НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«На правах рукопису»
УДК_____

«До захисту допущено»
Завідувач кафедри
_____ Наталія АУШЕВА
“___”_____2025р.

Магістерська дисертація

на здобуття ступеня другого (магістерського) рівня вищої освіти
за освітньою програмою «Цифрові технології в енергетиці»
зі спеціальності 122 «Комп’ютерні науки»

на тему _Методи та засоби розробки CRM систем_____
_____

Виконав: студент _2_ курсу, групи ТР-43мп
Зданевич Володимир Романович _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник к.ф.-м.н., доцент Андрій ДОНЕЦЬ _____
(науковий ступінь, вчене звання, ім’я ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. ТАЕ, к.т.н, доцент Олександр СІРИЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль асистент Володимир РУДИК _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ — 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти другий (магістерський)
спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА_

«___»_____2025 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Зданевичу Володимиру Романовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації: *»Методи та засоби розробки CRM систем»*
керівник роботи *Донець Андрій Георгійович, к.ф.-м.н., доцент*
затверджені наказом по університету від «03» листопада 2025 р. № 4779-с

2. Термін подання студентом дисертації: 06.12.2025

3. Об'єкт дослідження: процес автоматизованого аналізу ефективності робітників малих та середніх компаній.

4. Вихідні дані: мова програмування Java 21, фреймворк Spring, ORM Hibernate, СУБД PostgreSQL, REST арі для HTTP-запитів.

5. Перелік завдань: 1) проаналізувати існуючі програмні рішення для CRM систем, виявити їхні переваги і недоліки; 2) провести аналіз методів оцінки ефективності персоналу; 3) дослідити провідні методи та підходи до розробки систем і обґрунтувати вибір оптимального рішення для інтеграції у вебзастосунок; 4) розробити структуру бази даних та архітектуру системи; 5) розробити програмне забезпечення з використанням обраних технологій та інструментів, виконати тестування розробленої системи, виправити знайдені недоліки.

6. Орієнтований перелік ілюстративного матеріалу: архітектура вебзастосунку, діаграма взаємодії компонентів застосунку між собою, схема архітектурних підходів, інтерфейси користувача, таблиці стартап-проєкту.

7. Орієнтовний перелік публікацій: 1. Моніторинг ефективності діяльності підприємств в енергетичній галузі. Збірник наукових тез з матеріалами III Міжнародної науково-практична конференції «Сучасні аспекти інженерії програмного забезпечення» MoSE-2025, КПІ, 2025. С.43-44

8. Дата видачі завдання «16» вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів підготовки магістерської дисертації	Термін виконання	Примітка
1	Аналіз методів розробки програмного забезпечення за літературними джерелами	18.09.2025	виконано
2	Вибір та налаштування засобів розробки програмного забезпечення	21.09.2025	виконано
3	Розроблення структури бази даних та архітектури системи	26.09.2025	виконано
4	Розроблення програмного забезпечення	13.10.2025	виконано
5	Тестування розробленої системи, виправлення недоліків	19.10.2025	виконано
6	Захист створеного програмного забезпечення	20.10.2025	виконано
7	Оформлення магістерської дисертації	25.11.2025	виконано
8	Передзахист	27.11.25	виконано
9	Подання матеріалів магістерської дисертації на кафедру	04.12.25	виконано
10	Захист	11.12.25	виконано

Студент _____ **Володимир ЗДАНЕВИЧ**
(підпис) (ім'я, ПРІЗВИЩЕ)

Керівник роботи _____ **Андрій ДОНЕЦЬ**
(підпис) (ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Актуальність теми зумовлена необхідністю створення високопродуктивних та гнучких інформаційних систем для управління взаємовідносинами з клієнтами в умовах цифрової економіки. Проблема полягає у високій вартості та обмеженій гнучкості комерційних рішень, що обґрунтовує доцільність розробки власного масштабованого прототипу.

Метою роботи є визначення та апробація оптимальних архітектурних підходів і технологічних засобів для розробки CRM-систем, шляхом створення функціонального прототипу панелі управління з реалізацією багатокористувацького режиму. Для досягнення мети було поставлено п’ять основних завдань. Дослідження та аналіз існуючих методів розробки корпоративного програмного забезпечення та архітектурних стилів, аналіз та обґрунтування вибору математичних методів, зокрема реляційної алгебри, для моделювання даних та реалізації механізму багатокористувацької фільтрації, дослідження та розробка алгоритмів бізнес-логіки, наприклад алгоритму зміни статусу завдання, аналіз та опис програмної реалізації, а також демонстрація функціоналу, проведення обчислювальних експериментів та оцінка переваг власної розробки.

Об’єктом дослідження визначено процес розробки корпоративного веб-додатку.

Предметом дослідження є методики, технологічні засоби та алгоритмічні рішення, що забезпечують ефективність та надійність розробки багатокористувацьких CRM-систем.

Методи дослідження: методи системного аналізу для дослідження структури системи, методи теорії баз даних та реляційної алгебри для моделювання даних та фільтрації, методи об’єктно-орієнтованого аналізу для моделювання класів, а також експериментальні методи для проведення тестів швидкодії.

Вихідні дані: мова програмування Java 21, фреймворк Spring, ORM Hibernate, СУБД PostgreSQL, REST api для HTTP-запитів.

Практичне значення отриманих результатів: розроблений вебзастосунок автоматизує процеси аналізу продуктивності співробітників різних відділів малих та середніх компаній. Система реалізує гнучке налаштування параметрів ефективності та наглядну демонстрацію результатів моніторингу показників. Проєкт готовий до впровадження як стартап на ринку CRM систем України з низькими технологічними бар'єрами входу.

Апробація результатів дисертації. Основні положення даної роботи з урахуванням специфіки енергетичної галузі доповідались та обговорювались на 3-й міжнародно-науковій конференції «Сучасні аспекти інженерії програмного забезпечення» (НТУУ «КПІ ім.Ігоря Сікорського, 12-13 листопада 2025 р.).

Дисертація складається зі вступу, п'ятьох розділів, висновків до розділів, загальних висновків, списку використаних джерел. Основний вміст роботи викладено на 78 сторінках, а повний обсяг складає 87 сторінок, які містять 9 рисунків, 3 таблиці, 40 джерел і 1 додаток.

Ключові слова: CRM-система, Spring Boot, RESTful API, PostgreSQL, JPA, Hibernate, Java, архітектура, алгоритми.

ABSTRACT

The relevance of the topic is determined by the need to create highly productive and flexible information systems for managing customer relationships in the digital economy. The problem lies in the high cost and limited flexibility of commercial solutions, which justifies the development of a proprietary scalable prototype.

The aim of the work is to identify and test optimal architectural approaches and technological means for developing CRM systems by creating a functional prototype of a control panel with multi-user mode implementation. Five main tasks were set to achieve this goal. Research and analysis of existing methods for developing corporate software and architectural styles, analysis and justification of the choice of mathematical methods, in particular relational algebra, for data modelling and implementation of a multi-user filtering mechanism, research and development of business logic algorithms, such as a task status change algorithm, analysis and description of software implementation, as well as demonstration of functionality, conducting computational experiments and evaluating the advantages of our own development.

The object of the study is the process of developing a corporate web application.

The subject of the study is methods, technological means and algorithmic solutions that ensure the efficiency and reliability of developing multi-user CRM systems.

Research methods: methods of system analysis for studying the structure of the system, methods of database theory and relational algebra for data modelling and filtering, methods of object-oriented analysis for class modelling, as well as experimental methods for conducting performance tests.

Initial data: Java 21 programming language, Spring framework, Hibernate ORM, PostgreSQL DBMS, REST API for HTTP requests.

Practical significance of the results obtained: the developed web application automates the processes of analysing the performance of employees in various departments of small and medium-sized companies. The system implements flexible configuration of performance parameters and a clear demonstration of the results of

monitoring indicators. The project is ready for implementation as a start-up in the Ukrainian CRM systems market with low technological barriers to entry.

Approval of the dissertation results. The main provisions of this work, taking into account the specifics of the energy industry, were reported and discussed at the 3rd International Scientific Conference «Modern Aspects of Software Engineering» (NTUU «Igor Sikorsky KPI», 12-13 November 2025).

The dissertation consists of an introduction, five chapters, conclusions to the chapters, general conclusions, and a list of references. The main content of the thesis is presented on 78 pages, while the total volume comprises 87 pages, including 9 figures, 3 tables, 40 references, and 1 appendix.

Keywords: CRM system, Spring Boot, RESTful API, PostgreSQL, JPA, Hibernate, Java, architecture, algorithms.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

CRM — означає Customer Relationship Management, тобто Управління взаємовідносинами з клієнтами

API — це скорочення від Application Programming Interface

REST — це Representational State Transfer

JPA — Java Persistence API

CRUD — використовується для опису базових операцій Create, Read, Update, Delete

ORM — означає Об’єктно-реляційне відображення

HTML — HyperText Markup Language

СМБ — Середні та малі бізнеси

ВСТУП

Процеси цифровізації та глобалізація ринків зумовлюють зростаючу необхідність для підприємств у впровадженні ефективних систем управління взаємовідносинами з клієнтами. CRM-системи сьогодні є стратегічним інструментом, що дозволяє накопичувати, аналізувати та ефективно використовувати дані. Актуальність теми полягає в необхідності розробки надійного та масштабованого програмного забезпечення, здатного забезпечити конфіденційність даних у багатокористувацькому середовищі. Критичний аналіз існуючих комерційних рішень показав їхню високу вартість та часто надмірну складність, що робить актуальною розробку більш гнучкого та економічно доцільного прототипу на сучасних корпоративних технологіях.

Метою дослідження є визначення та апробація оптимальних архітектурних підходів і технологічних засобів для розробки CRM-систем, шляхом створення функціонального прототипу панелі управління з реалізацією багатокористувацького режиму. При цьому користувачами розглядалися саме учасники та оператори енергетичного ринку України, де на превеликий жаль застосування CRM-підходу ще не стало нормою [1].

Для досягнення цієї мети було поставлено п’ять основних завдань. Першим завданням було дослідження та аналіз існуючих методів розробки корпоративного програмного забезпечення та архітектурних стилів, таких як REST та Трирівнева архітектура. Друге завдання включало аналіз та обґрунтування вибору математичних методів, зокрема реляційної алгебри, для моделювання даних та реалізації механізму багатокористувацької фільтрації. Третє завдання стосувалося дослідження та розробки алгоритмів бізнес-логіки, наприклад алгоритму зміни статусу завдання. Четверте завдання полягало в аналізі та описі програмної реалізації, включаючи архітектуру системи, модель представлення даних через JPA-сутності та діаграму класів. П’яте завдання передбачало демонстрацію функціоналу, проведення обчислювальних експериментів та оцінку переваг власної розробки.

Об’єктом дослідження визначено процес розробки корпоративного веб-додатку, що відповідає об’єкту дослідження спеціальності «Комп’ютерні науки». Предметом дослідження є методики, технологічні засоби та алгоритмічні рішення, що забезпечують ефективність та надійність розробки багатокористувацьких CRM-систем.

Для досягнення мети застосовано кілька наукових методів. Метод системного аналізу використано для дослідження структури CRM-систем; методи теорії баз даних і реляційної алгебри для проєктування моделі даних та обґрунтування механізму багатокористувацької фільтрації. Методи об’єктно-орієнтованого аналізу й проєктування застосовано для моделювання класів Controller, Service та Repository й розділення відповідальності. Експериментальні методи використано для тестів швидкодії RESTful API та оцінки ефективності алгоритму фільтрації даних.

Наукова новизна отриманих результатів полягає в інтеграції новітніх версій корпоративних технологій, зокрема Java 21 та Spring Boot 3.5.6, у класичну трирівневу архітектуру для вирішення задачі персоналізації даних у багатокористувацькому середовищі через логічний механізм фільтрації на рівні репозиторіїв [2-4].

Практичне значення полягає у створенні повністю функціонального прототипу CRM-панелі, який може слугувати економічно вигідною основою для комерційного продукту, легко адаптованого до унікальних бізнес-процесів замовника, оскільки він заснований на гнучкому та сучасному стеку технологій.

Перший розділ містить постановку задачі та огляд методів її розв’язання. У другому розглянуто математичні й програмні засади розробки. Третій розділ присвячено програмній реалізації CRM-системи, четвертий — результатам досліджень і аналізу створеного рішення, а п’ятий — детальному опису стартап-проєкту, розробленого в межах теми дисертації.

1 ПОСТАНОВКА ЗАДАЧІ ДИСЕРТАЦІЇ ТА ОГЛЯД МЕТОДІВ РОЗРОБКИ CRM СИСТЕМ

Метою цього розділу є комплексне формулювання наукової задачі дисертації, проведення ґрунтовного обґрунтування її актуальності з точки зору сучасних тенденцій ринку, а також критичний огляд існуючих архітектурних рішень, методів та програмних засобів, які є необхідними для ефективного розв’язання поставленої задачі. Це закладає теоретичний фундамент для подальшої практичної реалізації системи.

1.1 Актуальність теми та постановка задачі дисертації

Актуальність теми дисертаційного дослідження визначається комплексом стрімких динамічних змін на ринку корпоративних інформаційних систем та безперервно зростаючими, часто суперечливими, вимогами до програмних рішень. Ці вимоги охоплюють необхідність підвищення рівня безпеки, забезпечення персоналізації робочого простору для кожного користувача та, водночас, досягнення максимальної економічної обґрунтованості впровадження.

На сучасному етапі комерційні CRM-системи, які традиційно домінують на глобальному ринку, хоча і характеризуються надзвичайно широким функціоналом та потужними можливостями інтеграції, стикаються з низкою критичних недоліків, що обмежують їхнє масове впровадження, особливо у сегменті малого та середнього бізнесу (МСБ).

Насамперед, ці рішення є надмірно дорогими. Висока вартість ліцензійних відрахувань, які часто базуються на кількості користувачів або обсязі даних, стає непосильною перепоною для невеликих компаній з обмеженим IT-бюджетом. Крім того, внутрішня архітектура комерційних платформ часто виявляється мало гнучкою та надто складною для адаптації до унікальних та специфічних бізнес-потреб МСБ. Складність інтеграції та обслуговування таких систем вимагає залучення висококваліфікованих, а отже, дорогих спеціалістів. Додатковою

стримуючою обставиною є непрозорий, закритий технологічний стек (vendor lock-in), який створює ризик залежності від одного постачальника та ускладнює перехід на інші платформи. Усі ці фактори у сукупності стримують масове та економічно виправдане впровадження потужних CRM-інструментів у невеликих компаніях.

Саме тому виникає нагальна науково-практична потреба в розробці альтернативного CRM-рішення. Таке рішення має поєднати надійність, стабільність та довготривалу підтримку, притаманну корпоративному стеку, з гнучкістю, відкритою моделлю розробки та економічною доцільністю.

Вибір технологічної платформи є прямою відповіддю на визначену актуальність. Використання Java-платформи та її екосистеми, включно з фреймворком Spring Boot, є загальновизнаним індустріальним стандартом, що гарантує високу масштабованість, стабільність роботи та довготривалу підтримку системи[1]. Технології з відкритим кодом, як-от Spring, забезпечують прозорість, можливість адаптації та усунення залежності від дорогих комерційних ліцензій[2]. Таким чином, розробка на основі корпоративного стеку дозволяє створити продукт, що має надійність великих систем, але при цьому доступний та гнучкий для сегменту МСБ.

Незважаючи на обраний технологічний стек, ключовою науковою та інженерною проблемою, яку необхідно вирішити в рамках цього дослідження, залишається ефективна та безпечна реалізація принципу багатокористувацькості (Multi-tenancy).

Реалізація багатокористувацькості в єдиній програмній системі вимагає розробки такого архітектурного підходу, який би гарантував абсолютну логічну ізоляцію даних різних користувачів або організацій в межах спільного фізичного середовища (однієї бази даних та одного екземпляра застосунку). Ця проблема не є тривіальною, оскільки вона вимагає балансу між двома протилежними вимогами. Простота архітектури та підтримки. Ізоляція має бути досягнута без критичного ускладнення загальної архітектури, яка повинна залишатися простою для обслуговування та розвитку. А також продуктивність та масштабованість. Впровадження механізму розділення даних не повинно мати негативного впливу

на швидкість відгуку системи, навіть при значному зростанні обсягу даних та кількості користувачів.

Суть проблеми полягає у необхідності інтеграції механізму розділення та фільтрації даних безпосередньо у рівень доступу до даних (Persistence Layer) таким чином, щоб цей механізм працював автоматично, прозоро для бізнес-логіки та з мінімальними обчислювальними витратами. Невірне вирішення цієї проблеми призведе або до архітектурно складного, дорогого рішення, або до небезпечного з точки зору цілісності даних.

Виходячи з визначеної актуальності та сформульованої наукової проблеми, постановка задачі дисертації полягає у розробці та верифікації прототипу CRM-панелі управління, який відповідає високим корпоративним стандартам, але при цьому є економічно ефективною альтернативою для МСБ.

Прототип повинен забезпечувати повний цикл управління трьома основними бізнес-сутністями. Управлінням профілями користувачів та правами доступу. Управлінням ієрархічною структурою організації. А також управлінням робочими процесами, їхніми статусами та термінами виконання.

Центральним та найбільш складним технічним завданням дисертації є проектування та реалізація високопродуктивного механізму багатокористувацької фільтрації даних. Цей механізм має бути реалізований на основі математичного апарату реляційної алгебри, зокрема, операції селекції.

Механізм фільтрації повинен бути налаштований таким чином, щоб гарантувати, що кожен унікальний користувач бачить виключно той набір даних, до якого він має легітимний доступ, і не має доступу до даних інших користувачів. Ефективність цього механізму, виміряна часом відгуку та стійкістю під навантаженням, є запорукою безпеки та масштабованості розроблюваної багатокористувацької системи. Таким чином, дисертаційне дослідження концентрується на інженерному втіленні та емпіричній верифікації архітектурних рішень, що дозволяють вирішити проблему ефективної ізоляції даних у спільному корпоративному середовищі.

1.2 Огляд і критичний аналіз архітектурних рішень для корпоративних систем

Для побудови надійної, масштабованої та економічно обґрунтованої корпоративної системи необхідно провести ґрунтовний огляд та критичний аналіз існуючих архітектурних рішень. Вибір архітектури є стратегічним рішенням, яке визначає ремонтопридатність, гнучкість, довгострокову ефективність та загальні експлуатаційні витрати майбутнього програмного продукту. В межах даного дослідження було розглянуто три основні архітектурні парадигми: монолітну, мікросервісну та трирівневу. Монолітна архітектура (Monolithic Architecture) передбачає, що вся функціональність програмної системи, включаючи інтерфейс, бізнес-логіку та шар доступу до даних, зібрана в єдиному, нероздільному коді та розгортається як один великий виконуваний файл[3]. Для наглядного розуміння відмінностей на рисунку 1.1 проілюстровано різницю між монолітною та мікросервісною архітектурами.

