Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка API для системи управління завданнями із використанням Spring Framework»

Виконав: студент 2 курсу, групи 1ІСТ-24м спеціальності 126 – Інформаційні системи та технології.
(шифр і назва спеціальності)

Анатолій ГАЛІБРОДА
(ПІБ студента)

Керівник: к. т. н., доцент каф. АІІТ

Владислав КАБАЧІЙ
(науковий ступінь, вчене звання / посада, ПІБ керівника)

«5» 12 2025 р.

Опонент: Марія ЮХИМЧУК
(науковий ступінь, вчене звання / посада, ПІБ опонента)

«11» 12 2025 р.

Допущено до захисту
Завідувач кафедри АІІТ
д.т.н., проф. Олег БІСІКАЛО
(науковий ступінь, вчене звання)

«12» 12 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра автоматизації та інтелектуальних інформаційних технологій
Рівень вищої освіти ІІ-й (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 126 – Інформаційні системи та технології
Освітньо-професійна програма – Інформаційні технології аналізу даних та зображень

ЗАТВЕРДЖУЮ
Завідувач кафедри АІІТ
д.т.н., проф. Бісікало О. В.

«26» вересня 2025 р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Галіброді Анатолію Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: Розробка API для системи управління завданнями із використанням Spring Framework
Керівник роботи Кабачій В. В. к. т. н., доцент кафедри АІІТ
Затверджені наказом ВНТУ від "24" вересня 2025 року №313
2. Строк подання студентом роботи 10.12.2025 р.
3. Вихідні дані до роботи:
операційна система: Ubuntu 20.04 або вище/Windows 7 або вище/Mac 10.12 або вище. Мінімальні вимоги до ресурсів: CPU 2 ядра частотою 3.5 ГГц, RAM 4 ГБ, Відеопам'ять 1 ГБ, місце на диску до 1 ГБ.
4. Зміст текстової частини: вступ, аналіз предметної області та постановка задачі, проєктування API для системи управління завданнями, реалізація API на базі Spring Framework, експериментальне дослідження та оцінка ефективності, економічна частина, висновки, список використаних джерел
5. Перелік ілюстративного (або графічнного) матеріалу: ER-діаграма, UML діаграма класів, UML діаграми послідовності (автентифікація, дані користувача, створення списків завдань, видалення завдань), UML діаграма пакетів.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	виконання прийняв
1-4	Владислав КАБАЧІЙ к. т. н., доцент кафедри АІІТ	25.09.2025	29.09.2025
5	Наталія БУРЄННІКОВА д.е.н., проф. каф. ЕПтаВМ	04.11.2025	07.11.2025

7. Дата видачі завдання 25.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Строк виконання етапів роботи	Примітка
1.	Аналіз методів розробки та постановка задачі дослідження	30.09.2025	виконав
2.	Дослідження інструментів та розробка архітектури	13.09.2025	виконав
3.	Розробка та тестування програмного забезпечення	18.11.2025	виконав
4.	Підготовка економічного розділу	25.11.2025	виконав
5.	Оформлення пояснювальної записки і графічного матеріалу	28.11.2025	виконав
6.	Попередній захист роботи	01.12.2025	виконав
7.	Захист роботи	18.12.2025	виконав

Студент _____ (підпис) Галіброда А. С.

Керівник роботи _____ (підпис) Кабачій В. В.

АНОТАЦІЯ

УДК 004.42

Галіброда А. С. Розробка API для системи управління завданнями із використанням Spring Framework. Магістерська кваліфікаційна робота зі спеціальності 126 – Інформаційні системи та технології, освітня програма – Інформаційні технології аналізу даних та зображень. Вінниця: ВНТУ, 2025. 157 с.

На укр. мові. Бібліогр.: 35 назв; рис.: 27; табл.: 7.

У магістерській кваліфікаційній роботі подано результати дослідження, проєктування та реалізації API для системи управління завданнями, орієнтованої на використання в багатокористувацьких середовищах. Обґрунтовано доцільність створення серверного застосунку, що реалізує архітектурний стиль REST і інтегрує сучасні підходи до побудови сервісно-орієнтованих систем. Наведено аналіз існуючих рішень та технологій, визначено переваги використання Java, Spring Boot Framework та PostgreSQL.

