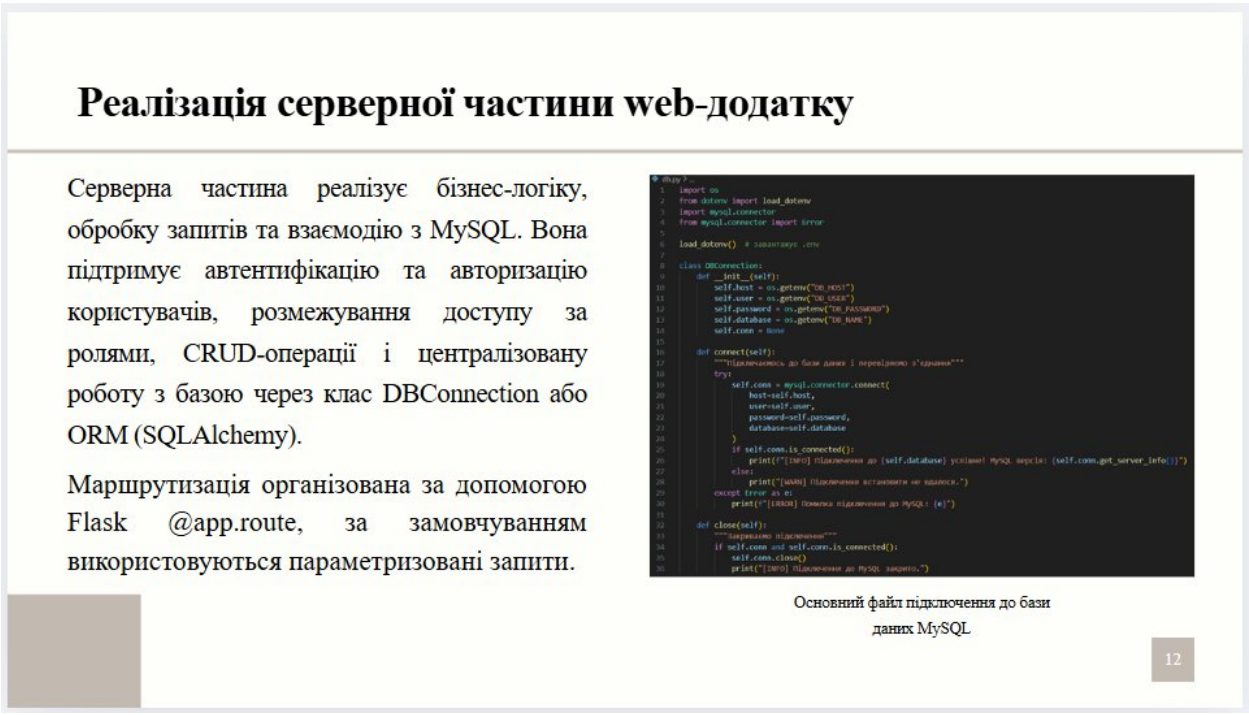


Рисунок Є.12 – Реалізація серверної частини web-додатку

Методика проведення аналізу продуктивності

Для аналізу використовувався розроблений Python-додаток з MySQL, де доступ до БД здійснювався трьома способами:

– сирими SQL-запитами;

– параметризованими SQL-запитами;

– із застосуванням ORM-підходу.

Методика тестування полягала у виконанні типових запитів до БД (CRUD, JOIN, AGGREGATION, N+1) у ході двох експериментів, а саме:

– багаторазовому виконанні запитів (1000 для кожного методу);

– навантажувальному тестуванні (за допомогою Locust).

13

Рисунок Є.13 – Методика проведення аналізу продуктивності

Вимірювані показники продуктивності

При багаторазовому послідовному виконанні запитів для кожного запиту та способу доступу до даних (RAW, PARAM, ORM) вимірюються наступні показники:

– середній час виконання запиту (Avg Time, c):

$$Avg\ Time = \frac{1}{N}\sum_{i}^{N} t_i$$

– середня затримка відповіді (Latency, c) :

$$Latency = \frac{1}{N}\sum_{i=1}^{N}(t_{first\ byte,i} - t_{start,i})$$

– пропускна здатність (Transactions per second, TPS) :

$$TPS = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{t_i}$$

– середні використання процесора (CPU%) та ОЗП (MEM, bytes):

$$CPU\% = \frac{1}{N}\sum_{i=1}^{N}(CPU_{before,i} - CPU_{after,i}) \qquad MEM = \frac{1}{N}\sum_{i=1}^{N}(RSS_{after,i} - RSS_{before,i})$$

Основними показниками продуктивності під час навантажувального тестування є:

– кількість виконаних запитів (# Requests);

– кількість помилок (# Fails);

– середній час відповіді (Average, ms);

– мінімальний та максимальний час відповіді (Min, Max, ms);

– середній розмір відповіді (Average size, bytes) і пропускна здатність (RPS);

– кількість помилок за секунду (Failures/s) і процентильні показники.