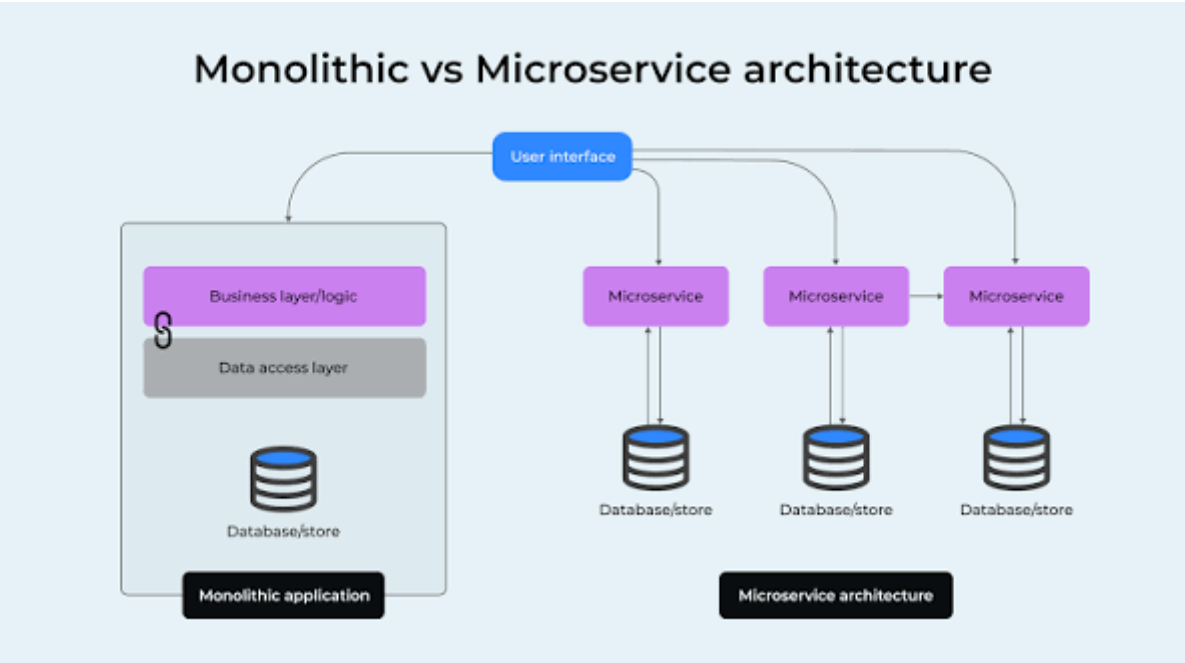

Рисунок 1.1 — Порівняння архітектур

Цей підхід має суттєву перевагу на початкових етапах розробки для невеликих проєктів, оскільки вся база коду знаходиться в єдиному місці. Це спрощує первинне розгортання, налагодження та управління залежностями. Однак, моноліт виявляється неефективним у довгостроковій перспективі та не відповідає вимогам до сучасних корпоративних систем, що зростають.

Критичні недоліки монолітної архітектури включають складність масштабування. Моноліт критично ускладнює горизонтальне та вертикальне масштабування системи. Для збільшення пропускної здатності навіть невеликого компонента необхідно тиражувати весь застосунок, що є неефективним використанням обчислювальних ресурсів.

Низька ремонтопридатність. При значному зростанні функціоналу та збільшенні бази користувачів, обслуговування моноліту стає надзвичайно складним через високу зв’язаність компонентів. Зміни в одній частині коду можуть непередбачувано вплинути на інші функціональні області.

Високий ризик розгортання. Процес внесення змін і розгортання нових версій стає повільним, дорогим і ризикованим. Відмова одного, навіть невеликого компонента, часто призводить до відмови всієї системи (Single Point of Failure), що є неприпустимим для CRM-системи, яка має забезпечувати безперервність бізнес-процесів[4].

Таким чином, незважаючи на початкову простоту, монолітна архітектура була відхилена як невідповідна вимогам до гнучкості та надійності, необхідних для економічно ефективного масштабування МСБ.

Мікросервісна архітектура (Microservices Architecture) є антиподом моноліту і передбачає, що застосунок розкладається на набір невеликих, незалежних сервісів, кожен з яких реалізує окремий бізнес-функціонал і може бути розроблений, розгорнутий та масштабований окремо[5].

Цей підхід забезпечує найвищий ступінь гнучкості, технологічної незалежності та дозволяє незалежно розгортати кожен сервіс. Це ідеально підходить для дуже великих, розподілених систем, які вимагають постійного та швидкого розвитку різних функціональних областей.

Проте, мікросервісна архітектура має істотні експлуатаційні та інфраструктурні недоліки, які роблять її недоцільною для економічно чутливого сегменту МСБ.

Мікросервіси вимагають значних інвестицій у складну інфраструктуру для їхнього управління та підтримки. Це включає необхідність використання інструментів управління контейнерами, таких як Docker та Kubernetes, для оркестрації сервісів, а також складних систем моніторингу, логування та трасування розподілених транзакцій[6][7].

Складність мікросервісного середовища створює високий поріг входу для команди розробників і значні експлуатаційні витрати на підтримку інфраструктури. Ці витрати часто перевищують економічну вигоду для компаній, які не мають мільйонів користувачів.

Оскільки дисертаційна робота орієнтована на створення економічно доцільного рішення для МСБ, архітектура мікросервісів, через свою надмірну складність та високі інфраструктурні вимоги, була відхилена як така, що не відповідає критеріям цільової аудиторії

Як оптимальний компроміс між простотою моноліту та гнучкістю мікросервісів, була обрана Трирівнева архітектура (Three-Tier Architecture)[8]. Цей архітектурний шаблон є загальновизнаним стандартом для середніх за складністю корпоративних веб-додатків.

Трирівнева архітектура забезпечує чітке розділення відповідальності між трьома логічними та фізичними шарами. Рівень Представлення (Presentation Tier)[9]. Ізольований шар, відповідальний виключно за взаємодію з клієнтом (браузером).

Рівень Застосунку (Application/Business Logic Tier)[10]. Це центральний шар, який інкапсулює всю бізнес-логіку та управління транзакціями. Та рівень даних (Data Tier)[11]. Шар, який є єдиним місцем для роботи з базою даних та стійким зберіганням інформації. Для наглядності схему архітектури продемонстровано на рисунку 1.2.

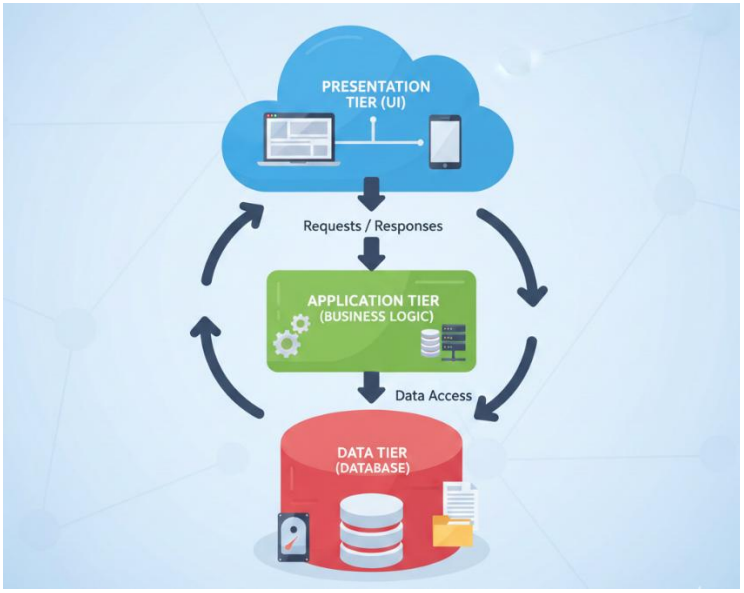

Рисунок 1.2 — Трирівнева архітектура

Цей вибір дозволяє незалежно оптимізувати та масштабувати кожен рівень. Збільшення навантаження на інтерфейс (зростання кількості клієнтів) може бути компенсоване масштабуванням Рівня Представлення, тоді як зростання обсягу даних вирішується оптимізацією Рівня Даних. Така архітектура є ідеальною для середніх за складністю корпоративних додатків, оскільки вона забезпечує достатню гнучкість без інфраструктурної складності мікросервісів.

Взаємодія між обраними рівнями побудована за архітектурним стилем REST (Representational State Transfer)[12]. REST є стандартом для створення веб-сервісів, що базуються на протоколі HTTP, і він вносить вирішальний вклад у масштабованість системи[13]. Для наглядності на рисунку 1.3 продемонстрована схема REST API.

Ключова перевага REST, що має вирішальне значення для забезпечення високої продуктивності та масштабування, полягає у бездержавності (Statelessness). Цей принцип означає, що сервер не зберігає жодних даних про стан сесії між послідовними запитами. Кожен запит, який надходить від клієнта до бекенду, є повністю самодостатнім, оскільки містить всю необхідну інформацію (автентифікаційні токени, ідентифікатори користувача) для його обробки.

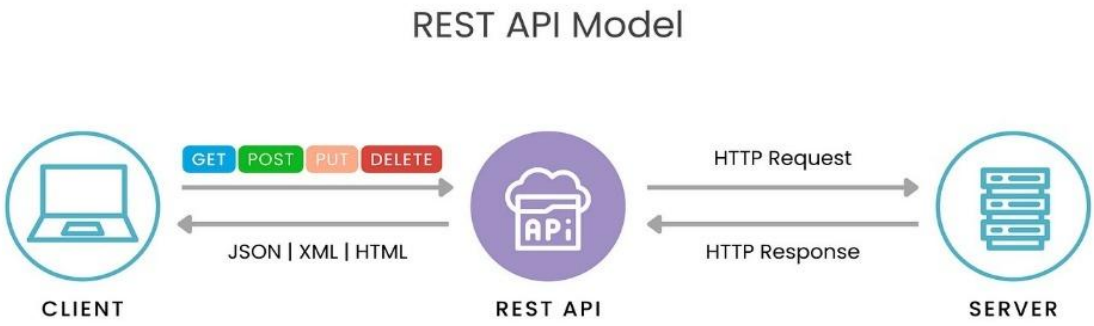

Рисунок 1.3 — Схема REST API

Ця властивість значно спрощує горизонтальне масштабування Рівня Застосунку. Оскільки стан клієнта не прив'язаний до конкретного екземпляра сервера, будь-який вільний екземпляр бекенд-сервера може обробити будь-який вхідний запит, незалежно від того, який сервер обробляв попередні запити цього ж клієнта.

Це дозволяє легко розміщувати екземпляри сервера за балансувальником навантаження (Load Balancer). Таким чином, принцип бездержавності є критичною вимогою для забезпечення високої доступності та горизонтальної масштабованості високопродуктивної CRM-системи, що дозволяє системі ефективно справлятися з піковими навантаженнями.

У підсумку, комбінація Трирівневої архітектури з принципами RESTful є оптимальним інженерним рішенням, що забезпечує необхідний баланс між ремонтопридатністю, масштабованістю та економічною доцільністю для цільового сегменту малого та середнього бізнесу.

1.3 Огляд методів та програмних засобів, необхідних для розв'язання поставлених завдань

Для успішної реалізації поставленого завдання — розробки прототипу CRM-панелі з гарантованою багатокористувацькою ізоляцією — було обрано

технологічний стек, який відповідає високим корпоративним стандартам, забезпечує необхідну надійність, швидкість розробки та довготривалу підтримку. Вибір програмних засобів базується на критичному аналізі їхньої ефективності, масштабованості та економічної доцільності, що є ключовим для цільового сегменту малого та середнього бізнесу.

Для створення Рівня Застосунку та реалізації основної бізнес-логіки системи обрано мову Java 21 у поєднанні з фреймворком Spring Boot 3.5.6[14]. Використання Java як основної мови гарантує високу надійність та продуктивність, які забезпечуються її віртуальною машиною JVM (Java Virtual Machine)[15]. JVM виконує роль незалежного середовища, що забезпечує переносимість коду та його стабільну роботу на різних операційних системах. Додатковою перевагою є доступ до великої та зрілої екосистеми корпоративних інструментів та бібліотек.

Spring Boot був обраний за його виняткову здатність різко прискорювати процес розробки[16]. Це досягається завдяки механізму автоконфігурації, який автоматично налаштовує стандартні компоненти та залежності на основі підключених бібліотек. Цей підхід мінімізує рутинні налаштування, що дозволяє розробнику максимально зосередитися на безпосередньому вирішенні бізнес-вимог, а не на конфігураційних файлах. Spring, будучи архітектурним каркасом, підтримує принципи Інверсії Керування (IoC) та Впровадження Залежностей (DI), що забезпечує слабку зв’язаність компонентів та легкість їхнього тестування[17].

Управління даними реалізується через поєднання надійної реляційної СУБД та передового ORM-інструментарію. Обрано реляційну СУБД PostgreSQL, яка є потужною, відкритою системою управління базами даних[18]. PostgreSQL відома своєю надійністю, стійкістю до високих навантажень та повною підтримкою всіх необхідних механізмів забезпечення цілісності даних, що є критичним для корпоративної системи. Це включає підтримку складних транзакцій з гарантіями ACID-властивостей, а також механізми індексування, необхідні для швидкої багатокористувацької фільтрації[19].

Для взаємодії з цією СУБД використовується бібліотека Spring Data JPA (Java Persistence API) у поєднанні з Hibernate ORM як її провідною реалізацією[20]. JPA

виступає як інтерфейс для об’єктно-реляційного відображення, а Hibernate виконує фактичну роботу по перетворенню об’єктів Java на реляційну модель та навпаки[21]. Цей підхід дозволяє відображати бізнес-об’єкти (JPA-сутності) безпосередньо на таблиці бази даних, що значно мінімізує необхідність ручного написання та підтримки SQL-коду, підвищуючи продуктивність розробки.

Spring Data JPA додатково спрощує роботу, надаючи готові шаблони для репозиторіїв та стандартних операцій доступу до даних. Це також дозволяє декларативно створювати складні методи запитів, необхідні для реалізації багатокористувацької ізоляції (наприклад, findByUserId), автоматично генеруючи оптимізований SQL.

Внутрішня архітектура коду в Рівні Застосунку побудована за класичним патерном Controller-Service-Repository (рис 1.4)[22].

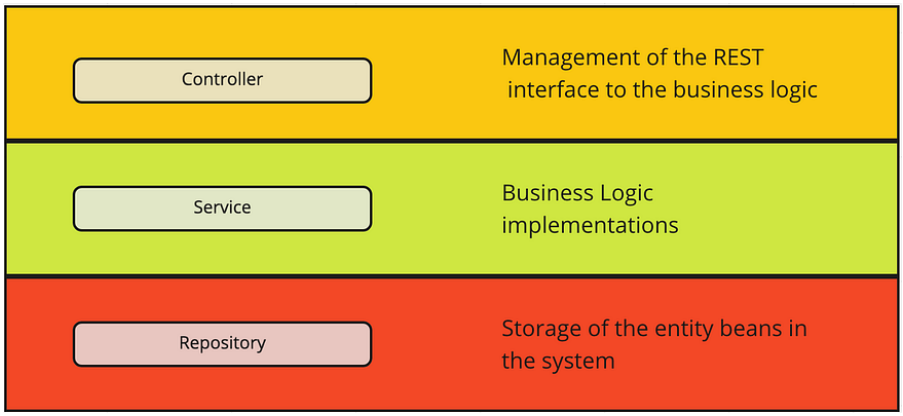

Рисунок 1.4 — Рівні застосунку

Цей метод об’єктно-орієнтованого проектування є фундаментальним для забезпечення ремонтопридатності системи. Контролер (Controller) відповідає виключно за прийом вхідних HTTP-запитів, обробку вихідних даних та стандартизацію API-комунікації. Сервіс (Service) інкапсулює всі бізнес-правила та алгоритми. Саме на цьому рівні буде реалізована складна логіка багатокористувацькості, включаючи верифікацію прав доступу та логіку переходу

статусів. Репозиторій (Repository) інкапсулює логіку взаємодії з базою даних, абстрагуючи Сервісний рівень від специфіки SQL.

Таке чітке розділення функціоналу є критичним для забезпечення юніт-тестування, оскільки кожен компонент може бути протестований незалежно. Це значно підвищує надійність системи та прискорює процес ітераційної розробки.

Клієнтський інтерфейс (Рівень Представлення) реалізовано з використанням стандартних веб-технологій. Для забезпечення динаміки та обробки подій використовується JavaScript, а для стилізації обрано Tailwind CSS[23]. Tailwind CSS, як утилітарний фреймворк, дозволяє швидко та адаптивно створювати сучасний інтерфейс[24].

Взаємодія між фронтендом та бекендом (Рівнем Застосунку) здійснюється через методи асинхронної взаємодії AJAX, зокрема з використанням сучасного Fetch API[25]. Це дозволяє динамічно оновлювати частини користувацького інтерфейсу після отримання даних у форматі JSON, уникнувши повного перезавантаження сторінки[26]. Такий підхід забезпечує високу швидкість відгуку системи, сучасний користувацький досвід та повністю відповідає архітектурному стилю RESTful API, обраному для цього проєкту.

Таким чином, обраний технологічний стек — Java/Spring Boot, PostgreSQL/JPA, Controller-Service-Repository, RESTful API — становить цілісний та високопродуктивний набір методів та програмних засобів, необхідних для ефективного розв’язання поставлених завдань, зокрема, реалізації складної багатокористувацької логіки.

1.4 Механізми реалізації багатокористувацькості та високопродуктивної фільтрації даних

Реалізація ефективного та безпечного механізму багатокористувацькості є центральною інженерною задачею, що безпосередньо випливає з наукової проблеми забезпечення безпеки даних в економічно доцільній архітектурі.

Оскільки економічна ефективність є ключовою вимогою для CRM-рішення, орієнтованого на малий та середній бізнес, для реалізації багатокористувацькості обрана архітектурна модель спільної бази даних з логічно ізольованими записами (Shared Database, Separate Schema/Data).

Вибір моделі спільної бази даних є найбільш економічно доцільним підходом для цільового сегменту. У цій моделі всі користувачі системи використовують єдину фізичну інсталяцію бази даних, а їхні записи зберігаються в одних і тих же таблицях. Це мінімізує експлуатаційні витрати, оскільки немає потреби в інфраструктурному управлінні численними базами даних або схемами, як це вимагається в інших, більш дорогих моделях ізоляції.

Проте, такий підхід вимагає бездоганного механізму забезпечення логічної ізоляції. Ключова проблема полягає у необхідності гарантувати, що кожен запит на вибірку, оновлення чи видалення даних, який надходить у систему, обов'язково та автоматично містить умову фільтрації. Ця умова має обмежувати доступ лише до тих записів, які асоційовані з ідентифікатором поточного користувача. Будь-яка помилка в застосуванні цього фільтра призведе до критичного порушення безпеки та конфіденційності даних.