Робота містить детальний опис архітектурних принципів, моделювання структури даних, створення сутностей доменної області та побудови сервісного шару з використанням репозиторіїв і DTO. Впроваджено механізми автентифікації та авторизації на основі JWT-токенів, що забезпечує надійний контроль доступу. У роботі представлено результати експериментального дослідження продуктивності.

Практичне значення роботи полягає в розробці серверної частини системи управління завданнями, яка може бути інтегрована у більші корпоративні рішення, адаптована до потреб командної роботи та розширена шляхом додавання нових функціональних модулів. Запропонований підхід дозволяє створити високонавантажені системи завдяки чіткій архітектурі, модульності та застосуванню перевірених інженерних практик.

Ключові слова: API, Spring, управління завданнями, системи управління проєктами, REST, автентифікація, JWT, PostgreSQL.

ABSTRACT

Halibroda A. S. Development of an API for a task management system using the Spring Framework. Master's qualification work in specialty 126 – Information Systems and Technologies, educational program – Information Technologies for Data and Image Analysis. Vinnytsia: VNTU, 2025. 157 p.

In Ukrainian language. Bibliography: 35 titles; figures: 27; tables: 7.

The master's qualification work presents the results of research, design, and implementation of an API for a task management system intended for use in multi-user environments. The feasibility of creating a server application that implements the REST architectural style and integrates modern approaches to building service-oriented systems is substantiated. An analysis of existing solutions and technologies is provided, highlighting the advantages of using Java, the Spring Boot Framework, and PostgreSQL.

The work includes a detailed description of architectural principles, data structure modeling, creation of domain entities, and development of the service layer using repositories and DTOs. Mechanisms for authentication and authorization based on JWT tokens have been implemented to ensure secure access control. The thesis also presents the results of experimental performance evaluation.

The practical significance of the work lies in the development of the server side of a task management system that can be integrated into larger corporate solutions, adapted to team collaboration needs, and extended with new functional modules. The proposed approach enables the creation of high-load systems due to its clear architecture, modularity, and the use of proven engineering practices.

Keywords: API, Spring, task management, project management systems, REST, authentication, JWT, PostgreSQL.

ЗМІСТ

ВСТУП

Актуальність роботи. У сучасних умовах цифрової трансформації бізнесу системи управління завданнями (task management systems) відіграють ключову роль у забезпеченні ефективної організації робочих процесів як у командах, так і серед індивідуальних користувачів. Вони дозволяють формувати та структурувати робочі плани, контролювати виконання задач, координувати взаємодію між учасниками проєкту та оперативно реагувати на зміни. У корпоративному середовищі такі системи стають основою прозорості проєктної діяльності, підвищення відповідальності працівників та оптимізації внутрішніх комунікацій.

Попри значну кількість існуючих комерційних рішень, більшість із них мають суттєві обмеження. Частина сервісів орієнтована на масового користувача і містить лише базові інструменти для керування завданнями, що недостатньо для складних бізнес-процесів. Інші платформи часто є монолітними, складно масштабуються, мають високий поріг входу або не дозволяють повноцінно інтегруватися з внутрішніми корпоративними системами. Це створює перешкоди для компаній, яким необхідно адаптувати функціональність під власні робочі сценарії, розширювати можливості системи або включати її у більшу інфраструктуру цифрових сервісів.

Саме тому зростає потреба у створенні власного спеціалізованого програмного продукту – REST API для системи управління завданнями. REST API виступає ядром серверної логіки та забезпечує низку критично важливих переваг:

- масштабованість, що дозволяє системі підтримувати збільшення кількості користувачів і обсягів даних;
- безпечність, завдяки впровадженню сучасних механізмів автентифікації та авторизації;

- гнучкість та розширюваність, що дозволяють легко додавати нові модулі, змінювати бізнес-правила або інтегрувати систему з зовнішніми сервісами;
- універсальність, оскільки один серверний інтерфейс може використовуватися різними клієнтськими застосунками – веб-інтерфейсами, мобільними додатками чи сторонніми платформами.