14

Рисунок Є.14 – Вимірювані показники продуктивності

Проведення експериментів

```python
def measure_performance(func, ids):
    elapsed_times = []
    tps_list = []
    latencies = []
    cpu_used_list = []
    mem_used_list = []

    for cid in ids:
        mem_before = process.memory_info().rss
        cpu_before = process.cpu_percent(interval=0.01)

        start = time.perf_counter()
        first_byte_time = [None]

        func(cid, first_byte_time)

        end = time.perf_counter()
        mem_after = process.memory_info().rss
        cpu_after = process.cpu_percent(interval=0.01)

        elapsed = end - start
        tps = 1 / elapsed if elapsed > 0 else float('inf')
        latency = first_byte_time[0] - start if first_byte_time[0] else elapsed
        cpu_used = max(cpu_after - cpu_before, 0)
        mem_used = max(mem_after - mem_before, 0)

        elapsed_times.append(elapsed)
        tps_list.append(tps)
        latencies.append(latency)
        cpu_used_list.append(cpu_used)
        mem_used_list.append(mem_used)

    avg_elapsed = sum(elapsed_times) / len(ids)
    avg_tps = sum(tps_list) / len(ids)
    avg_latency = sum(latencies) / len(ids)
    avg_cpu = sum(cpu_used_list) / len(ids)
    avg_mem = sum(mem_used_list) / len(ids)

    return avg_elapsed, avg_tps, avg_latency, avg_cpu, avg_mem
```

Функція для замірів при проведенні багаторазових послідовних запитів

```python
from locust import HttpUser, task, between

class GetCouriersRawUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_raw(self):
        params = {"method": "raw", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers RAW")

class GetCouriersParamUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_param(self):
        params = {"method": "param", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers PARAM")

class GetCouriersOrmUser(HttpUser):
    wait_time = between(0.5, 1.5)

    @task
    def get_couriers_orm(self):
        params = {"method": "orm", "test_mode": "1"}
        self.client.get("/couriers", params=params, name="Get Couriers ORM")
```

Приклад скрипта для навантажувального тестування за допомогою Locust

15

Рисунок Є.15 – Проведення експериментів

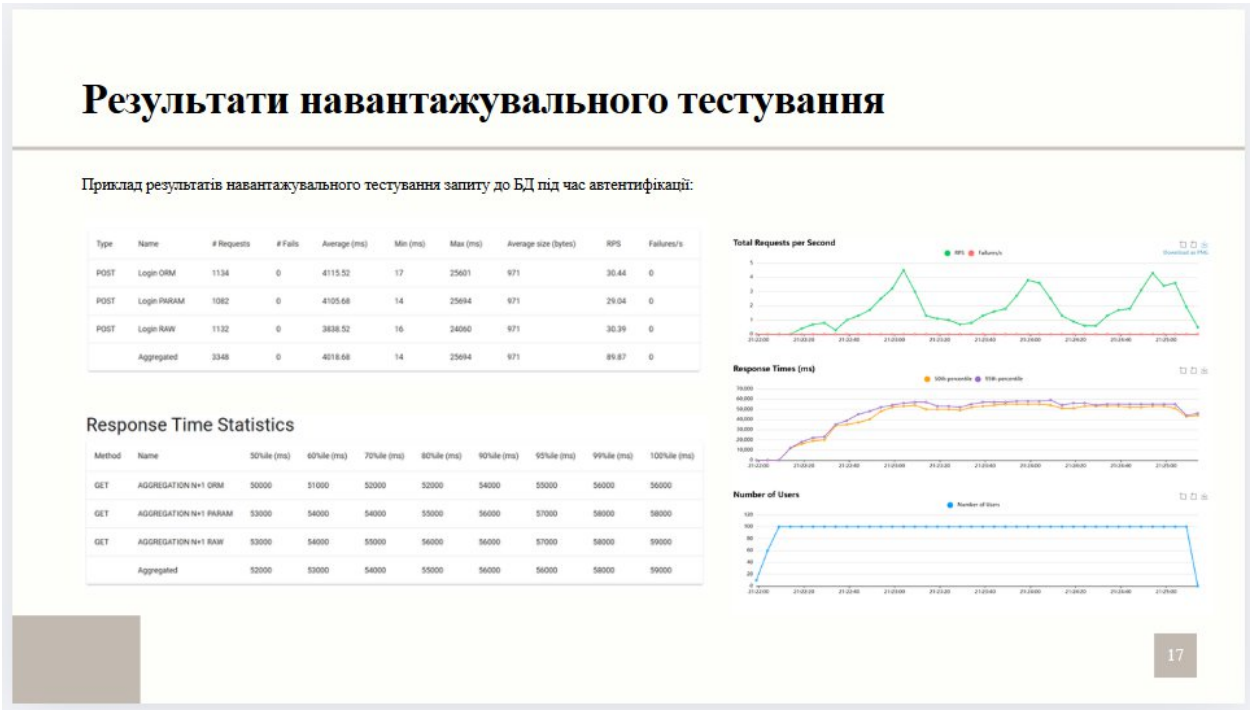
Результати багаторазового послідовного виконання запитів