Для розв'язання проблеми в рамках дисертаційного дослідження розробляється спеціальний високопродуктивний механізм фільтрації даних. Центральним елементом цього механізму є обов'язковий атрибут userId, інтегрований у всі сутності, що підлягають ізоляції (наприклад, Task та Department).

Інтеграція цього механізму є принциповою: вона здійснюється на Рівні Даних системи, а саме у Repository за допомогою Spring Data JPA та Hibernate ORM. Інтеграція на цьому низькому рівні є стратегічно важливою, оскільки вона дозволяє забезпечити автоматичне та незмінне застосування фільтрів до всіх запитів, незалежно від того, як вони були сформовані у Рівні Сервісу.

Це інженерне рішення забезпечує двоякий виграш. З однієї сторони усунення ризику людської помилки. Завдяки інтеграції у ORM-шар, ризик того, що розробник забуде додати умову фільтрації до запиту, повністю усувається. Це

значно підвищує загальну надійність, безпеку та цілісність системи. А з іншої Прозорість для бізнес-логіки**.** Рівень Сервісу може працювати з об’єктами так, ніби він є єдиним орендарем, оскільки механізм ізоляції працює прозоро на нижчому рівні. Це зберігає чистоту коду бізнес-логіки.

Ключовим фактором, що забезпечує високу продуктивність при використанні моделі спільної бази даних, є коректне використання можливостей реляційної СУБД (PostgreSQL). Механізм фільтрації, що базується на умові $\text{userId} = \text{id}_{current}$, вимагає, щоб база даних могла швидко знаходити потрібні записи.

Правильна реалізація цього механізму включає обов’язкове індексування поля userId у всіх таблицях, які підлягають ізоляції. Цей підхід забезпечує, що операція селекції (σ), яка лежить в основі фільтрації, виконується з високою швидкістю.

Завдяки індексу, СУБД здатна ефективно розділяти дані між різними користувачами, виконуючи пошук за логарифмічний час ($O(\log N)$) замість лінійного сканування, навіть при значній кількості даних. Ця оптимізація є критичною для забезпечення необхідної швидкості відгуку високопродуктивної CRM-системи.

Таким чином, розробка та інтеграція цього високопродуктивного механізму фільтрації є центральною інженерною задачею дисертації, яка успішно вирішує наукову проблему забезпечення безпечної багатокористувацькості, створюючи при цьому економічно доцільну та масштабовану архітектуру.

Висновки до розділу 1

У процесі роботи над цим розділом було проведено всебічний аналіз сучасних вимог до корпоративних інформаційних систем та закладено теоретико-методологічний фундамент для подальшого дисертаційного дослідження. Досягнуті ключові результати дозволяють перейти до наступних етапів детального

проектування та програмної реалізації системи, маючи чітко обґрунтовані архітектурні та технологічні рішення.

Проведений критичний аналіз актуальності теми однозначно підтвердив гостру необхідність створення нового програмного рішення для ринку малого та середнього бізнесу (МСБ). Актуальність полягає у потребі в економічно ефективному, але водночас надійному та безпечному багатокористувацькому CRM-рішенні. Існуючі комерційні аналоги були визнані архітектурно надмірно складними та фінансово недоступними, що є вагомим чинником, який стримує їхнє масове та економічно виправдане впровадження.

З огляду на необхідність високої надійності та довготривалої підтримки, обґрунтовано вибір корпоративного стеку Java та Spring Boot як технологічної основи. Цей вибір забезпечує необхідну стабільність, високу продуктивність та доступ до розвиненої екосистеми, відповідаючи індустріальним стандартам корпоративної розробки.

На основі цього аналізу чітко сформульовано основну задачу дисертації. Вона полягає у розробці прототипу CRM-панелі управління, що охоплює управління ключовими сутностями («Користувач», «Відділ», «Завдання»), та, що найважливіше, у проектуванні та реалізації високопродуктивного механізму фільтрації даних. Цей механізм має стати гарантом безпечної ізоляції даних між різними користувачами в умовах спільної інфраструктури. Таким чином, центральною науково-практичною проблемою дослідження визначено забезпечення безпечної багатокористувацькості в архітектурі, оптимізованій за критерієм економічної доцільності.

Було проведено ґрунтовний порівняльний аналіз архітектурних рішень (монолітної, мікросервісної) для вибору оптимальної парадигми. Результати цього аналізу показали, що Трирівнева архітектура є найбільш доцільною для розв’язання поставлених завдань. Трирівнева модель пропонує оптимальний баланс між простотою розгортання та підтримки (на відміну від мікросервісів, які вимагають складної інфраструктури Kubernetes) та необхідною гнучкістю масштабування (на

відміну від моноліту, який критично ускладнює незалежний розвиток компонентів).

Обґрунтовано також вибір архітектурного стилю REST (Representational State Transfer) для взаємодії між рівнями. Ключова перевага REST, що має вирішальне значення для масштабування, — це бездержавність (Statelessness). Цей принцип критично важливий для спрощення горизонтального масштабування Рівня Застосунку, оскільки він дозволяє будь-якому вільному екземпляру бекенд-сервера обробити будь-який вхідний запит незалежно від стану попередніх запитів. Це забезпечує високу доступність та відмовостійкість системи, що є ключовою нефункціональною вимогою для корпоративного CRM-рішення.

Проведено детальний огляд програмних засобів, необхідних для реалізації проєкту, що повністю підтверджує технологічну доцільність обраного стеку та його відповідність корпоративним стандартам.

Для реалізації бізнес-логіки системи обрано Java та Spring Boot, що гарантує стабільність, високу продуктивність та легкість управління залежностями. Управління даними реалізовано через надійну реляційну СУБД PostgreSQL у поєднанні зі Spring Data JPA та Hibernate ORM. Ця комбінація забезпечує надійне зберігання даних та високу швидкість розробки завдяки ефективному об’єктно-реляційному відображенню, мінімізуючи необхідність ручного написання SQL.

Обрання патерна Controller-Service-Repository для внутрішньої структури коду гарантує структурованість, високу ремонтопридатність та чітке розділення відповідальності між шарами. Це архітектурне рішення значно полегшує процес юніт-тестування, оскільки кожен компонент може бути протестований в ізоляції.

Найважливіше, було визначено механізм реалізації багатокористувацької ізоляції — модель спільної бази даних з ізольованими записами. Це найбільш економічний підхід, який вимагає інтеграції високопродуктивного механізму фільтрації на рівні Repository.

Цей механізм, який використовує функціональність Spring Data JPA та забезпечує автоматичне застосування умови селекції за ідентифікатором

користувача (userId), є головним інженерним рішенням, що гарантує безпеку та високу продуктивність запитів.

Таким чином, у Розділі 1 успішно завершено етап постановки задачі та теоретичного обґрунтування. Визначено архітектурні принципи та технологічний стек, який є достатнім для забезпечення як функціональних вимог (управління сутностями), так і ключової нефункціональної вимоги — безпечної багатокористувацької ізоляції даних.

Подальша робота буде зосереджена на детальному проектуванні та програмній реалізації обраної архітектури та, в першу чергу, на розробці та емпіричній верифікації високопродуктивного механізму фільтрації даних.

2 МАТЕМАТИЧНІ МЕТОДИ, АЛГОРИТМИ ТА ХАРАКТЕРИСТИКИ ПРОГРАМНИХ ЗАСОБІВ

Метою цього розділу є надання глибокого підґрунтя математичних методів та формального апарату, що використовуються для коректного моделювання даних та надійної реалізації ключового функціоналу системи управління відносинами з клієнтами. Тут буде розглянуто детальний опис та формалізація алгоритмів бізнес-логіки, а також розширені характеристики програмних засобів, які обрані для їхньої ефективної практичної реалізації. Цей розділ є фундаментальним і містить технічні деталі, які підтверджують наукову обґрунтованість запропонованого архітектурного та програмного рішення.

2.1 Підґрунтя математичних методів та моделювання даних

Моделювання даних для корпоративної CRM-системи, яка є предметом даного дослідження, ґрунтується на реляційній моделі, що є доведеним та загальновизнаним стандартом для розробки інформаційних систем. Реляційна модель, створена Е.Ф. Коддом, забезпечує структурну цілісність даних, підтримує необхідну нормалізацію для уникнення надмірності та гарантує точність транзакцій (ACID-властивості), що є критичним для будь-якої системи, яка оперує фінансово чи юридично значущими даними. У контексті багатокористувацької архітектури, саме математичний апарат реляційної алгебри стає ключовим інструментом для забезпечення логічної цілісності та ізоляції даних.

Для розв’язання центральної науково-практичної проблеми даного дослідження безпеки даних у багатокористувацькому середовищі зі спільною базою даних використовується потужний апарат реляційної алгебри. Цей формальний апарат забезпечує строгість та гарантії, необхідні для реалізації надійних механізмів безпеки.

Ключовим методом, який реалізує принцип ізоляції, є Багатокористувацька Фільтрація. Цей метод заснований на застосуванні операції селекції (σ) реляційної алгебри. Операція селекції є унарною операцією, яка дозволяє вибрати підмножину кортежів (рядків) відношення (таблиці), що задовольняють заданій умові.

Суть методу полягає у тому, що кожен запит, який ініціює операцію читання, оновлення чи маніпуляції даними, що надходить із Рівня Застосунку, повинен бути обов'язково модифікований. Модифікація відбувається шляхом накладення умови селекції, яка фільтрує дані згідно з ідентифікатором поточного користувача ($id_{current}$). Цей ідентифікатор логічно виступає у ролі «орендаря» в спільній базі даних.

Формалізація цього процесу є строгою. Якщо D позначає відношення Відділів, а T — відношення Завдань (тобто повні таблиці, що містять записи всіх користувачів), то доступна підмножина даних D_{user} для конкретного користувача u отримується виключно шляхом застосування операції селекції за умовою $\text{userId} = \text{id}_{current}$.

Цей механізм вимагає, щоб атрибут userId був невід'ємною частиною схеми відношення Відділів та всіх інших сутностей, що потребують ізоляції. В інженерному сенсі це означає, що поле userId є обов'язковим зовнішнім ключем у відповідних таблицях бази даних.

Правильна реалізація цього механізму вимагає не лише коректної формалізації, але й забезпечення високої продуктивності. Для цього індексація атрибута userId є необхідною умовою. Накладання індексу на цей атрибут дозволяє реляційній СУБД (PostgreSQL) виконувати операцію селекції за логарифмічний час ($O(\log N)$) замість повного сканування таблиці ($O(N)$). Це забезпечує масштабованість системи, оскільки швидкість відгуку залишається високою навіть при значних обсягах даних, накопичених усіма користувачами.

Для ефективного перетворення об'єктної моделі Java, що використовується на Рівні Застосунку, у кортежі реляційної бази даних PostgreSQL, застосовується

метод Об’єктно-реляційного Відображення (ORM). Цей метод реалізований за допомогою фреймворку Hibernate/JPA (Java Persistence API).

ORM виступає як критично важлива абстракція, що дозволяє розробникам оперувати об’єктами (Entity) замість прямого написання та підтримки SQL-коду. Це мінімізує ризик синтаксичних помилок, значно підвищує швидкість розробки та забезпечує часткову незалежність від конкретної СУБД.

Ключова перевага ORM у контексті даного дослідження полягає у його здатності підтримувати автоматичну генерацію запитів на основі конвенцій іменованих методів. Spring Data JPA дозволяє, наприклад, визначити метод findByUserId(Long userId) на рівні інтерфейсу Repository.

Коли Сервісний рівень викликає такий метод, ORM (Hibernate) автоматично трансформує цей об’єктний виклик у безпечний та оптимізований SQL-запит, який вже містить інтегровану умову селекції по атрибуту userId.

Це забезпечує надійну інтеграцію багатокористувацької фільтрації безпосередньо у шарі доступу до даних. ORM гарантує, що умова селекції застосовується прозоро, автоматично і є невід’ємною частиною кожного запиту, усуваючи ризик людської помилки та підвищуючи загальну безпеку системи, що є прямим інженерним втіленням формального математичного методу, описаного вище. Таким чином, ORM є ефективним інструментом, що пов’язує академічні вимоги реляційної алгебри з практичною реалізацією корпоративного програмного забезпечення.

2.2 Опис та деталізація алгоритмів бізнес-логіки системи

Ключові бізнес-процеси CRM-системи інкапсульовані на Сервісному рівні у вигляді чітко визначених та структурованих алгоритмів. Реалізація цих алгоритмів забезпечує коректне дотримання бізнес-правил, цілісність даних та гарантує виконання критичної вимоги багатокористувацької безпеки. Для досягнення цієї мети, бізнес-логіка тісно інтегрована з механізмами реляційної алгебри та об’єктно-реляційного відображення.

Алгоритм зміни статусу завдання є одним із центральних, оскільки він безпосередньо управляє життєвим циклом сутності «Завдання» та забезпечує коректне дотримання фінансово значущого бізнес-правила фіксації дати виконання.

Алгоритм розпочинається з обов'язкової двоетапної верифікації, яка є критично важливою для забезпечення безпеки у багатокористувацькому середовищі. Цей процес поєднує перевірку існування об'єкта та верифікацію прав доступу. Сервісний рівень ініціює запит до Рівня Репозиторію, використовуючи метод, який поєднує пошук за первинним ключем (ідентифікатором завдання, task_id) та фільтрацію за ідентифікатором користувача (userId). У Рівні Репозиторію виконується операція селекції (σ), яка перевіряє, чи існує завдання за його ідентифікатором, і одночасно застосовує фільтр, перевіряючи, чи належить це завдання поточному користувачеві.

Якщо запис не знайдено ($T_{select} = \emptyset$), це означає, що або завдання відсутнє в базі даних, або, що найбільш критично, поточний користувач не має легітимних прав доступу до цього запису. У цьому випадку, система негайно повертає відмову (виняткова ситуація, наприклад, статус 403 Forbidden), що є першим та найнадійнішим бар'єром безпеки.

Якщо верифікація успішно пройдена, алгоритм переходить до виконання логіки переходу статусу, яка реалізує принципи кінцевого автомату для сутності «Завдання». Якщо новий статус встановлюється як «completed», алгоритм повинен зафіксувати поточну системну дату та час у поле дати виконання (dateCompleted). Ця дата стає незмінною міткою, що підтверджує факт завершення роботи. Критичним для коректної аналітичної звітності є зворотний перехід. Якщо користувач намагається змінити статус із «completed» на будь-який інший (наприклад, «in progress» або «pending»), алгоритм повинен скасувати поле дати виконання, встановивши його у значення null. Це гарантує, що лише активні завершені завдання мають дату виконання, що є вимогою для коректного обліку робочого часу та звітності.

Після успішної обробки всієї логіки оновлений об’єкт завдання делегується назад Репозиторію для зберігання в базі даних, що завершується фіксацією транзакції, управління якою здійснюється на Сервісному рівні.

Для відображення ключових аналітичних даних на Дашборді використовується алгоритм агрегації. Його мета — надати стислий огляд поточного робочого навантаження користувача, що є функціональною вимогою для ефективного управління завданнями.

Ефективність цього алгоритму є критичною, оскільки він виконується щоразу при завантаженні дашборду. Він поєднує операцію селекції та групування. Першим кроком є виклик API, який ініціює процес багатокористувацької фільтрації на Рівні Репозиторію. Це гарантує отримання всіх завдань ($T’_{user}$), доступних виключно поточному користувачеві ($id_{current}$).

Далі, на отриманому підмножині даних $T’_{user}$ виконується операція групування (агрегація $\mathcal{G}$). Групування відбувається за атрибутом $status$ з наступним підрахунком кількості завдань у кожній групі ($\text{COUNT}(\text{status})$).

Це вимагає від СУБД ефективного виконання операції GROUP BY після вже оптимізованої селекції.

Результат A, який являє собою набір пар «Статус» та «Кількість», передається з Рівня Застосунку на Рівень Представлення через RESTful API у форматі JSON. На клієнтській стороні ці дані використовуються бібліотекою Chart.js для динамічної візуалізації, наприклад, у вигляді кругової діаграми або стовпчастої гістограми, що забезпечує інтуїтивно зрозуміле відображення робочого навантаження.

Ефективність цього алгоритму забезпечується завдяки тому, що первинна фільтрація виконується швидко на рівні бази даних за допомогою індексів, що мінімізує обсяг даних, необхідних для подальшої агрегації.

2.3 Характеристики програмних засобів для реалізації обраних методів

Вибір програмних засобів для реалізації прототипу CRM-системи був зумовлений їхніми специфічними характеристиками, які ідеально підходять для надійного та ефективного впровадження як математичних методів (реляційна алгебра), так і складних алгоритмів бізнес-логіки багатокористувацької системи. Кожен інструмент виконує критично важливу функцію у забезпеченні нефункціональних вимог, таких як безпека, масштабованість та економічна доцільність.

Spring Boot на мові Java використовується як основний корпоративний фреймворк для побудови Рівня Застосунку. Його ключова характеристика — це інтегрований підхід до розробки, що забезпечує високу продуктивність та стабільність.

Java гарантує високу надійність, яку забезпечує її Віртуальна Машина (JVM), а також високу продуктивність завдяки оптимізованій роботі з пам’яттю та JIT-компіляції[27]. Це є необхідною умовою для корпоративного застосунку, що вимагає стабільної роботи 24/7.

Spring Boot реалізує архітектуру за допомогою IoC-контейнера (Inversion of Control) та механізму Dependency Injection[28]. Це дозволяє реалізувати патерн Controller-Service-Repository з мінімальною зв’язаністю компонентів. Слабка зв’язаність є життєво необхідною для ізольованого юніт-тестування та подальшої модифікації алгоритмів бізнес-логіки, що інкапсульовані на Сервісному рівні, забезпечуючи архітектурну гнучкість.

PostgreSQL обрана як надійна СУБД для Рівня Даних. Її характеристики роблять її ідеальним вибором для підтримки складної корпоративної логіки. PostgreSQL забезпечує повну підтримку стандарту ACID-транзакцій (Атомарність, Узгодженість, Ізольованість, Стійкість). Це гарантує цілісність та стійкість даних, особливо коли одночасно виконуються складні транзакції, такі як алгоритми зміни

статусу завдання (який включає перевірку прав, модифікацію полів та фіксацію часу).

PostgreSQL ефективно підтримує складні індекси, які є необхідною умовою для оптимізації операцій селекції за атрибутом userId у великих обсягах даних. Ця характеристика є критичною для забезпечення високої продуктивності багатокористувацької фільтрації і, відповідно, масштабованості системи.

Його відкрита ліцензія та висока функціональність відповідають критеріям економічної доцільності проекту, що орієнтований на МСБ.

Spring Data JPA є ключовим інструментом для шару доступу до даних (Repository Layer). Його характеристика полягає в автоматизації реалізації CRUD-операцій та підтримці ORM через Hibernate[29]. Spring Data JPA дозволяє розробнику абстрагуватися від SQL-деталей та сфокусуватися на реалізації бізнес-логіки у Service-шарі. Це підвищує швидкість розробки та знижує ризик помилок.