Розробка власного REST API є актуальною як з точки зору технічних переваг, так і з позиції реальних потреб сучасних організацій, яким необхідні адаптивні, надійні та масштабовані інструменти для керування завданнями в умовах динамічного цифрового середовища.

Метою роботи підвищення ефективності командної взаємодії, забезпечення прозорості робочих процесів і автоматизація управління завданнями за допомогою сучасних технологій Java та Spring Framework. Розроблений API має забезпечити надійне зберігання даних, зручний обмін інформацією між користувачами та можливість інтеграції з іншими системами.

Для досягнення поставленої мети потрібно вирішити **наступні задачі**:

- Провести аналіз предметної області та існуючих аналогів систем управління завданнями.
- Дослідити архітектурні підходи до побудови REST API та обґрунтувати вибір технологій і фреймворків.
- Спроєктувати архітектуру API, визначити структуру даних і зв’язки між сутностями.
- Розробити специфікацію ендпоінтів та механізм безпеки на основі Spring Security і JWT.
- Підготувати основу для подальшої реалізації програмного рішення та інтеграції з клієнтськими застосунками.

Об’єктом дослідження є процес управління завданнями в межах інформаційної системи, який охоплює створення, призначення, моніторинг і контроль виконання задач у командному або індивідуальному середовищі.

Предметом дослідження є засоби, технології та методи розроблення REST API для побудови системи управління завданнями, зокрема архітектурні

рішення, принципи побудови шарової структури застосунку, методи забезпечення безпеки та зберігання даних.

Основний науково-технічний результат роботи роботи полягає у створенні концептуальної моделі REST API для таск-менеджера, яка базується на Spring Framework та реалізує принципи REST, SOLID і MVC. Розроблена архітектура забезпечує високу модульність, безпеку, можливість масштабування та подальшого розширення системи відповідно до потреб користувачів.

Практичне значення полягає у створенні серверного застосунку, який може використовуватись як окремий продукт або як основа для розробки корпоративних систем управління процесами. Розроблений API має легко інтегруватись із веб та мобільними інтерфейсами, що зробить його універсальним інструментом для подальшого розвитку.

Апробація результатів дослідження. Основні результати роботи та архітектурні рішення були представлені на Всеукраїнській науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: Дослідження, проблеми, перспективи (МН-2025)» [1] та «Молодь в науці: Дослідження, проблеми, перспективи (МН-2026)» [2].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Поняття та класифікація систем управління завданнями

Системи управління завданнями (англ. Task Management Systems, TMS) є невід’ємною складовою сучасних інформаційних технологій і відіграють ключову роль у підтримці ефективної організації робочих процесів у різних сферах діяльності. У контексті зростання складності бізнес-процесів, збільшення кількості одночасних проєктів та необхідності координації великої кількості учасників саме TMS забезпечують структуроване планування, моніторинг і контроль виконання задач. Вони дозволяють формалізувати взаємодію між членами команди, забезпечують швидкий доступ до актуальної інформації, підвищують прозорість робочих процесів і мінімізують ризики, пов’язані з неповнотою або втратою даних.

Сучасні системи управління завданнями суттєво розширили свій функціонал порівняно з ранніми інструментами, які виконували лише роль електронних списків. TMS інтегруються з інструментами для спільної роботи, аналітичними модулями, системами звітності та календарним плануванням, що робить їх універсальним інструментом для організації діяльності як окремих працівників, так і великих команд. Вони дають змогу оптимізувати розподіл ресурсів, враховувати залежності між задачами, забезпечувати автоматичні нагадування, відстежувати зміни в режимі реального часу та формувати історію взаємодії користувачів.