Метод	Запит	Avg Time (c)	TPS	Latency (c)	CPU (%)	MEM (байт)
1	2	3	4	5	6	7
RAW	SELECT (автентифікація)	0.001208	896.46	0.001198	2.12	94.208
PARAM	SELECT (автентифікація)	0.001261	858.80	0.001251	1.50	49.152
ORM	SELECT (автентифікація)	0.001363	803.23	0.001351	0.80	42.192
RAW	INSERT	0.005748	185.77	0.001526	1.49	20.48
PARAM	INSERT	0.006003	174.47	0.001658	1.41	8.192

1	2	3	4	5	6	7
ORM	INSERT	0.005886	179.77	0.001631	1.90	4.096
RAW	SELECT	0.054644	18.91	0.001640	6.46	69550.08
PARAM	SELECT	0.063711	17.93	0.001676	5.61	34160.64
ORM	SELECT	0.154521	7.07	0.003865	5.48	18388.608
RAW	UPDATE	0.001681	673.28	0.001315	1.80	24.576
PARAM	UPDATE	0.001908	582.17	0.001504	1.51	20.48
ORM	UPDATE	0.007420	144.08	0.007384	1.19	49.008
RAW	DELETE	0.002358	549.67	0.002324	2.02	204.8
PARAM	DELETE	0.005461	192.23	0.005442	1.04	180.0
ORM	DELETE	0.010841	104.17	0.010809	0.98	279.04
RAW	JOIN	0.079007	18.58	0.002942	13.02	49176.576
PARAM	JOIN	0.075905	20.09	0.003274	13.09	15593.472
ORM	JOIN	0.151019	9.10	0.133506	19.36	25989.12
RAW	Агрегація	0.029328	35.16	0.029045	3.54	110.808
PARAM	Агрегація	0.030352	33.56	0.030103	4.04	94.208
ORM	Агрегація	0.030599	33.33	0.030552	4.09	81.92
RAW	N+1	0.071824	14.43	0.071808	1.95	33136.64
PARAM	N+1	0.111866	9.09	0.111843	3.00	33300.00
ORM	N+1	0.304199	3.35	0.303158	1.04	78979.84

16

Рисунок Є.16 – Результати багаторазового послідовного виконання запитів



Результати навантажувального тестування

Приклад результатів навантажувального тестування запиту до БД під час автентифікації:

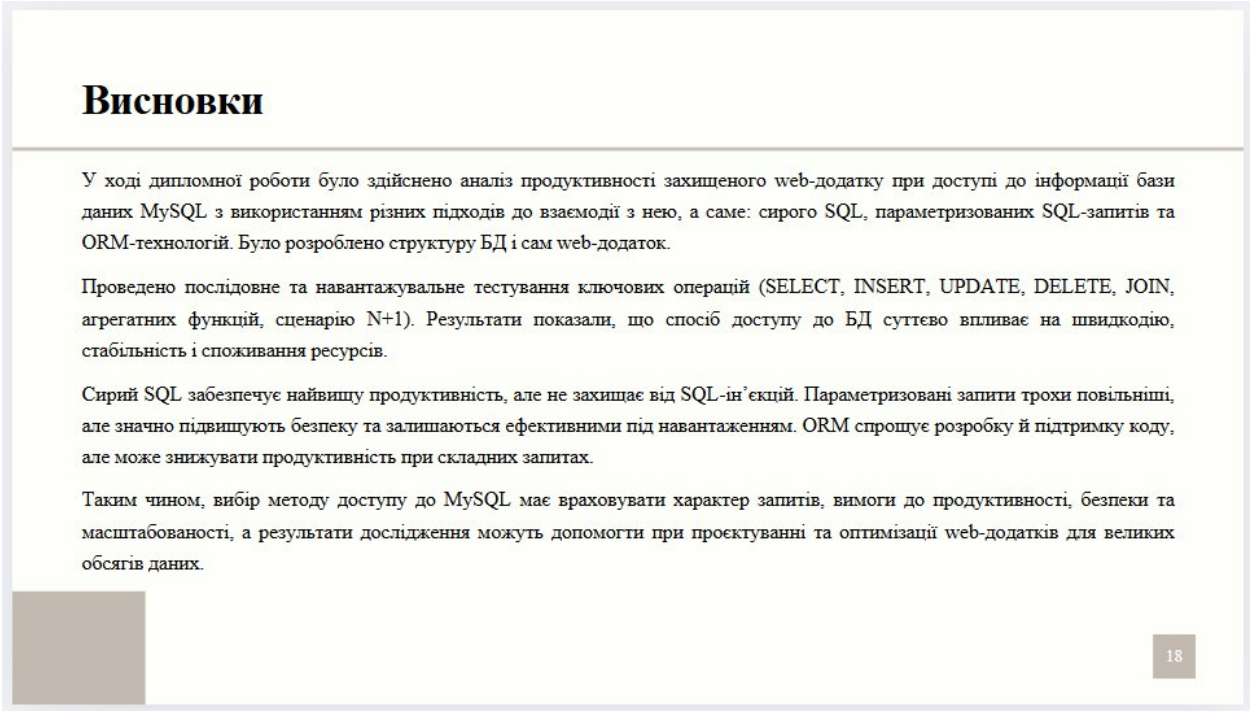
Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	Login ORM	1134	0	4115.52	17	25601	971	30.44	0
POST	Login PARAM	1082	0	4105.68	14	25694	971	29.04	0
POST	Login RAW	1132	0	3838.52	16	24060	971	30.39	0
	Aggregated	3348	0	4018.68	14	25694	971	89.87	0

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	AGGREGATION N+1 ORM	50000	51000	52000	52000	54000	55000	56000	56000
GET	AGGREGATION N+1 PARAM	53000	54000	54000	55000	56000	57000	58000	58000
GET	AGGREGATION N+1 RAW	53000	54000	55000	56000	56000	57000	58000	59000
	Aggregated	52000	53000	54000	55000	56000	56000	58000	59000

17

Рисунок Є.17 – Результати навантажувального тестування



Висновки

У ході дипломної роботи було здійснено аналіз продуктивності захищеного web-додатку при доступі до інформації бази даних MySQL з використанням різних підходів до взаємодії з нею, а саме: сирого SQL, параметризованих SQL-запитів та ORM-технологій. Було розроблено структуру БД і сам web-додаток.

Проведено послідовне та навантажувальне тестування ключових операцій (SELECT, INSERT, UPDATE, DELETE, JOIN, агрегатних функцій, сценарію N+1). Результати показали, що спосіб доступу до БД суттєво впливає на швидкодію, стабільність і споживання ресурсів.

Сирий SQL забезпечує найвищу продуктивність, але не захищає від SQL-ін'єкцій. Параметризовані запити трохи повільніші, але значно підвищують безпеку та залишаються ефективними під навантаженням. ORM спрощує розробку й підтримку коду, але може знижувати продуктивність при складних запитах.

Таким чином, вибір методу доступу до MySQL має враховувати характер запитів, вимоги до продуктивності, безпеки та масштабованості, а результати дослідження можуть допомогти при проєктуванні та оптимізації web-додатків для великих обсягів даних.

18

Рисунок Є.18 – Висновки