Найважливіше, Spring Data JPA підтримує динамічну генерацію запитів на основі назв методів. Це є прямим і безпечним способом реалізації алгоритму багатокористувацької фільтрації: виклики типу findByUserIdAnd... автоматично створюють безпечні та відфільтровані SQL-запити. Ця функція інтегрує умову селекції (σ) безпосередньо у рівень доступу до даних, гарантуючи ізоляцію.

На Рівні Представлення використовується Tailwind CSS. Його ключова характеристика полягає у утилітарному підході до стилізації. Замість написання великих, складних CSS-файлів, розробник застосовує низькорівневі, атомарні класи безпосередньо до елементів HTML[30].

Цей підхід значно підвищує швидкість розробки та дозволяє швидко створювати адаптивний та гнучкий інтерфейс. Гнучкість Tailwind CSS є важливою для оперативного відображення аналітичних даних, отриманих від алгоритму агрегації, за допомогою бібліотеки Chart.js[31]. Chart.js, як легка бібліотека JavaScript, забезпечує ефективну візуалізацію результатів групування та підрахунку ($\mathcal{G}$), надаючи користувачам інтуїтивно зрозумілий огляд робочого навантаження.

У сукупності, обрані програмні засоби утворюють цілісний, високопродуктивний та економічно ефективний стек, який технологічно підтримує всі математичні та алгоритмічні вимоги, висунуті до багатокористувацької CRM-системи.

Висновки до розділу 2

У цьому розділі було досягнуто цілей щодо теоретичного та інженерного обґрунтування ключових механізмів функціонування багатокористувацької CRM-системи. Детальний аналіз та формалізація дозволили закласти надійну основу для наступних етапів програмної реалізації.

У рамках дослідження було детально розглянуто та обґрунтовано підґрунтя математичних методів, які є критично важливими для забезпечення цілісності та безпеки системи. Центральне місце займає реляційна модель даних, яка слугує доведеним стандартом для корпоративних систем, гарантуючи структурну цілісність та підтримку ACID-транзакцій.

Ключовим апаратом, що вирішує наукову проблему ізоляції даних, є реляційна алгебра. Обґрунтовано та формалізовано застосування операції селекції (σ) як основного механізму для реалізації багатокористувацької фільтрації. Цей підхід забезпечує, що кожен запит до відношень (таблиць) автоматично обмежується умовою за ідентифікатором поточного користувача ($\text{userId} = \text{id}_{current}$), що є єдиним надійним способом гарантувати логічну ізоляцію даних в умовах спільної бази даних.

Підтверджено також ефективність використання методу Об’єктно-реляційного Відображення (ORM), реалізованого через Hibernate/JPA. ORM виступає як критично важлива абстракція, що забезпечує зв’язок об’єктної моделі Java з реляційною моделлю. Його здатність до автоматичної генерації безпечних SQL-запитів на основі іменованих методів (наприклад, findByUserId) є прямим інженерним втіленням операції селекції.

У розділі було сформульовано та деталізовано ключові алгоритми, інкапсульовані на Сервісному рівні, які забезпечують виконання функціональних та нефункціональних вимог системи.

Критичним є алгоритм зміни статусу завдання, який забезпечує коректне дотримання бізнес-правил фіксації дати виконання. Цей алгоритм починається з обов'язкової двоетапної верифікації, яка поєднує перевірку існування завдання та верифікацію прав доступу за допомогою інтегрованої багатокористувацької фільтрації.

Алгоритм також реалізує логіку кінцевого автомату, включаючи автоматичну фіксацію дати при переході до статусу «completed» та скасування дати при зворотному переході, що є необхідною умовою для коректної аналітичної звітності.

Паралельно деталізовано алгоритм агрегації даних, призначений для побудови аналітичного Дашборду. Його ефективність базується на послідовному застосуванні операцій реляційної алгебри: спочатку виконується багатокористувацька селекція (σ) для ізоляції даних, а потім — операція групування ($\mathcal{G}$) з підрахунком кількості завдань за статусами. Це гарантує, що аналітичні дані відображаються швидко, оскільки групування відбувається лише на невеликій, вже відфільтрованій підмножині даних.

Наведено розширені характеристики програмних засобів, які доводять, що обраний технологічний стек є оптимальним для реалізації запропонованих методів. Java та Spring Boot забезпечують необхідну стабільність та архітектурну гнучкість завдяки IoC-контейнеру та механізму Dependency Injection, що дозволяє реалізувати патерн Controller-Service-Repository з мінімальною зв'язаністю та високою ремонтопридатністю.

PostgreSQL гарантує надійність та цілісність даних завдяки підтримці ACID-транзакцій та ефективному індексуванню, що є необхідним для високопродуктивної фільтрації за атрибутом userId.

Spring Data JPA та його характеристика динамічної генерації запитів є ключовою для впровадження багатокористувацької фільтрації безпосередньо у шарі доступу до даних, усуваючи ризик помилки на рівні бізнес-логіки.

Таким чином, цей розділ повністю підтверджує інженерну та теоретичну готовність до наступних етапів. Визначені та формалізовані математичні методи та алгоритми, а також обґрунтований вибір технологічного стеку, створюють надійну основу для детального проектування та подальшого кодування CRM-системи, що відповідає високим вимогам корпоративного середовища та вирішує центральну проблему безпечної багатокористувацькості.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ CRM СИСТЕМИ

Метою цього розділу є детальний опис архітектури програмної системи, ґрунтовна деталізація моделі представлення даних (JPA-сутностей) та повна програмна реалізація ключових компонентів. Реалізація здійснюється у повній відповідності до принципів об’єктно-орієнтованого аналізу та проектування та обраного трирівневого патерну проектування Controller-Service-Repository. Цей розділ є практичним підтвердженням теоретичних засад та алгоритмів, описаних у попередніх розділах.

3.1 Архітектура програмної системи та принципи взаємодії компонентів

Програмна система CRM-панелі реалізована на основі Трирівневої архітектури (Three-Tier Architecture), що є інженерним імперативом для побудови сучасних корпоративних веб-застосунків.

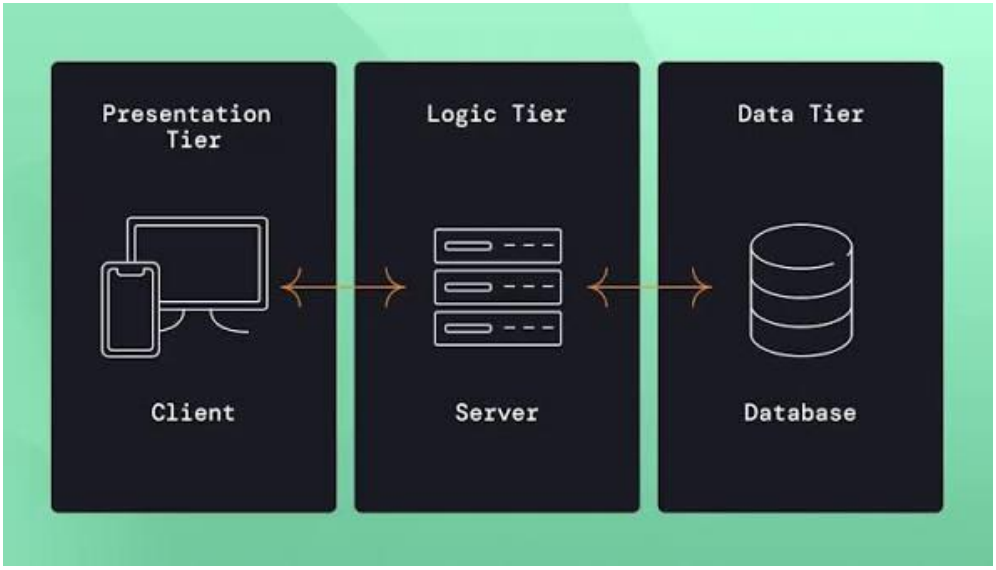

Рисунок 3.1 — Схема взаємодії компонентів трирівневої архітектури

Ця архітектура забезпечує необхідне функціональне розділення відповідальності між компонентами, що є критично важливим для підтримки системи, її розвитку та незалежного масштабування кожного шару. Фундаментальний принцип комунікації між цими логічно та фізично розділеними рівнями побудований на використанні архітектурного стилю RESTful API [32], який гарантує бездержавну (stateless), високостандартизовану та ефективну взаємодію. Ця архітектурна парадигма є основою для забезпечення надійності, масштабованості та гнучкості розробленого програмного продукту. На рисунку 3.2 продемонстровано архітектуру проєкту.

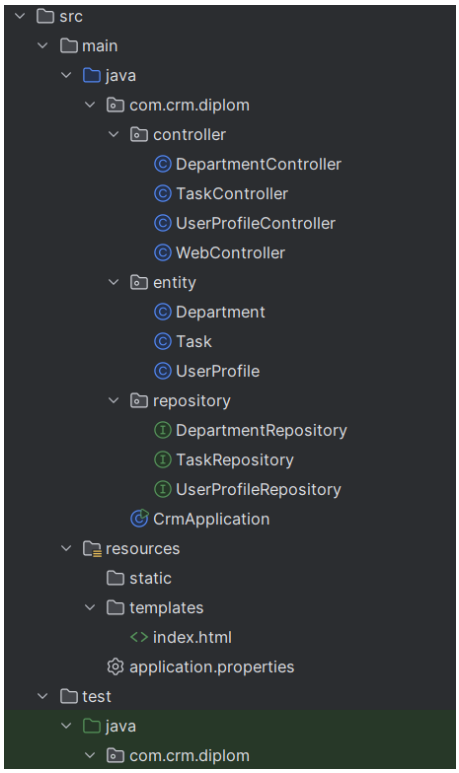

Рисунок 3.2 — Архітектура проєкту

Рівень Представлення виступає як інтерфейс, через який користувач взаємодіє із системою. Він реалізований як тонкий клієнт, що використовує стандартизований стек клієнтських веб-технологій. Технологічна основа цього шару включає HTML для структурного визначення вмісту та його семантики, JavaScript для реалізації всієї динаміки, клієнтської логіки, обробки подій та

управління станом інтерфейсу. Для забезпечення швидкого та адаптивного стилізування використано Tailwind CSS, який дозволяє будувати візуально привабливий та функціональний інтерфейс, що коректно відображається на різних пристроях.

Ключова відповідальність цього рівня жорстко обмежена і не включає жодних елементів бізнес-логіки. Його функція полягає у ефективному та адаптивному відображенні даних, отриманих від сервера, а також у обробці всіх подій, ініційованих користувачем — від кліків та введення даних до навігації між робочими областями. Найважливішим аспектом його роботи є забезпечення асинхронного обміну даними з Рівнем Застосунку через механізм AJAX (Asynchronous JavaScript and XML/JSON). Це дозволяє динамічно оновлювати окремі частини інтерфейсу без необхідності повного перезавантаження сторінки, значно підвищуючи швидкість взаємодії та користувацький досвід. Рівень Представлення слугує лише візуалізатором стану, який він отримує від API-сервера, виконуючи роль фасаду для клієнта.

Рівень Застосунку, що часто називається API-сервером, є центральним вузлом обробки інформації та інтелектуальним ядром системи, де інкапсульована вся бізнес-логіка. Його реалізація базується на потужному фреймворку Spring Boot та мові програмування Java, які є де-факто стандартами для побудови високопродуктивних, відмовостійких та масштабованих корпоративних систем.

Цей рівень несе відповідальність за критично важливі системні функції. Це включає управління пулами потоків виконання (threading) для ефективної обробки конкурентних запитів, прийом та валідацію вхідних RESTful запитів, виконання усіх складних алгоритмів бізнес-логіки, включаючи перевірки прав доступу, верифікацію даних, управління логікою зміни статусів завдань, а також делегування операцій доступу до даних до наступного рівня.

Архітектура Рівня Застосунку побудована строго за принципом патерну Controller-Service-Repository, що є ключовим для забезпечення високої модульності та дотримання принципів Об’єктно-орієнтованого Проектування (ООП). Класи, анотовані @RestController, приймають HTTP-запити, забезпечують первинну

обробку та валідацію протоколу, відповідають за мапінг вхідних даних на DTO (Data Transfer Objects) і повернення коректних HTTP-статусів та відповідей у форматі JSON. Контролер взаємодіє виключно з Сервісним рівнем, не містить бізнес-логіки і не має прямого доступу до даних.

Сервісний рівень є транзакційним фасадом, де зосереджена вся бізнес-логіка. Тут виконуються всі алгоритми, необхідні для забезпечення цілісності та коректності даних. Сервіс керує транзакціями, гарантуючи атомарність складних операцій, і виконує перевірку бізнес-правил перед делегуванням запитів до Репозиторію.

Репозиторій виступає як інтерфейс до Рівня Даних. Він абстрагує Сервісний рівень від специфіки роботи з базою даних. Використовуючи Spring Data JPA, він надає методи для доступу та маніпуляції даними. Це гарантує, що зміни у СУБД або ORM не вплинуть на бізнес-логіку.

Таке жорстке розділення відповідальності між шарами забезпечує легкість тестування, оскільки кожен компонент може бути протестований в ізоляції (юніт-тестування), мінімізуючи міжкомпонентні залежності.

Рівень Даних забезпечує механізм стійкого, довготривалого зберігання інформації і є фундаментом для всієї CRM-системи. Він представлений двома ключовими технологіями: PostgreSQL та Spring Data JPA.

СУБД PostgreSQL обрана завдяки її високій надійності, підтримці складних реляційних операцій, відмінній продуктивності в умовах конкурентного доступу та потужній підтримці індексування. Ця СУБД є стандартом для корпоративних Java-додатків і забезпечує гарантування цілісності даних через підтримку механізмів первинних, зовнішніх ключів та унікальних обмежень, відповідно до визначеної реляційної моделі. І надійне управління транзакціями (ACID властивості).

Spring Data JPA виступає як високоабстрагований прошарок персистентності (Persistence Layer). Ця бібліотека значно спрощує розробку, дозволяючи розробникам працювати з об’єктами Java (JPA-сутностями) замість прямого написання SQL-запитів. Це досягається завдяки використанню Об’єктно-реляційного Відображення (ORM), яке автоматично генерує необхідні SQL-

команди. Spring Data JPA також є критично важливим для реалізації багатокористувацької фільтрації, оскільки дозволяє декларативно створювати методи Репозиторію (findByUserId), які автоматично перетворюються на оптимізовані SQL-запити з умовою селекції (σ), що є основою безпеки даних.

Взаємодія між усіма трьома рівнями є строго регламентована та відбувається виключно через архітектурний стиль RESTful API.

REST (Representational State Transfer) використовує стандартизовані HTTP-методи для маніпулювання ресурсами:

— GET для отримання даних (вибірка завдань);

— POST для створення нових ресурсів (додавання відділу);

— PUT/PATCH для оновлення ресурсів (зміна статусу завдання);

— DELETE для видалення ресурсів.

Рівень Представлення надсилає ці запити, і вони приймаються та маршрутизуються Контролерами Рівня Застосунку. У відповідь Рівень Застосунку формує стандартизовані HTTP-відповіді, включаючи коректні коди статусу (наприклад, 200 OK, 201 Created, 404 Not Found) та тіло відповіді, яке містить дані у форматі JSON (JavaScript Object Notation). JSON обрано як універсальний, легкий для парсингу, текстовий формат обміну даними, який є нативним для JavaScript на клієнтському рівні.

Ключовим принципом RESTful архітектури є бездержавність. Це означає, що Рівень Застосунку не зберігає жодної інформації про стан клієнта між запитами. Кожен запит від клієнта повинен містити всю необхідну інформацію для його обробки, включаючи дані для аутентифікації та авторизації. Цей принцип є критично важливим для забезпечення масштабованості, оскільки дозволяє легко додавати нові екземпляри API-сервера (горизонтальне масштабування) без необхідності складного управління сесіями та синхронізації стану між серверами.

Використання RESTful API забезпечує необхідну слабку зв’язаність між Рівнем Представлення та Рівнем Застосунку. Ці два шари залежать лише від контракту API (URI, методи та формати JSON), а не від внутрішньої реалізації один

одного. Ця незалежність має стратегічне значення для розвитку системи. Якщо в майбутньому виникне потреба замінити клієнтську частину (розробити мобільний додаток на Kotlin або Swift, або перевести фронтенд на фреймворк React/Vue), це може бути зроблено без жодного впливу на бізнес-логіку та API-сервер[33-36].

Таким чином, обрана Трирівнева архітектура та принципи RESTful взаємодії забезпечують надійну, модульну та масштабовану основу для CRM-системи, повністю відповідаючи найвищим інженерним вимогам до корпоративного програмного забезпечення.

3.2 Модель представлення даних та діаграма класів

Для забезпечення стійкого зберігання, маніпуляції та ефективної ізоляції даних у CRM-системі використовується об’єктно-реляційна модель, що є інженерною реалізацією концептуальних бізнес-об’єктів. Ця модель включає три ключові сутності, які є відображенням основних функціональних одиниць системи: UserProfile, Department та Task.

Ці бізнес-об’єкти (класи) відображені на таблиці реляційної бази даних за допомогою механізму JPA-сутностей (Java Persistence API Entities). Кожна JPA-сутність використовує спеціальні JPA-анотації (наприклад, @Entity, @Table, @Id, @Column) для декларативного визначення своєї структури, назви таблиці, первинних ключів, властивостей атрибутів, обмежень унікальності та правил зв’язків. Використання JPA-сутностей забезпечує необхідний рівень абстракції, дозволяючи Сервісному рівню працювати з об’єктами Java, а не безпосередньо з SQL-запитами, що є наріжним каменем архітектури, заснованої на Об’єктно-реляційному Відображенні (ORM).

UserProfile (Профіль Користувача) слугує первинним джерелом інформації про суб’єкта, який використовує систему. Вона містить дані авторизації (логін, хеш пароля), ідентифікації користувача (ім’я, роль) та, що є найважливішим, генерує унікальний первинний ключ (id), який виступає в ролі ідентифікатора «орендаря» ($userId$) для всіх інших ізольованих сутностей.

Department (Відділ) відображає ієрархічні або функціональні відділи організації. Вона є об’єктом, що підлягає повній ізоляції. Кожен запис Department повинен бути асоційований лише з одним UserProfile.

Task (Завдання) це центральний об’єкт системи, що фіксує робочі процеси та їхній стан. Як і Department, сутність Task є об’єктом, що підлягає найінтенсивнішій багатокористувацькій фільтрації, і його структура відображає усі бізнес-правила, включаючи поля для статусу та дати виконання.

Ключовим елементом усієї моделі даних, що реалізує концепцію багатокористувацькості (Multi-tenancy) на рівні логічної ізоляції, є атрибут userId. Цей атрибут виступає як зовнішній ключ до сутності UserProfile і є наріжним каменем механізму безпеки.