Особливої актуальності TMS набувають у сфері управління проєктами, де необхідно контролювати виконання великої кількості взаємопов’язаних задач, відслідковувати прогрес та забезпечувати досягнення стратегічних цілей у встановлені терміни. У корпоративних рішеннях вони виступають ключовим інструментом координації між відділами та забезпечують узгодженість дій між різними учасниками бізнес-процесів. У персональних органайзерах та додатках

для самоменеджменту такі системи допомагають структурувати повсякденні завдання, підвищувати особисту продуктивність та покращувати тайм-менеджмент користувачів.

Завдяки розвитку хмарних технологій, мобільних платформ та інтеграційних механізмів (API, вебхуки, синхронізація між пристроями) сучасні системи управління завданнями забезпечують безперервний доступ до даних у будь-якому місці та в будь-який час. Це робить їх важливим компонентом цифрової інфраструктури організацій та індивідуальних користувачів, сприяючи побудові більш ефективних, прозорих та керованих процесів роботи [3].

Поняття управління завданнями охоплює комплекс процесів, які забезпечують повний життєвий цикл роботи із завданнями – від моменту їхнього формулювання до завершення та оцінки результатів. До таких процесів належать планування, постановка, розподіл, відстеження, коригування та аналіз виконання завдань. У межах сучасних інформаційних систем управління завданнями ці процеси реалізуються у вигляді чітко формалізованих процедур, що дозволяє мінімізувати ризики помилок, втрати інформації та неефективного використання часу.

Система управління завданнями у широкому розумінні є поєднанням методів організації роботи, програмних засобів, алгоритмів та регламентів взаємодії між учасниками робочого процесу. Вона забезпечує структуроване представлення завдань, визначення їхніх атрибутів (пріоритет, виконавець, термін виконання, статус), а також створення умов для контролю виконання та підтримки зворотного зв’язку. Такі системи дають змогу координувати діяльність у командах будь-якого масштабу, забезпечуючи прозорість і відтворюваність процесів, що є особливо важливим у межах великих проєктів з високим рівнем взаємозалежності між задачами.

Застосування систем управління завданнями дозволяє оптимізувати розподіл робочих ресурсів, автоматизувати рутинні операції та швидко реагувати на зміни у робочому середовищі. Крім того, сучасні TMS включають інструменти аналітики, що дають можливість оцінювати ефективність

виконання завдань, виявляти проблемні ділянки, прогнозувати можливі затримки та формувати рекомендації на основі історичних даних. Це сприяє не лише підвищенню продуктивності окремих співробітників, а й загальній оптимізації діяльності організації.

Основна мета таких систем полягає у зменшенні часових витрат на координацію діяльності, покращенні якості комунікацій та підвищенні рівня відповідальності виконавців. Завдяки структурованому підходу та наявності інструментів контролю система управління завданнями забезпечує високу якість виконання завдань, точність дотримання термінів та відповідність результатів очікуванням. Таким чином, TMS виступають ключовим елементом цифрової організації праці та фундаментальною складовою сучасних інформаційних систем [4].

Класифікація систем управління завданнями може здійснюватися за різними критеріями, що відображають особливості їх структури, функціональності та призначення. Один із базових підходів передбачає поділ за рівнем масштабності, характером взаємодії користувачів та сферою застосування. Такий поділ дозволяє не лише охарактеризувати функціональні можливості систем, а й визначити оптимальні сценарії їх використання в залежності від потреб окремих осіб, робочих груп чи великих організацій.

Першу групу становлять індивідуальні системи управління завданнями, такі як Todoist та Microsoft To Do. Ці інструменти спрямовані на підтримку персональної продуктивності, організацію щоденної діяльності, управління невеликими списками справ та забезпечення нагадувань. Вони характеризуються простим інтерфейсом, мінімалістичним набором функцій і високою зручністю використання. Основна увага приділяється швидкості взаємодії з інтерфейсом та оптимальному представленню інформації, що дозволяє користувачеві легко структурувати власні обов’язки без надмірного функціонального навантаження.