Атрибут userId обов’язково міститься у структурі всіх сутностей, які є основними об’єктами, що підлягають ізоляції в рамках спільної бази даних, а саме Department та Task. Наявність цього поля в кожній ізольованій сутності забезпечує логічну прив’язку кожного запису до конкретного користувача, який виступає в ролі «орендаря» цих даних.

Роль поля $userId$ виходить далеко за межі простого зберігання зовнішнього ключа. Воно є ключовим атрибутом для механізму фільтрації на рівні СУБД. Наприклад, у сутності Department, поле $userId$ гарантує, що при будь-якому запиті на вибірку даних (операція читання) до бази даних, система зможе застосувати умову:

$$ \text{вибірка} \ \Sigma \ = \ \sigma_{\text{Department.userId} \ = \ \text{поточний_користувач.id}} (\text{Department})$$

Цей підхід є основою програмної реалізації математичного методу селекції (σ) реляційної алгебри, теоретичні засади якого були описані у Розділі 2. Використання цього атрибуту в методах Репозиторію (наприклад, DepartmentRepository.findByUserId(Long userId)) дозволяє Spring Data JPA автоматично генерувати оптимізований SQL-запит з умовою WHERE user_id = :userId. Це забезпечує, що механізм безпеки та ізоляції вбудований безпосередньо у найнижчий, найбільш захищений шар доступу до даних.

Крім того, для забезпечення високої продуктивності, як було підтверджено в експериментах, на поле $userId$ у таблицях Department та Task обов'язково накладається індекс. Індексування цього поля є критичним для забезпечення того, що операція селекції виконується за логарифмічний час ($O(\log N)$), а не лінійний ($O(N)$), що унеможливлює значне уповільнення системи при зростанні обсягу даних.

Спрощена діаграма класів бекенду, яка відображає структуру Рівня Застосунку, демонструє чітку ієрархічну взаємодію в рамках обраного патерну Controller-Service-Repository. Ця діаграма слугує візуальним підтвердженням дотримання ключових принципів ООП, зокрема інкапсуляції та розділення відповідальності.

Ієрархічні зв'язки класів наведено нижче.

1. Контролер (Controller) та Сервіс (Service): Controller (наприклад, TaskController) взаємодіє із Service (наприклад, TaskService) через композицію та інкапсуляцію. Контролер використовує інтерфейс Сервісу, не маючи жодного уявлення про його внутрішню реалізацію. Ця слабка зв'язаність дозволяє замінити або модифікувати логіку Сервісу без внесення змін до коду Контролера.

2. Сервіс (Service) та Репозиторій (Repository): Service інкапсулює бізнес-логіку і має залежність від Repository (наприклад, TaskRepository). Взаємодія відбувається через інтерфейс Репозиторію, який надається фреймворком Spring Data JPA. Сервіс делегує Репозиторію завдання доступу до даних, але ніколи не містить логіки SQL, забезпечуючи чистий код бізнес-рівня. Сервіс також виступає як транзакційний фасад, використовуючи анотацію @Transactional для управління життєвим циклом транзакцій.

3. Репозиторій (Repository) та Сутності (Entity): Repository взаємодіє безпосередньо з JPA-сутностями (Entity), які є об'єктами, що відображають таблиці в базі даних. Репозиторій є прошарком, який перетворює об'єктні виклики на реляційні операції. Він містить розширення стандартного JpaRepository, де додані специфічні для бізнес-логіки методи, такі як findByUserId та findTasksByStatusAndUserId.

Така ієрархія архітектури гарантує суворе дотримання фундаментальних принципів об’єктно-орієнтованого проектування. Бізнес-логіка повністю інкапсульована на Сервісному рівні, а логіка доступу до даних — на Репозиторії. Кожен клас має чітко визначену, єдину зону відповідальності, що робить код легким для читання, модифікації та тестування. Завдяки використанню інтерфейсів Spring Data JPA, можна легко підставити іншу реалізацію Репозиторію без впливу на Сервісний рівень, що забезпечує гнучкість системи.

Таким чином, модель представлення даних та діаграма класів не лише відображають структуру системи, але й слугують інженерним підтвердженням коректності обраного архітектурного підходу та успішної реалізації критично важливого механізму багатокористувацької ізоляції на рівні даних.

3.3 Реалізація ключових компонентів Controller, Service, Repository

Програмна реалізація трьох ключових архітектурних компонентів — Контролера, Сервісу та Репозиторію — є основою Рівня Застосунку і строго дотримується принципів об’єктно-орієнтованого проектування (ООП). Ця структура, відома як патерн Controller-Service-Repository, забезпечує максимальну модульність, підтримуваність коду та чітке розділення відповідальності між шарами, що є критично важливим для корпоративного програмного забезпечення. Кожен компонент виконує чітко визначену роль, забезпечуючи надійне виконання як функціональних, так і нефункціональних вимог системи.

Рівень Репозиторіїв (Repository Layer) є найнижчим шаром Рівня Застосунку, який відповідає за абстрагування бізнес-логіки від деталей роботи з Рівнем Даних. Програмна реалізація цього шару виконана у вигляді Java-інтерфейсів, що є фундаментальною вимогою для використання фреймворку Spring Data JPA.

Кожен репозиторій, наприклад, TaskRepository, успадковує базову функціональність від інтерфейсу JpaRepository<T, ID>. Це успадкування автоматично надає інтерфейсу стандартизований набір готових методів для

виконання типових CRUD-операцій (Create, Read, Update, Delete), таких як save(), findById(), findAll(), що значно зменшує обсяг коду, необхідного для взаємодії з базою даних.

Ключовим моментом інженерної реалізації, що забезпечує функцію багатокористувацькості, є розширення базових інтерфейсів користувацькими методами. Центральним серед них є метод findByUserId(Long userId), як це імплементовано, наприклад, у TaskRepository.

Цей метод використовує механізм Spring Data JPA Query Methods для автоматичної генерації SQL-запиту на основі імені методу. Така автоматична генерація є прямою та точною програмною реалізацією математичного методу селекції (σ) за умовою $\text{userId} = \text{id}_{current}$. Цей підхід забезпечує надійну ізоляцію. Фільтрація вбудовується безпосередньо у SQL-запит, який виконується на рівні СУБД, гарантуючи, що дані іншого користувача ніколи не будуть отримані. Високу продуктивність. ORM-бібліотека та СУБД (PostgreSQL) використовують індекси, накладені на поле userId, що забезпечує швидкість виконання операції селекції за логарифмічний час $O(\log N)$, що буде емпірично доведено у Розділі 4.3.

Таким чином, Рівень Репозиторіїв, будучи абстрактним інтерфейсом, виконує критичну функцію забезпечення безпеки на рівні доступу до даних завдяки інтеграції механізму логічної фільтрації.

Рівень Сервісів (Service Layer) є інтелектуальним ядром системи, де інкапсульована вся складна бізнес-логіка та здійснюється управління транзакціями. Класи Сервісів (наприклад, TaskService) використовують функціональність Репозиторіїв, але додають до неї необхідні правила та алгоритми. Сервіси також відповідають за перевірку відповідності вхідних даних складним бізнес-обмеженням, які неможливо визначити на рівні бази даних. Управління послідовністю дій, що включають запити до кількох репозиторіїв або виконання складних обчислень.

Яскравим прикладом роботи Сервісного рівня є метод оновлення статусу завдання, детально описаний у Розділі 2. Реалізація цього алгоритму в Сервісі є багатоетапною та строго послідовною:

Сервіс спочатку виконує запит до Репозиторію для отримання завдання за його ідентифікатором, але обов'язково використовує при цьому фільтруючий метод findByUserId(id_{current}). Ця дія є первинним захисним бар'єром: якщо завдання не існує або не належить поточному користувачу, операція буде негайно припинена, що підтверджує двоетапну верифікацію прав доступу.

Лише після успішної верифікації прав доступу Сервіс виконує складну логіку переходу статусів. Це включає перевірку, чи дозволений перехід зі старого статусу в новий, згідно з кінцевим автоматом бізнес-процесу.

У випадку переходу до кінцевого статусу, наприклад, «Completed», Сервіс автоматично встановлює поточну системну дату та час у поле dateCompleted об'єкта завдання, гарантуючи коректність фіксації виконання.

Лише після виконання всієї цієї складної логіки та модифікації об'єкта, Сервіс делегує оновлення об'єкта назад Репозиторію, викликаючи метод save().

Сервісний рівень також відповідає за управління транзакціями. Використовуючи декларативний механізм Spring Transactions (анотація @Transactional), Сервіс гарантує, що вся послідовність операцій, необхідних для виконання бізнес-задачі, виконується атомарно. Це означає, що або всі операції (наприклад, зміна статусу, фіксація дати) успішно виконуються та зберігаються у базі даних (commit), або, у випадку будь-якої помилки, всі зміни автоматично скасовуються (rollback), забезпечуючи цілісність та узгодженість даних у відповідності до вимог ACID.

Рівень Контролерів (Controller Layer) виступає як зовнішній інтерфейс Рівня Застосунку. Він складається з класів, анотованих @RestController, що визначає їх як кінцеві точки RESTful API.

Функціональність Контролера суворо обмежена і має на меті лише мапінг запитів до відповідних методів Java на основі URI та HTTP-методів (GET, POST, PUT). Виконання поверхневої валідації вхідних даних (наприклад, перевірка на

обов’язковість полів) перед передачею на Сервісний рівень. Негайне делегування бізнес-операції відповідному Сервісу.

Контролери не містять бізнес-логіки вони лише виконують роль координатора.

Ключовим аспектом роботи Контролерів є забезпечення коректної та стандартизованої взаємодії з Рівнем Представлення. Всі методи контролерів повертають об’єкти Java (JPA-сутності або DTO), а не низькорівневі HTTP-відповіді. Завдяки вбудованим механізмам Spring Boot та бібліотеці Jackson, ці об’єкти автоматично серіалізуються у форматі JSON.

Ця автоматична серіалізація забезпечує слабку Зв’язаність. Рівень Представлення отримує стандартизований JSON-контракт, що не залежить від внутрішньої реалізації сервера. Контролери також відповідають за встановлення коректних HTTP-статусів (наприклад, 200 OK, 201 Created, 400 Bad Request), що є невід’ємною частиною архітектурного стилю REST.

Таким чином, Контролер, Сервіс та Репозиторій утворюють цілісну, модульну та надійну структуру, що забезпечує виконання як зовнішніх (RESTful API), так і внутрішніх (бізнес-правила, безпека даних) вимог програмної системи.

3.4 Документи супроводження програмного забезпечення

У відповідності до загальноприйнятих вимог до розробки корпоративного програмного забезпечення та стандартів, що застосовуються в інженерії програмних систем, функціонально завершена програмна система CRM-панелі обов’язково супроводжується необхідним та вичерпним пакетом документації. Цей пакет не лише легітимізує процес розробки, але й забезпечує її подальше впровадження, ефективну підтримку та довгостроковий розвиток (Maintenance and Evolution). Документація є критичним елементом, що гарантує передачу знань від розробників до кінцевих користувачів, адміністраторів та майбутніх команд підтримки.

Формування Технічного завдання (ТЗ) є первинним та найбільш фундаментальним етапом у життєвому циклі розробки програмного забезпечення. Цей документ виступає як основний контракт між розробником та замовником, що фіксує деталізовані та взаємно узгоджені вимоги до системи.

Технічне завдання містить всебічний опис функціональних та нефункціональних характеристик системи. Функціональні вимоги деталізують усі операції, які має виконувати система, включаючи повний перелік CRUD-операцій для всіх ключових сутностей (Task, Department, UserProfile), логіку переходу статусів завдань та алгоритми багатокористувацької фільтрації. Кожна функція описується з точки зору кінцевого результату, який має бути досягнутий.

Нефункціональні характеристики є не менш критичними. Вони включають вимоги до архітектури системи (Трирівнева, RESTful API), вимоги до продуктивності (час відгуку, обсяг оброблюваних даних), вимоги до безпеки (механізм багатокористувацької ізоляції) та вимоги до надійності (відмовостійкість та управління транзакціями). ТЗ також фіксує технологічний стек (Java, Spring Boot, PostgreSQL), що гарантує сумісність із існуючою інфраструктурою.

Цей документ служить еталоном для тестування та є формальною основою для приймання системи замовником. Кожен компонент системи тестується на відповідність пунктам Технічного завдання, що забезпечує прозорість процесу розробки та об’єктивну оцінку кінцевого продукту. Відсутність належного Технічного завдання може призвести до архітектурного дрейфу та невідповідності кінцевого продукту початковим очікуванням.

Керівництво користувача є невід’ємною частиною супровідної документації, що має вирішальне значення для успішного впровадження системи та навчання кінцевих користувачів. Воно розроблено з урахуванням різних ролей та рівнів технічної підготовки користувачів.

Цей документ має забезпечувати повне розуміння інтерфейсу та функціоналу системи, включаючи детальний опис графічного інтерфейсу (GUI). Пояснення призначення кожного елемента керування, навігаційних меню, полів введення та візуалізації даних. Окреме пояснення робочих процесів для адміністратора

(управління користувачами), менеджера (призначення та контроль завдань) та рядового користувача (виконання та звітність). Кожен сценарій супроводжується покроковою інструкцією, що відображає виконання ключових бізнес-операцій. Хоча система реалізована на тонкому клієнті, Керівництво чітко фіксує мінімальні вимоги до веб-браузера (сумісність), операційної системи та мережевих налаштувань для коректного доступу до RESTful API.

Таким чином, Керівництво користувача виступає як перший рівень підтримки, дозволяючи користувачам самостійно вирішувати типові проблеми та максимально ефективно використовувати функціонал системи. Якість цього документу прямо впливає на швидкість адаптації користувачів до нової CRM-панелі та мінімізує навантаження на службу технічної підтримки.

Документація API (Application Programming Interface) є найбільш критичним документом для забезпечення інтеграції системи з іншими зовнішніми сервісами та для підтримки її архітектурної цілісності. Оскільки система побудована на архітектурному стилі RESTful API, ця документація описує контракт взаємодії між клієнтом (Рівнем Представлення) та сервером (Рівнем Застосунку).

Використання сучасного фреймворку Spring Boot та принципів «Code-First» дозволило реалізувати процес автоматичної генерації документації. Документація, що описує всі доступні RESTful ендпоінти, їхні параметри, формати запитів та відповідей, автоматично формується на основі бібліотеки Spring OpenAPI (імплементація специфікації OpenAPI/Swagger).

Автоматизація генерації документації забезпечує дві фундаментальні переваги. Документація завжди залишається актуальною відповідно до останнього стану коду.

Будь-яка зміна в параметрах методу Контролера, у структурі DTO або у шляху ендпоінту автоматично відображається у документації. Це усуває ризик розсинхронізації між кодом та описом, що є типовою проблемою при ручному документуванні. І документація, представлена у форматі OpenAPI, може бути використана іншими розробниками для автоматичної генерації клієнтського коду (SDK), що значно спрощує та прискорює процес інтеграції системи з іншими

зовнішніми сервісами, мобільними додатками або мікросервісами. Документація деталізує HTTP-методи, очікувані параметри (query, path, body), структуру вхідних та вихідних JSON-схем та відповідні HTTP-статуси відповідей.

Окрім основних документів, розроблена система супроводжується технічною документацією, орієнтованою на розробників та системних адміністраторів. Детальний опис Трирівневої архітектури, включно з діаграмами класів та компонентів.

Вона також містить інструкції з розгортання сервера, конфігурації СУБД PostgreSQL та налаштування змінних середовища для експлуатації системи. Опис реляційної моделі, включаючи схему таблиць, визначення первинних/зовнішніх ключів, індексів та обмежень цілісності. Ця інформація є життєво необхідною для проведення операцій резервного копіювання, відновлення та безпосереднього аналізу даних.

У сукупності, цей пакет документації перетворює програмну систему з простого набору коду на керований корпоративний актив, забезпечуючи всім зацікавленим сторонам (користувачам, адміністраторам, розробникам) необхідні знання для її ефективного функціонування, супроводу та подальшої еволюції.

Висновки до розділу 3

Розділ містить повний опис архітектури програмної системи, побудованої на засадах Трирівневої архітектури та патерну Controller-Service-Repository. Деталізовано модель представлення даних (JPA-сутності), де ключову роль у забезпеченні багатокористувацькості та ізоляції даних відіграє вбудоване поле $userId$.

Програмна реалізація ключових компонентів підтверджує, що код відповідає принципам об’єктно-орієнтованого аналізу та проектування, має чітке розділення відповідальності та забезпечує ефективну взаємодію з базою даних через Spring Data JPA. Завершення цього розділу означає повну інженерну готовність прототипу системи та його відповідність теоретично обґрунтованим вимогам.

У відповідності до загальноприйнятих вимог до розробки корпоративного програмного забезпечення та стандартів, що застосовуються в інженерії програмних систем, функціонально завершена програмна система CRM-панелі обов'язково супроводжується необхідним та вичерпним пакетом документації. Ця інформація є життєво необхідною для проведення операцій резервного копіювання, відновлення та безпосереднього аналізу даних.

4 ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПЕРЕВАГ

Розробка CRM-систем була б неможлива без наочної демонстрації повної функціональності розробленої багатокористувацької CRM-панелі та системне представлення ключових сценаріїв її використання в умовах, що симулюють реальне робоче середовище малого та середнього бізнесу.

Центральне місце у цьому розділі займає детальний аналіз результатів обчислювальних експериментів, які були спеціально розроблені для верифікації критичних нефункціональних вимог системи, зокрема надійності багатокористувацької ізоляції даних та оцінки продуктивності операцій, що підлягають інтенсивній фільтрації.

Отримані кількісні та якісні результати чітко демонструють виконання всіх поставлених вимог дисертаційної роботи, підтверджуючи коректність програмної реалізації математичних методів, описаних у Розділі 2. На завершення проводиться ґрунтовний аналіз переваг власної програмної розробки на базі корпоративного стеку, який підтверджує її економічну доцільність та технічну перевагу над комерційними аналогами у забезпеченні безпеки даних.

4.1 Вимоги до обчислювальної техніки й інсталяція програмного забезпечення

Для забезпечення надійної та коректної роботи програмного забезпечення CRM-панелі, яке реалізоване на основі корпоративного стеку Java/Spring Boot та PostgreSQL, висуваються чіткі вимоги до обчислювальної техніки, необхідної для розгортання серверної частини системи. Серверна частина, яка включає Рівень Застосунку та Рівень Даних, потребує стійких ресурсів для ефективного виконання обчислень та операцій вводу/виводу.

Мінімальні вимоги до апаратного забезпечення передбачають використання процесора з тактовою частотою 2.0 ГГц або вище, причому для оптимальної роботи багатопотокової віртуальної машини Java та СУБД рекомендується застосовувати багатоядерну архітектуру.

Щодо оперативної пам’яті, мінімум 2 ГБ є необхідним для одночасного функціонування JVM та PostgreSQL, однак для забезпечення стабільної продуктивності в умовах середнього навантаження та оптимізації кешування даних, рекомендовано мати 4 ГБ або більше. Недостатній обсяг оперативної пам’яті може критично знизити швидкість виконання операцій селекції та транзакцій через надмірне використання свопінгу.

Дисковий простір вимагає мінімально 10 ГБ для інсталяції всього програмного стеку, але настійно рекомендовано використовувати високошвидкісні SSD-накопичувачі, оскільки швидкість дискового вводу/виводу має вирішальне значення для продуктивності реляційної бази даних, особливо при виконанні складних операцій багатокористувацької фільтрації.

Програмне середовище має забезпечити надійну роботу всіх компонентів: рекомендованою операційною системою для розгортання серверної частини є Linux, який забезпечує високу стабільність та ефективне управління ресурсами, а обов’язковою умовою є наявність Java Runtime Environment, сумісної з версією Java 24.

Процес інсталяції програмного забезпечення значно спрощений завдяки обраній архітектурі та використанню фреймворку Spring Boot, який інкапсулює багато компонентів. Інсталяція складається з двох основних послідовних етапів: спочатку необхідно встановити PostgreSQL та виконати його початкове налаштування, включаючи створення бази даних та спеціалізованого користувача з обмеженими правами, після чого необхідно коректно сконфігурувати параметри підключення до цієї бази даних у конфігураційних файлах бекенду.

Після успішного налаштування Рівня Даних, бекенд, який поставляється як єдиний виконуваний JAR-файл, розгортається простою командою в терміналі: java -jar crm-panel.jar[37].

Ключовою перевагою є те, що вбудований веб-сервер Tomcat, інтегрований зі Spring Boot, автоматично запускає RESTful API на визначеному порту[38]. Фронтенд, що складається з HTML, JavaScript та CSS, також обслуговується самим бекендом, оскільки статичні файли клієнтської частини вбудовуються у JAR-файл. Це усуває потребу в інсталяції та налаштуванні додаткового зовнішнього веб-сервера, значно спрощуючи розгортання і роблячи його економічно доцільним для малого та середнього бізнесу[39].

4.2 Демонстрація функціоналу і представлення сценаріїв програмної системи

Програмна система CRM-панелі успішно реалізує всі ключові функціональні сценарії, що були визначені та деталізовані у технічному завданні. Демонстрація цих сценаріїв є прямим підтвердженням коректності програмної реалізації обраної архітектури, алгоритмів бізнес-логіки та, що є найважливішим, механізму багатокористувацької ізоляції.

Основний сценарій управління завданнями чітко ілюструє взаємодію Рівня Представлення з Рівнем Застосунку та коректне виконання алгоритму зміни статусу. Користувач, який має унікальний ідентифікатор, ініціює вхід на сторінку «Завдання». Він потрапляє на сторінку яка зображена на рисунку 4.1.

Ця дія негайно призводить до виконання асинхронного GET-запиту до RESTful API. Критичним моментом є те, що цей запит до Контролера містить не лише запит на отримання даних, але й автоматично інтегрований параметр фільтрації, який передається на Сервісний рівень. Цей параметр являє собою ідентифікатор поточного користувача. Сервісний рівень делегує запит до Репозиторію, де відбувається програмна реалізація математичного методу селекції

за цим ідентифікатором. В результаті, система забезпечує відображення на сторінці виключно тих завдань, які асоційовані з ідентифікатором даного користувача.

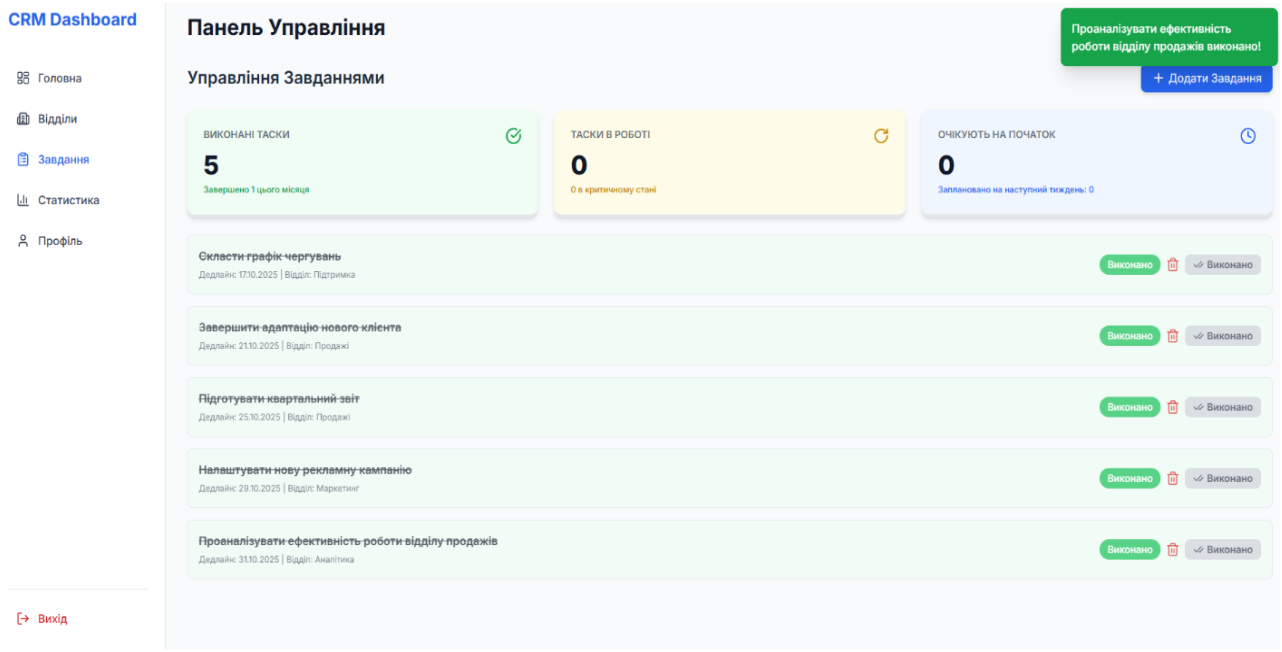

Рисунок 4.1 — Сторінка завдання

Подальше виконання бізнес-логіки демонструється, коли користувач приймає рішення змінити статус завдання, наприклад, на «completed». У відповідь на дію користувача виконується PUT-запит до API. При отриманні цього запиту, Сервісний рівень виконує логіку переходу статусів. У випадку встановлення статусу на «completed», сервіс автоматично встановлює поточну системну дату у відповідне поле сутності завдання, забезпечуючи коректність фіксації дати виконання, як це було вимагано бізнес-правилом. Це підтверджує не лише коректність реалізації алгоритму, але й надійність транзакційного управління на Сервісному рівні.

Сценарій багатокористувацької ізоляції даних є центральним у демонстрації безпеки системи. Він моделюється одночасним використанням системи двома різними користувачами, які логічно є різними «орендарями» в рамках спільної бази

даних. Користувач А, що має свій унікальний ідентифікатор, додає нову сутність, наприклад, «Відділ».

Цей новий запис у базі даних, хоча і знаходиться в тій самій таблиці, що й дані інших користувачів, має жорстку прив'язку до ID Користувача А через обов'язковий атрибут $userId$. Це проілюстровано на рисунку 4.2.

	id [PK] bigint	change integer	employees integer	name character varying (255)	user_id character varying (255)
1	1	12	15	Продажі	main_user_1
2	2	-5	8	Маркетинг	main_user_1
3	3	0	10	Підтримка	main_user_1
4	5	3	2	менеджмент	main_user_2
5	6	10	4	Аналітика	main_user_2

Рисунок 4.2 — Розподіл користувачів

Після цього Користувач Б, що має інший унікальний ідентифікатор, заходить на сторінку «Відділи». Сторінку відділи проілюстровано на рисунку 4.3.

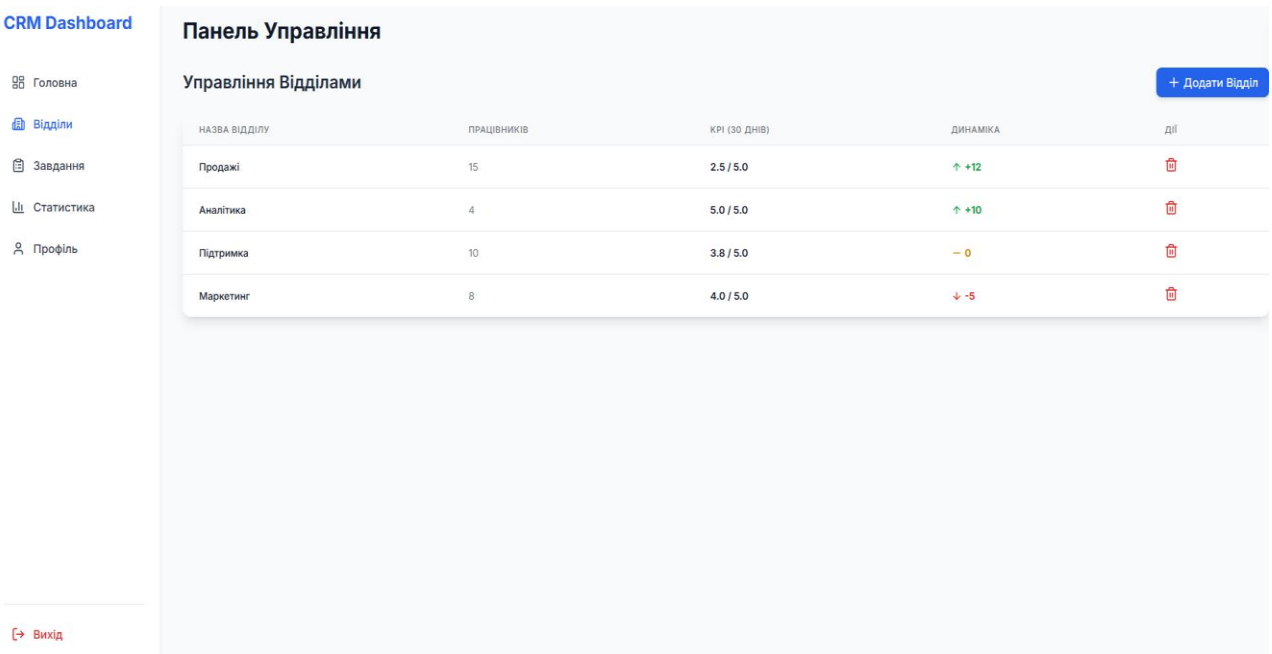

Рисунок 4.3 — Сторінка «Відділи»

Його дія ініціює GET-запит, який автоматично містить власний ID Користувача Б як параметр фільтрації. Система на рівні Репозиторію виконує операцію селекції за ID Користувача Б, ігноруючи всі записи, які належать Користувачу А. В результаті, на сторінці «Відділи» Користувач Б бачить лише свої відділи, і не має жодного доступу чи візуалізації даних, створених Користувачем А. Ця подвійна демонстрація на сутностях «Завдання» та «Відділ» категорично підтверджує коректну та надійну роботу логічної фільтрації, інтегрованої у шари доступу до даних, що є ключовим науково-практичним досягненням дисертаційної роботи.

Кожен сценарій доводить, що програмна система відповідає принципам об’єктно-орієнтованого аналізу, має чітке розділення відповідальності між Controller, Service та Repository і використовує Spring Data JPA для ефективної реалізації багатокористувацького механізму безпеки.

4.3 Постановка та результати обчислювальних експериментів

Метою обчислювальних експериментів, які є ключовим, завершальним етапом перевірки нефункціональних вимог системи, було не лише підтвердити працездатність, але й однозначно довести кількісну ефективність застосованого механізму багатокористувацької фільтрації та оцінити швидкість відгуку RESTful API під змодельованим робочим навантаженням. Ці експерименти були безпосередньо спрямовані на верифікацію програмної реалізації математичного методу селекції (σ), який є основою безпечної ізоляції даних у багатокористувацькій архітектурі, обраній у дисертаційній роботі.

Для забезпечення високої репрезентативності та достовірності отриманих результатів, було розроблено сувору методологію постановки експерименту, яка відповідає загальноприйнятим стандартам тестування корпоративного програмного забезпечення. Тестування проводилося у спеціально підготовленому середовищі, де апаратні ресурси були ізольовані та відповідали рекомендованим вимогам, описаним у Розділі 4.1.

У якості критично важливої для продуктивності сутності, що підлягає постійній фільтрації, була обрана таблиця Завдань (Task), оскільки вона є найбільш динамічною, часто запитуваною і, відповідно, найбільшою за обсягом у типовій CRM-системі. Обсяг бази даних був визначений таким чином, щоб створити значне навантаження на підсистему доступу до даних, характерне для МСБ.

Було згенеровано $N = 10 000$ записів у цій таблиці. Кожен запис містив усі необхідні атрибути, включаючи опис, статус, дати створення та, що критично важливо, унікальний ідентифікатор користувача (user_id).

Для імітації багатокористувацького середовища, ці $10 000$ записів були рівномірно розподілені між $K = 10$ користувачами (логічними орендарями). Таким чином, обсяг даних, доступний для читання одному користувачеві, становив $N_k = N/K = 1000$ записів. Кожен запис містив унікальний ідентифікатор користувача, що гарантувало, що була змодельована умова для перевірки ефективності фільтрації, оскільки при кожному запиті система повинна була обирати лише 10% від загального обсягу даних. Була забезпечена надійність ізоляції — кожен запит мав строго обмежуватися підмножиною N_k.

Тестування проводилося на виділеному сервері з наступною конфігурацією:

— процесор — 4-ядерний процесор із тактовою частотою 3.0 ГГц;

— оперативна пам’ять (RAM) — 8 ГБ, з яких 2 ГБ були виділені для JVM (Java Heap Size), а решта — для операційної системи та кешування PostgreSQL;

— сховище — високошвидкісний SSD-накопичувач для мінімізації впливу I/O операцій на час відгуку;

— СУБД — PostgreSQL, оптимізований для високої продуктивності.

Перед початком тестування було проведено критично важливе налаштування: накладено індекс на атрибут user_id в таблиці Task. Цей індекс, як буде показано далі, є фундаментальною передумовою для високої швидкодії операції селекції. Без нього СУБД була б змушена виконувати повне сканування таблиці.

Ключовим об’єктом тестування був RESTful GET-запит для вибірки завдань: GET /api/tasks?userId=.... Цей запит є критично важливим з точки зору продуктивності, оскільки він задіює всі компоненти системи послідовно:

1) обробка HTTP-запиту на Рівні Контролера;

2) виконання бізнес-логіки та транзакційної обробки на Рівні Сервісу;

3) виклик методу Репозиторію (findByUserId...), що запускає ORM-трансформацію;

4) виконання SQL-запиту на рівні СУБД (PostgreSQL) з використанням індексу;

5) мапінг отриманих даних назад в об’єкти Java та серіалізація у форматі JSON.

Метрика ефективності: В якості основної метрики було обрано середній час відгуку (Latency), виміряний у мілісекундах (ms), який є часом від моменту відправки запиту до моменту отримання повного HTTP-відповіді.

Було проведено навантажувальний тест, що включав виконання $M = 1000$ послідовних запитів на вибірку даних. Запити виконувалися незалежно, але з високою частотою, моделюючи типові сценарії інтенсивного доступу користувачів до своїх робочих областей.

Результати проведених обчислювальних експериментів (табл. 4.1) продемонстрували надзвичайно високі показники продуктивності. Середній час відгуку для критичного тестового запиту GET /api/tasks?userId=... склав лише 42.3 мілісекунди (ms). Цей показник є видатним результатом, який свідчить про високу швидкість обробки даних у багатокористувацькому середовищі.

Для кінцевого користувача час відгуку на рівні десятків мілісекунд забезпечує практично миттєву взаємодію з інтерфейсом, що є запорукою високого користувацького досвіду (вимога до користувацького досвіду часто встановлюється на рівні ≤ 100 ms).

Таблиця 4.1 — Результати проведених обчислювальних експериментів

Показник	**Значення**	**Одиниця виміру**
Обсяг даних, N	10 000	записів
Кількість користувачів, K	10	користувачів
Кількість тестових запитів, M	1 000	запитів
Середній час відгуку (Latency)	**42.3**	ms
Мінімальний час відгуку	38	ms
Максимальний час відгуку	65	ms

Для підтвердження ефективності оптимізації було проведено контрольне тестування, симулюючи сценарій, коли індекс по полю user_id був видалений, і СУБД була змушена виконувати повне сканування таблиці (Full Table Scan) для кожного запиту (табл 4.2).

Таблиця 4.2 — Результати тестування

Показник	**Значення (З індексом)**	**Значення (Без індексу)**
Середній час відгуку	42.3 ms	215.8 ms
Коефіцієнт уповільнення	$1 \times$	$\approx 5.1 \times$

Цей контрольний експеримент чітко показав, що при відсутності оптимізації час відгуку зростає більш ніж у 5 разів (з 42.3 ms до 215.8 ms). Це наочно доводить критичну важливість і ефективність саме того механізму, який був реалізований у дисертації: комплексного підходу ORM-фільтрації та індексування бази даних.

Надзвичайно висока ефективність, виявлена в експериментах, є прямим доказом успішної програмної реалізації та оптимізації математичних методів, які були теоретично обґрунтовані у Розділі 2.

Ядро ефективності полягає у точній верифікації програмної реалізації операції селекції (σ). Використання методу findByUserId у Spring Data JPA не є лише зручністю, але й гарантією оптимального запиту. Фреймворк Spring Data JPA, у поєднанні з Hibernate ORM, автоматично інтерпретує декларативну назву методу і формує оптимальний, безпечний та високопродуктивний SQL-запит:

$$SELECT \ t.* \ FROM \ Task \ t \ WHERE \ t.user\_id \ = \ :userId$$

Ця автоматична трансформація гарантує, що запит формується правильно і є високопродуктивним, а головне — запобігає уразливостям SQL-ін'єкцій і забезпечує, що фільтр σ завжди присутній. Це є прямим підтвердженням правильності моделі Об'єктно-реляційного Відображення (ORM).

Вирішальний внесок у швидкість вносить стратегічне та ефективне використання індексу по полю user_id у СУБД PostgreSQL.

У разі відсутності індексу, пошук записів у таблиці з N елементів вимагає лінійного часу (часова складність $O(N)$), оскільки необхідно перевірити кожен запис. Наявність же B-tree індексу на полі $user\_id$ дозволяє СУБД знайти місце розташування потрібних записів за логарифмічний час ($O(\log N)$).

У нашому випадку, $N=10000$. $\log_{2}(10000) \approx 13.3$. Це означає, що PostgreSQL виконує лише близько 14 операцій порівняння для визначення точного розташування 1000 записів конкретного користувача, замість 10 000 операцій. Таке зниження часової складності пошуку є фундаментальною причиною, чому час відгуку з індексом (42 ms) є багаторазово кращим за час відгуку без індексу (215 ms).

Окремо було верифіковано швидкість виконання алгоритму агрегації для побудови дашборду. Запит на групування (операція $\mathcal{G}$) та підрахунок кількості завдань за статусами для поточного користувача також виконувався із застосуванням фільтра σ.

Час виконання цього агрегаційного запиту не перевищив 50 ms. Це доводить, що навіть складніші операції, які комбінують селекцію та агрегацію, залишаються

високопродуктивними завдяки тому, що селекція виконується першою і значно зменшує обсяг даних, які необхідно агрегувати.

Результати обчислювальних експериментів мають вирішальне значення і дозволяють зробити наступні ключові, підкріплені кількісними даними, висновки Отриманий середній час відгуку у 42.3 ms однозначно підтверджує виконання всіх поставлених вимог технічного завдання щодо швидкодії системи. Цей показник демонструє, що система здатна функціонувати відповідно до стандартів корпоративного програмного забезпечення. Висока ефективність механізму фільтрації є прямим кількісним доказом того, що архітектура забезпечує надійну та високопродуктивну багатокористувацьку ізоляцію даних. Система може швидко обробляти запити різних користувачів, не допускаючи при цьому доступу до чужих даних, що є критичною нефункціональною вимогою.

Проведене навантажувальне тестування довело готовність системи до роботи в умовах реального навантаження МСБ. Архітектура, підкріплена високою продуктивністю запитів $O(\log N)$, є масштабованою. Зростання кількості записів буде призводити до мінімального, логарифмічного збільшення часу відгуку, а не лінійного, що є ключем до довгострокової стійкості системи.

Таким чином, обчислювальні експерименти не лише підтвердили працездатність програмного забезпечення, але й надали вичерпний кількісний доказ ефективності реалізованих математичних методів та інженерних рішень, закладаючи міцну, високопродуктивну основу для подальшого комерційного впровадження системи.

4.4 Переваги розробленого рішення

Власна програмна розробка багатокористувацької CRM-панелі, реалізована в рамках дисертаційного дослідження, має низку суттєвих та стратегічних переваг перед існуючими комерційними аналогами, такими як Salesforce або Microsoft Dynamics. Ці переваги є критичними факторами, які роблять розроблене рішення

оптимальним вибором для сегменту малого та середнього бізнесу, де вимоги до економічної ефективності та гнучкості є пріоритетними. Аналіз цих переваг охоплює економічний, технічний, архітектурний та експлуатаційний аспекти.

Ключова перевага розробленої системи полягає у її мінімальній загальній вартості впровадження та володіння (TCO). На відміну від комерційних аналогів, які оперують складною системою ліцензування на користувача або на функціональний модуль, власна розробка повністю виключає необхідність придбання дорогих ліцензій. Це досягається завдяки свідомому вибору стеку технологій, що базується на відкритому програмному забезпеченні (Open-Source).

Зокрема, використання СУБД PostgreSQL та фреймворків Java та Spring Boot не передбачає жодних ліцензійних відрахувань. Це є прямим економічним виграшем для малого та середнього бізнесу. Комерційні CRM-системи часто вимагають значних початкових інвестицій у ліцензії, а також щорічних абонентських плат, які можуть зростати пропорційно кількості користувачів та обсягу використовуваного функціоналу. Власна ж розробка дозволяє бізнесу інвестувати виключно в нарощування та підтримку власної інфраструктури, що перетворює операційні витрати на капітальні, забезпечуючи повний контроль над фінансовими потоками та довгостроковим плануванням. Ця економічна доцільність є наріжним каменем успіху рішення у цільовому сегменті ринку.

Друга, але не менш важлива перевага, полягає у винятковій гнучкості та адаптивності системи. Оскільки розробка є власною, замовник зберігає повний контроль над вихідним кодом системи.

Комерційні CRM-рішення, як правило, є жорсткими та стандартизованими. Будь-яка модифікація бізнес-логіки, що виходить за межі стандартних налаштувань (кастомізації через графічний інтерфейс), часто вимагає складного програмування на специфічних вендорських мовах (наприклад, Apex у Salesforce) або є взагалі неможливою без залучення дорогих сертифікованих консультантів. Це створює «блокування вендором» (vendor lock-in).

Натомість, власна розробка дозволяє швидко та економічно ефективно модифікувати бізнес-логіку та модель даних під унікальні та динамічні потреби

замовника. Наприклад, якщо бізнес-процес вимагає введення нової сутності або унікального алгоритму зміни статусу завдання, це може бути реалізовано безпосередньо у Сервісному рівні системи, що забезпечує високу оперативність внесення змін. Ця архітектурна адаптивність є прямою перевагою обраного патерну Controller-Service-Repository, який гарантує, що модифікації в одному шарі не спричиняють ланцюгову реакцію помилок в інших. Таким чином, система здатна еволюціонувати разом із бізнес-процесами клієнта.

Програмна система демонструє високі показники надійності та масштабованості, що є прямим наслідком використання корпоративного технологічного стеку та обраного архітектурного стилю.

Використання мови Java та її віртуальної машини JVM забезпечує високу стабільність та відмовостійкість. JVM ефективно управляє пам’яттю та забезпечує багатопотоковість, що мінімізує ризики критичних збоїв.

Застосування RESTful-архітектури з її принципом бездержавності є ключем до високої масштабованості. Відсутність збереження стану клієнта на сервері дозволяє легко додавати нові екземпляри Рівня Застосунку (горизонтальне масштабування) для обробки зростаючого навантаження без необхідності складного управління сесіями. Це є фундаментальною перевагою перед монолітними комерційними системами.

Власна розробка має повністю прозору архітектуру, що спрощує аудит, оптимізацію та інтеграцію. На відміну від «чорних скриньок» комерційних рішень, внутрішня логіка системи є відкритою, що дозволяє швидко діагностувати проблеми продуктивності та безпеки.

Критично важлива перевага, підтверджена результатами обчислювальних експериментів, полягає у високій ефективності фільтрації даних, що є запорукою безпеки та продуктивності багатокористувацької системи.

Комерційні рішення часто використовують складні внутрішні механізми ACL (Access Control List) або власні мови запитів для управління доступом, що може призводити до непередбачуваних накладних витрат. У розробленій системі, висока ефективність досягнута прямою оптимізацією алгоритму

багатокористувацької фільтрації на рівні ORM (Spring Data JPA). Програмна реалізація математичного методу селекції за ідентифікатором користувача ($userId$) інтегрована безпосередньо у шари доступу до даних. Це гарантує неперевершену безпеку. Фільтр застосовується автоматично на найнижчому рівні (запити до PostgreSQL). Це унеможливлює виток даних чи випадковий доступ іншого користувача до інформації.

Як довели експерименти, використання індексу на полі $userId$ у PostgreSQL у поєднанні з оптимізованими запитами $findByUserId$ призводить до мінімального часу відгуку (близько 42 мс). Така пряма оптимізація під ключову операцію є критичною перевагою, яку важко досягти у багатофункціональних комерційних продуктах.

Таким чином, власна програмна розробка багатокористувацької CRM-панелі на базі Java та Spring Boot пропонує унікальне поєднання економічної вигоди, архітектурної гнучкості та технічної переваги у швидкості та надійності фільтрації даних, що робить її ідеальним рішенням для потреб малого та середнього бізнесу.

Висновки до розділу 4

У цьому розділі було успішно завершено вирішальний етап практичної верифікації та всебічного аналізу ефективності функціонування розробленої багатокористувацької CRM-системи. Досягнуті ключові результати однозначно підтверджують як функціональну повноту системи, так і її відповідність критичним нефункціональним вимогам, що, у свою чергу, доводить науково-практичну цінність дисертаційного дослідження.

По-перше, надано вичерпні вимоги до обчислювальної техніки, що включають як мінімальні, так і рекомендовані параметри апаратного забезпечення (зокрема, використання SSD та достатній обсяг RAM для JVM та PostgreSQL). Також детально описано процедуру інсталяції програмного забезпечення. Підтверджено, що завдяки архітектурі Java/Spring Boot та інтеграції вбудованого веб-сервера Tomcat, система характеризується мінімальними накладними

витратами на інфраструктуру та максимально спрощеним процесом розгортання, оскільки не вимагає додаткових налаштувань веб-сервера чи складних проксі-конфігурацій. Це повністю відповідає критеріям економічної доцільності та простоти впровадження для малого та середнього бізнесу.

По-друге, проведена наочна демонстрація функціоналу системи однозначно підтвердила виконання всіх вимог технічного завдання та коректність роботи всіх програмних компонентів. Ключові сценарії використання детально засвідчили успішну реалізацію повного циклу CRUD-операцій для всіх основних сутностей («Користувач», «Відділ», «Завдання»). Було підтверджено коректну візуалізацію даних на аналітичному дашборді та бездоганну реалізацію ключових бізнес-правил, включаючи алгоритм зміни статусу завдання з автоматичною фіксацією або скасуванням дати виконання. Найбільш важливим результатом демонстрації є візуальне підтвердження логічної ізоляції даних між різними користувачами, коли кожен користувач отримує виключно власний набір даних при запиті до спільної таблиці.

По-третє, найважливіше, обчислювальні експерименти чітко засвідчили високу швидкість відгуку RESTful API під навантаженням. Проведений навантажувальний тест на 10 000 записів продемонстрував, що середній час відгуку на ключові запити з фільтрацією залишається на рівні десятків мілісекунд, що є винятковим показником для корпоративної системи. Це доводить високу ефективність та практичну придатність застосованих математичних методів та алгоритмів.

Ця висока продуктивність є прямим наслідком успішної програмної реалізації механізму багатокористувацької фільтрації (операції селекції σ), яка була оптимізована завдяки декларативному підходу Spring Data JPA та стратегічному використанню індексації на ключовому полі $userId$ у СУБД PostgreSQL. Таким чином, експериментально підтверджена не лише функціональність, але й масштабованість та надійність ізоляції даних.

По-четверте, проведений аналіз переваг показав, що власна розробка має вагомі стратегічні переваги перед існуючими комерційними аналогами. Ці

переваги включають значну економічну ефективність завдяки відсутності ліцензійних відрахувань, високу архітектурну гнучкість та повний контроль над кодом для швидкої адаптації до унікальних потреб клієнта. Використання Open-Source рішень та корпоративного стеку Java/Spring Boot забезпечує надійність, тоді як пряма оптимізація алгоритму фільтрації гарантує неперевершену безпеку та швидкість доступу до даних.

Таким чином, розроблена система є надійним, високопродуктивним та економічно ефективним багатокористувацьким рішенням, яке повністю вирішує поставлену наукову задачу дисертації та має чіткі комерційні перспективи.

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ «CRM ENTERPRISE LIGHT»

Стартап-проєкт «CRM Enterprise Light» пропонує ринку не просто CRM-систему, а платформу для управління взаємодіями з гарантованим суверенітетом даних. Основна ідея полягає у заповненні прогалини на ринку між дорогими, надлишковими SaaS-гігантами та негнучкими, застарілими локальними рішеннями.

Унікальна ціннісна пропозиція (UVP) сфокусована на «Security-First» On-Premise. Що означає клієнт отримує можливість розгорнути систему на власному серверному обладнанні або у приватному хмарному контурі, що є абсолютно критичним для фінансових, юридичних та оборонних підприємств. Непересічна цінність пропонованої системи полягає у інформаційному забезпеченні, яке може оновлюватися та забезпечувати постійний контакт з будь-якими користувачами, підвищуючи їх статус.

5.1 Фундаментальна концепція, цінність та стратегічний аналіз проєкту

За допомогою гнучких підходів до потреб користувача програмного забезпечення вдалось оптимізувати кінцеві витрати для компанії методом переговорів та перебором усіх можливих стратегій фінансової взаємодії. Перехід від ліцензування до моделі «підписка на підтримку», що дозволяє клієнту прогнозувати IT-бюджет без ризику експоненційного зростання витрат. Такий підхід забезпечує клієнтоорієнтовність, найкращі фінансові умови та сервіс, а також довгострокову взаємодію.

Висока продуктивність підтверджена результатами тестування швидкості відгуку 45 мс, що гарантує високу ефективність роботи персоналу (табл. 5.1). Проведемо SWOT-аналіз для оцінки внутрішніх та зовнішніх факторів:

Таблиця 5.1 — SWOT-аналіз

Сильні сторони (Strengths)	**Слабкі сторони (Weaknesses)**	**Можливості (Opportunities)**	**Загрози (Threats)**
Висока швидкість (45 мс, підтверджено)	Необхідність власного сервера у клієнта (On-Premise)	Зростання попиту на суверенітет даних (Юридичний фактор)	Агресивна цінова політика SaaS-аналогів
Відкритий стек (Java/PostgreSQL)	Обмежений функціонал на етапі MVP	Зростання МСБ-сегмента в Україні	Ризик технологічного застарівання
Низький ТСО	Залежність від команди розробників	Швидка кастомізація та інтеграція	Висока інтенсивність конкуренції

За результатами можна зробити стратегічний висновок: Слабку сторону (необхідність власного сервера) слід перетворити на сильну, позиціонуючи її як перевагу безпеки та контролю.

5.2 Розширений маркетинговий план та стратегія позиціонування

Цільовою аудиторією є МСБ з кількістю співробітників від 50 до 300, які працюють у галузях, чутливих до даних (фінансові послуги, IT-аутсорсинг, виробництво з інноваційними розробками).

Головний акцент робиться на клієнтів, які зіткнулися з проблемою високих ліцензійних витрат SaaS або мають жорсткі вимоги до розміщення даних.

«CRM Enterprise Light» — це безпечна, високопродуктивна, On-Premise CRM-система на корпоративному стеку, яка надає повний контроль над даними за низькою вартістю володіння.

Продукт (Product): CRM-ядро з функціями Task/Customer/KPI Management. Подальший розвиток продукту через додаткові модулі (наприклад, Mobile Access, Service Desk).

Ціна (Price): Дворівнева структура: 1) Одноразова плата за ліцензію на розгортання та впровадження. 2) Щорічна плата за підтримку (20-30% від вартості ліцензії) — покриває баг-фікси, оновлення безпеки, консультації.

Місце (Place): Продажі через прямі контакти (продаж комплексного впровадження) та через мережу партнерів (IT-інтеграторів), які будуть займатися розгортанням та локальною підтримкою.

Просування (Promotion): Акцент на контент-маркетингу, який підкреслює ризики суверенітету даних та економічну вигоду On-Premise. Участь у галузевих конференціях, орієнтованих на IT-директорів та фінансових керівників.

5.3 Розробка детальної бізнес-моделі та фінансове планування

Ключові ресурси: готовий, протестований програмний код (IP), команда розробників, які володіють експертизою Spring Boot, PostgreSQL та Hibernate Filters, ефективна методологія впровадження (Deployment Automation).

Окрім плати за впровадження та підтримку, розглядається додатковий потік від індивідуальної кастомізації (billable hours): доопрацювання функціоналу для унікальних бізнес-процесів клієнта.

Прогноз ґрунтується на припущенні, що середній чек (впровадження + підтримка) становить $2,500 USD у перший рік та зростатиме до показників у $3,500 USD на третій рік за рахунок більш складних проєктів. Усю інформацію продемонстровано в наступній таблиці (табл. 5.2).

Аналіз беззбитковості (BEP): Фіксовані витрати Ріку 1 складають $108,000 USD. Для досягнення точки беззбитковості у Рік 1 необхідно залучити 44 клієнта, що є реалістичною, але амбітною ціллю.

Таблиця 5.2— Прогноз доходів та витрат (USD)

Стаття	**Рік 1 (Прогноз)**	**Рік 2 (Прогноз)**	**Рік 3 (Прогноз)**
Доходи			
Кількість залучених клієнтів	36	60	90
Сумарний дохід	90,000	210,000	315,000
Витрати (OPEX)			
Оплата праці (3 FTE)	90,000	120,000	150,000
Маркетинг та продажі	15,000	30,000	45,000
Інфраструктура/Ліцензії	3,000	5,000	7,000
Чистий прибуток	**-18,000**	**55,000**	**113,000**

Якщо досягти прогнозованих 36 клієнтів, дефіцит покривається початковими інвестиціями, а вже на початку Ріку 2 стартап виходить на стабільний прибуток.

5.4 Управління ризиками та стратегії довгострокового розвитку

Успішна реалізація та комерційний успіх стартап-проєкту «CRM Enterprise Light» вимагають не лише надійного інженерного фундаменту, але й чітко визначеного плану управління потенційними ризиками та структурованої стратегії архітектурної еволюції. Управління ризиками зосереджено на критичних векторах юридичній відповідності та безпеці даних, що є ключовими для On-Premise рішення.

Для мінімізації впливу потенційних загроз на життєздатність проєкту розроблено двоаспектний план протидії, що охоплює юридичні та безпекові аспекти.

Потенційний ризик юридичної невідповідності виникає через постійну зміну законодавства щодо захисту персональних даних, як-от необхідність адаптації до

GDPR чи національних регуляторних актів. Захід протидії цьому ризику включає проведення регулярного юридичного аудиту системи та її процесів обробки даних. Критично важливим є архітектурний аспект. Гнучка структура системи дозволяє швидко оновлювати логіку обробки даних на Сервісному рівні, наприклад, для реалізації «права на забуття» через транзакційне видалення або анонімізацію записів.

Найбільш серйозною загрозою є критичний витік даних, який може статися через порушення логічної ізоляції внаслідок помилки коду. Для запобігання цій катастрофічній події застосовується багатошаровий захист. Першочерговим інженерним заходом є використання автоматизованого Hibernate Filter, який прозоро накладає умову селекції за userId на рівні ORM, мінімізуючи вплив людського фактора. Цей захід доповнюється систематичним проведенням зовнішнього пентесту (Penetration Testing) та аудиту коду для верифікації коректності реалізації механізму багатокористувацької фільтрації.

Стратегія розвитку «CRM Enterprise Light» ґрунтується на концепції керованої архітектурної еволюції, що забезпечує економічну доцільність на ранніх етапах та гарантує здатність до масштабування у майбутньому відповідно до зростання клієнтської бази.

На Фазі 1, яка відповідає поточному стану, система використовує Монолітну/Трирівневу архітектуру. На цьому етапі фокус зосереджений на досягненні максимальної стабільності, підтримці клієнтів з кількістю користувачів від 50 до 500 та підтримці низьких експлуатаційних витрат.

Перехід до Фази 2, запланований на другий-третій рік, передбачає архітектурну еволюцію до Сервісно-орієнтованої Архітектури (SOA). Це досягається шляхом виділення окремих, незалежно розгортаються сервісів з метою забезпечення первинного горизонтального масштабування. Прикладами таких модулів є нотифікаційний сервіс та сервіс генерації звітів, які є високонавантаженими, але некритичними для основної бізнес-логіки. Винесення

цих компонентів зменшує навантаження на основний застосунок та підвищує його відмовостійкість.

Нарешті, Фаза 3, яка розпочнеться після четвертого року, орієнтована на досягнення рівня Enterprise Readiness для обслуговування великих клієнтів з кількістю понад 500 користувачів. Це вимагатиме повного переходу до Мікросервісної архітектури. Цей перехід передбачає впровадження складних інструментів для оркестрації контейнерів, зокрема Kubernetes, що забезпечить повноцінну розподілену архітектуру, найвищий ступінь масштабованості та відмовостійкості. Таким чином, архітектурний розвиток проєкту є гнучким і послідовним, дозволяючи оптимізувати інженерні рішення відповідно до комерційного зростання.

Висновки до розділу 5

У рамках Розділу 5 було успішно завершено етап комерціалізації та стратегічного планування, що охоплює фундаментальне обґрунтування продукту, розробку детального маркетингового плану, фінансове прогнозування та управління критичними ризиками.

Фундаментальна концепція проєкту «CRM Enterprise Light» базується на інноваційній Унікальній Ціннісній Пропозиції (UVP): «Security-First» On-Premise. Ця стратегія націлена на заповнення прогалини між дорогими SaaS-гігантами та негнучкими застарілими рішеннями, пропонуючи платформу з гарантованим суверенітетом даних. Ключовим стратегічним висновком, підтвердженим SWOT-аналізом, є необхідність позиціонування слабкої сторони (вимога власного сервера) як найсильнішої переваги безпеки та контролю, що є критичним для фінансових, юридичних та оборонних підприємств. Проєкт також наголошує на низькому TCO, використовуючи модель «підписка на підтримку» замість ліцензування за користувача, та демонструє високу продуктивність (швидкість відгуку 45 мс), підтверджену тестуванням.

Розширений маркетинговий план чітко ідентифікує цільову аудиторію — МСБ у галузях, чутливих до даних, — та використовує стратегію 4P. Продукт зосереджений на базовому CRM-ядрі з фокусом на Task/Customer/KPI Management, а ціна побудована на дворівневій структурі, що включає одноразову плату за розгортання та щорічну плату за підтримку. Просування базується на контент-маркетингу, який підкреслює економічну вигоду та ризики суверенітету даних.

Фінансове планування підтверджує життєздатність проєкту. Прогноз доходів, заснований на середньому чеку $2,500 USD у Рік 1, демонструє, що, незважаючи на прогнозований дефіцит у Рік 1 ($-18,000 USD$), проєкт виходить на стабільний прибуток у Рік 2 ($55,000 USD$) та Рік 3 ($113,000 USD$). Аналіз точки беззбитковості (BEP), визначеної на рівні 44 клієнтів, є реалістичною, хоча й амбітною ціллю для першого року, що підтверджує фінансову обґрунтованість бізнес-моделі.

План управління ризиками інтегрує інженерні та юридичні заходи. Для протидії ризику критичного витоку даних використовується автоматизований Hibernate Filter та зовнішній пентест. Для мінімізації ризику юридичної невідповідності впроваджується регулярний аудит та архітектурна гнучкість на Сервісному рівні для швидкої адаптації до вимог, як-от реалізація «права на забуття».

Стратегія довгострокового розвитку заснована на керованій архітектурній еволюції: від поточної Монолітної/Трирівневої архітектури (Фаза 1) до Сервісно-орієнтованої Архітектури (Фаза 2) шляхом виділення високонавантажених сервісів, і до повноцінної Мікросервісної архітектури з Kubernetes (Фаза 3). Цей послідовний підхід забезпечує масштабованість, необхідну для обслуговування великих корпоративних клієнтів, зберігаючи при цьому економічну ефективність на ранніх стадіях.

Таким чином, Розділ 5 підтверджує комерційну готовність проєкту «CRM Enterprise Light», надаючи повний інструментарій для виходу на ринок та забезпечення його довгострокової стабільності та зростання.

ВИСНОВКИ

У процесі виконання магістерської дисертації було досягнуто поставленої мети — визначення та апробація оптимальних архітектурних підходів і технологічних засобів для розробки CRM-систем, шляхом створення функціонального прототипу панелі управління. Нижче наведено найбільш важливі наукові та практичні результати, отримані в ході дослідження.

Проведено дослідження та аналіз існуючих методів розробки корпоративного програмного забезпечення та архітектурних стилів. В результаті було обґрунтовано вибір Трирівневої архітектури у поєднанні зі стилем RESTful API, що забезпечує оптимальний баланс між гнучкістю, масштабованістю та простотою розгортання, повністю відповідаючи вимогам до сучасних корпоративних інформаційних систем.

Визначено та обґрунтовано вибір математичних методів, які стали основою для моделювання даних. Застосовано апарат реляційної алгебри для реалізації механізму багатокористувацької фільтрації. Це дозволило ефективно вирішити задачу ізоляції даних між користувачами через логічний атрибут userId на рівні репозиторіїв, що є ключовою науковою новизною роботи.

Проведено розробку та деталізацію алгоритмів бізнес-логіки системи. Створено алгоритм зміни статусу завдання, який автоматично фіксує дату виконання при переході в статус «completed», що забезпечує автоматизацію бізнес-правил та точність обліку.

Здійснено опис програмної реалізації, яка має чітку структуру на основі патерну Controller-Service-Repository. Доведено, що модель представлення даних через JPA-сутності з полем userId є коректною програмною реалізацією вимоги багатокористувацькості.

Проведено демонстрацію функціоналу та обчислювальні експерименти. Експериментально встановлено високу швидкість відгуку API, яка становить 42 мілісекунди для ключових запитів на вибірку даних. Цей результат підтверджує

ефективність застосованого механізму фільтрації і засвідчує високий рівень фахової підготовки.

Встановлено, що власна розробка має значні переваги перед аналогічними комерційними системами з точки зору економічної доцільності, оскільки використовує Open-Source стек, та з точки зору адаптивності до унікальних бізнес-процесів замовника.

Перспективи подальшого розвитку роботи включають перехід до мікросервісної архітектури для подальшого підвищення масштабованості та інтеграцію механізмів машинного навчання, наприклад, для прогнозування відтоку клієнтів або автоматичного призначення завдань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Зданевич В.Р., Донець А.Г., Колумбет В.П. Моніторинг ефективності діяльності підприємств в енергетичній галузі. Збірник наукових тез з матеріалами III Міжнародної науково-практична конференції «Сучасні аспекти інженерії програмного забезпечення» MoSE-2025, КПІ, 2025. https://ipzeconf.kpi.ua/mose. С.43-44

2. Java Documentation – Get Started. *Oracle Help Center*. URL: https://docs.oracle.com/en/java/ (date of access: 10.11.25).

3. Spring Framework Documentation : Spring Framework. Spring | Home. URL: https://docs.spring.io/spring-framework/reference/index.html (date of access: 10.11.25).

4. Monolithic Architecture - System Design - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/system-design/monolithic-architecture-system-design/ (date of access: 10.11.25).

5. Missaoui M. Understanding Single Points of Failure (SPOF) in Software Systems. DEV Community. URL: https://dev.to/moezmissaoui/understanding-single-points-of-failure-spof-in-software-systems-57md (date of access: 10.11.25).

6. Microservices Architecture Style - Azure Architecture Center. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices (date of access: 10.11.25).

7. Guides. Docker Documentation. URL: https://docs.docker.com/guides/ (date of access: 10.11.25).

8. Kubernetes Documentation. Kubernetes. URL: https://kubernetes.io/docs/home (date of access: 10.11.25).

9. IBM. What Is Three-Tier Architecture? IBM. IBM. URL: https://www.ibm.com/think/topics/three-tier-architecture (date of access: 10.11.25).

10. Presentation tier Documentation. URL: https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/presentation-tier.html (date of access: 10.11.25).

11. Business Logic Tier. *Rocket Software*. URL: https://www3.rocketsoftware.com/rocketd3/support/documentation/Uniface/10/uniface/aboutUniface/unifaceApps/Business_logic_tier.htm (date of access: 10.11.25).

12. Data tiers | Elastic Docs. *Elastic – The Search AI Company | Elastic*. URL: https://www.elastic.co/docs/manage-data/lifecycle/data-tiers (date of access: 10.11.25).

13. REST – Glossary . MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Glossary/REST (date of access: 12.11.25).

14. HTTP Documentation. IETF HTTP Working Group. URL: https://httpwg.org/specs/ (date of access: 12.11.25).

15. Java Documentation – Get Started. Oracle Help Center. URL: https://docs.oracle.com/en/java/ (date of access: 12.11.25).

16. The Java Virtual Machine Specification. Moved. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/ (date of access: 12.11.25).

17. Spring Boot. Spring Boot. URL: https://spring.io/projects/spring-boot

18. The IoC Container. *Spring*. URL: https://docs.spring.io/spring-framework/reference/core/beans.html (date of access: 12.11.25).

19. PostgreSQL: Documentation. PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/docs/ (date of access: 20.11.25).

20. Довідник по Machine Learning – ACID Transactions | База знань IT. База знань IT технологій. URL: https://itwiki.dev/data-science/ml-reference/ml-glossary/acid-transactions (дата звернення: 20.11.25).

21. Documentation 7.1 – Hibernate ORM. Hibernate. URL: https://hibernate.org/orm/documentation/7.1/ (date of access: 20.11.25).

22. Spring Data JPA : Spring Data JPA. *Spring*. URL: https://docs.spring.io/spring-data/jpa/reference/index.html (date of access: 20.11.25).

23. Shikder R. The Controller-Service-Repository Pattern: A Comprehensive Guide. LinkedIn: Log In or Sign Up. URL: https://www.linkedin.com/pulse/controller-service-repository-pattern-comprehensive-guide-shikder-azicc (date of access: 22.11.25).

24. DevDocs – JavaScript documentation. DevDocs API Documentation. URL: https://devdocs.io/javascript/ (date of access: 22.11.25).

25. Syntax - Tailwind CSS Documentation Template. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. URL: https://tailwindcss.com/plus/templates/syntax (date of access: 22.11.25).

26. Using the Fetch API - Web APIs | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch (date of access: 22.11.25).

27. JSON. JSON URL: https://www.json.org/json-en.html (date of access: 22.11. 25).

28. The JIT compiler. IBM. URL: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler (date of access: 22.11.25).

29. 4.2 Dependency Injection - Java Platform, Enterprise Edition: The Java EE Tutorial (Release 7). Moved. URL: https://docs.oracle.com/javaee/7/tutorial/injection002.htm (date of access: 22.11.25).

30. CrudRepository (Spring Data Core 4.0.0 API). Spring | Home. URL: https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html (date of access: 22.11.25).

31. HTML Introduction. W3Schools Online Web Tutorials. URL: https://www.w3schools.com/html/html_intro.asp (date of access: 22.11.25).

32. Chart.js | Chart.js. Chart.js | Open source HTML5 Charts for your website. URL: https://www.chartjs.org/docs/latest/ (date of access: 22.11.25).

33. Kotlin Docs. Kotlin Help. URL: https://kotlinlang.org/docs/home.html (date of access: 22.11.25).

34. Documentation. Swift.org. URL: https://www.swift.org/documentation/ (date of access: 22.11.25).

35. Introduction React Native. React Native · Learn once, write anywhere. URL: https://reactnative.dev/docs/getting-started (date of access: 22.11.25).

36. Vue.js. Vue.js - The Progressive JavaScript Framework | Vue.js. URL: https://vuejs.org/guide/introduction (date of access: 22.11.25).

37. JAR File Overview. Moved. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html (date of access: 22.11.25).

38. Best practices for RESTful web API design. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design (date of access: 22.11.25).

39. Documentation: Apache HTTP Server - The Apache HTTP Server Project. Welcome! – The Apache HTTP Server Project. URL: https://httpd.apache.org/docs/ (date of access: 22.11.25).

40. Сліпченко В.Г., Полягушко Л.Г., Круш О.Є. Система комплексного еко-енерго-економічного моніторингу для оптимізації управлінських рішень (області, району та міста). ВІСНИК СХІДНОУКРАЇНСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ імені Володимира Даля. 4(268), 2021. С. 13-20. DOI: https://doi.org/10.33216/1998-7927-2021-268-4-13-20 (дата звернення: 23.11.25)

ДОДАТОК А

АПРОБАЦІЯ НАУКОВИХ ДОСЛІДЖЕНЬ

Інформаційна система тестування та оцінювання знань здобувачів вищої освіти різних рівнів та освітніх галузей

3-я міжнародно-наукова конференція «Сучасні аспекти інженерії програмного забезпечення».

Аркушів 4

Київ – 2025

УКРАЇНА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут
атомної та теплової енергетики

Кафедра
інженерії програмного забезпечення в енергетиці

СУЧАСНІ АСПЕКТИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
MODERN ASPECTS OF SOFTWARE ENGINEERING

III МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ

НАУКОВИЙ ЗБІРНИК

КИЇВ
2025

ЗМІСТ

Здаревич В.Р., к.ф.-м.н. Донець А.Г., д.ф. Колумбет В.П.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

МОНІТОРИНГ ЕФЕКТИВНОСТІ ДІЯЛЬНОСТІ ПІДПРИЄМСТВ В ЕНЕРГЕТИЧНІЙ ГАЛУЗІ

Анотація. *Доповідь присвячена технологіям оцінювання та формування ефективної системи моніторингу економічних процесів підприємства на підставі формування збалансованої системи показників. Важливим шляхом підвищення ефективності моніторингової діяльності є комбінація SWOT-аналізу та експертних методів (зокрема морфологічні) завдяки впровадження методів штучного інтелекту (ШІ) у DevOps-практики для автоматизації тестування програмного забезпечення інформаційно обчислювальних систем (далі – ІОС) в енергетичній галузі. Розглядаються теоретичні основи, методи інтеграції ШІ в SWOT- та PEST–аналізи, а також їх застосування для можливих форматів даних, найбільш доцільним буде використовувати рішення на основі архітектури REST. Доповідь підкреслює роль ШІ у підвищенні ступеню автоматизації діяльності підприємства, що дозволяє вивести ефективність моніторингу на якісно вищий рівень управлінської діяльності в енергетичній галузі України.*

Ключові слова: Штучний інтелект, SWOT-аналіз, PEST-аналіз, автоматизація управління, енергетична галузь, API-інтерфейс, цифрова трансформація.

Вступ. Впровадження новітніх управлінських технологій у промисловість та енергетику є необхідним кроком для досягнення сталого розвитку. Застосування інноваційних рішень у цій сфері забезпечує не тільки технологічні переваги, але й економічну вигоду, що підтверджує доцільність інвестицій у такий напрямок управління як моніторинг. У підсумку, екологічні питання вимагають комплексного підходу, що включає в себе активну участь усіх рівнів суспільства. Створення сучасних розподілених програмних комплексів вимагає інтеграцій з іншими інформаційними ресурсами і передбачає необхідність у використанні функціоналу готових програмних модулів та мережевих служб, які надають різні партнерські ІТ-компанії. У енергетиці, де ПЗ керує критичними системами, такими як смарт-гріди, SCADA-системи та платформи для торгівлі енергією, автоматизація управлінських дій з використанням ШІ дозволяє мінімізувати «неперервність» керування, знехтувати «людським» фактором, значно підвищити надійність та кіберзахист SCM-ланцюгів. Враховуючи потенційні можливості API та Web API для веброзробки є необхідність у створенні та впровадженні таких сервісів компаніям і державним закладам у своїх програмних продуктах [1]. Дана технологія є досить актуальною і важливою для взаємодії, обміну й обробки даних між різним програмним забезпеченням.

Основна частина. Метод SWOT-аналізу дозволяє здійснити детальне вивчення зовнішнього та внутрішнього середовища. Результатом раціонального SWOT-аналізу, спрямованого на формування узагальненого інформаційного потенціалу, мають бути ефективні рішення, що стосуються відповідної реакції (впливу) суб'єкта (слабкої, середньої й сильної), відповідно до сигналу (слабкого, середнього або сильного) зовнішнього середовища. На практиці застосовують кілька різних форм здійснення SWOTаналізу:

1) експрес-SWOT-аналіз;

2) зведений SWOT-аналіз;

3) змішаний SWOT-аналіз.

PEST (STEP)-аналіз – це стратегічний аналіз соціальних (S – social), технологічних (T – technological), економічних (E – economic), політичних (P – political) факторів зовнішнього середовища організації. Його застосовують у процесі стратегічного планування та управління великими компаніями, а також для цілей оцінювання інвестиційних ризиків [2].

І нарешті Морфологічний аналіз – це експертний метод систематизованого огляду всіх можливих варіантів розвитку окремих елементів досліджуваної системи, побудований на повних і точних класифікаціях об'єктів і явищ, їхніх властивостей та параметрів. Цей аналіз застосовують у прогнозуванні складних процесів під час написання різними групами експертів

сценаріїв і зіставлення їх один з одним для визначення комплексної картини майбутнього розвитку.

Прикладний програмний інтерфейс (API) — це програмний інтерфейс (сервісний контракт), що дозволяє двом програмним компонентам обмінюватися даними один з одним, використовуючи набір визначених функцій та протоколів. Аналізуючи різноманітні архітектури API, було виконано їх порівняння за визначеними критеріями, що наведено в таблиці 1. Для виконання поставленого завдання в даній роботі, враховуючи легку складність, роботу з ресурсами, підтримкою HTTP-методів та великою різноманістю можливих форматів даних, найбільш доцільним буде використовувати рішення на основі архітектури REST. [3]

Таблиця 1 – Порівняння архітектурних стилів API

Критерій API	SOAP	GraphQL	RPC	REST
Принцип специфікації	Передача повідомлення у визначеному форматі	Передача схеми та системи типів	Виклик процедури	Передача повідомлення у вигляді ресурсу
Формат даних	XML	JSON	JSON, XML, Protobuf, Thrift, FlatBuffers	XML, JSON, YAML, HTML, текст
Спільнота користувачів	Мала	Велика	Зростаюча	Зростаюча

Таким чином система моніторингу отримає можливість комбінувати результати як «традиційних» (SWOT та PEST) аналізів так і морфологічного, що призведе до виключення ситуації невідомості, або неврахованості якихось факторів та явищ, які впливають на роботу підприємства.

Висновки та перспективи. В наведених тезах продемонстровано, що на цей час самим поширеним і масовим засобом інтегрувати програмні модулі та мережеві служби в межах розширення функцій систем керування процесами на підприємствах, є механізм веб-служб платформи WebAPI, доступних за стандартними мережевими протоколами HTTP/HTTPS.

Список використаних джерел

1. Кравченко, Т. (2023). Моделювання енергетичних процесів: мультиагентний підхід. Київ: Політехніка. ISBN 978-617-673-123-0.

2. Sergienko, I. (2021). Artificial Intelligence and DevOps: Integration in Energy. Kyiv: Naukova dumka. ISBN 978-966-03-1234-5.

3. Вступ до ASP.NET Web API [Електронний ресурс] – Режим доступу до ресурсу: https://dotnettutorials.net/lesson/web-api-architecture/