Другу групу формують командні системи управління завданнями, до яких належать Asana, Trello, ClickUp та інші. Такі рішення орієнтовані на колективну взаємодію, підтримують розподіл відповідальності між учасниками, дозволяють

відстежувати виконання завдань у реальному часі та забезпечують спільну роботу над проєктами різної складності. Їх ключовою характеристикою є можливість відтворення робочих процесів у візуальному форматі (дошки, списки, діаграми Ганта) та інтеграція комунікаційних механізмів, що сприяє оперативному обміну інформацією. Командні системи забезпечують баланс між простотою використання та функціональністю, роблячи їх універсальними для малих і середніх команд.

Найбільш масштабний рівень становлять корпоративні системи, такі як Jira, Monday чи Wrike, які призначені для великих організацій із багаторівневою структурою управління та комплексними бізнес-процесами. Вони підтримують інтеграцію з ключовими корпоративними платформами, серед яких CRM, ERP, DevOps-інструменти, що дозволяє автоматизувати міжвідділові процеси та забезпечити повну прозорість реалізації проєктів. Такі системи мають розширену аналітику, механізми контролю якості, модулі планування ресурсів та широкі можливості кастомізації, що робить їх незамінними у великих IT-компаніях, виробничих підприємствах та організаціях зі складною структурою взаємодії.

Таким чином, класифікація систем управління завданнями залежно від масштабності та сфери застосування дозволяє обрати рішення, що найбільш точно відповідає потребам користувачів. Індивідуальні системи забезпечують персональну організацію діяльності, командні – підтримують ефективну взаємодію групи, а корпоративні – реалізують комплексні сценарії управління на рівні підприємства, забезпечуючи інтеграцію з іншими інформаційними системами та підтримку стратегічного планування [5].

Іншим важливим критерієм класифікації систем управління завданнями є ступінь автоматизації та інтелектуалізації процесів, які вони підтримують. Рівень автоматизації визначає, наскільки система здатна самостійно виконувати рутинні операції, зменшуючи навантаження на користувача, тоді як інтелектуалізація відображає здатність системи аналізувати дані та приймати обґрунтовані рішення. Традиційні системи характеризуються мінімальною

автоматизацією: вони передбачають ручне створення завдань, їх редагування, встановлення термінів та розподіл відповідальності. У таких рішеннях користувач самостійно контролює етапи виконання задач, аналізує навантаження та визначає пріоритети, що може бути ефективним у невеликих командах або при простих сценаріях використання.

Однак із розвитком інформаційних технологій спостерігається чітка тенденція до впровадження інтелектуальних механізмів у системи управління завданнями. Сучасні платформи інтегрують алгоритми машинного навчання, системи прогнозної аналітики та інструменти автоматичного прийняття рішень. Такі системи здатні ідентифікувати закономірності у даних, аналізувати виконані раніше задачі, тривалість робочих процесів та ефективність учасників, формуючи рекомендації щодо оптимізації роботи. Наприклад, інтелектуальні рішення можуть автоматично визначати пріоритети завдань, прогнозувати ризики затримок, пропонувати перерозподіл ресурсів або коригувати робочі плани відповідно до виявлених тенденцій.

Крім того, системи з високим рівнем інтелектуалізації здатні адаптуватися до поведінки користувачів та специфіки проєктів. У таких рішеннях використовуються механізми персоналізації, що враховують стиль роботи окремих співробітників, їх продуктивність, типові патерни виконання задач та реакцію на навантаження. Це дозволяє формувати індивідуальні рекомендації, автоматично оптимізувати робочі графіки та сприяти покращенню загальної ефективності команди. В окремих системах навіть передбачено прогнозування потенційних проблем у проєкті, наприклад, надмірної кількості завдань у певного виконавця або зниження швидкості виконання задач.

Завдяки таким можливостям сучасні інтелектуалізовані системи управління завданнями виходять за межі традиційного інструментарію, перетворюючись на комплексні аналітичні рішення, що підтримують прийняття управлінських рішень. Вони не лише автоматизують виконання рутинних операцій, але й формують рекомендації, які сприяють стратегічному плануванню, підвищують якість управління та зменшують операційні ризики.

Таким чином, ступінь автоматизації та інтелектуалізації стає ключовим фактором відмінності між поколіннями систем управління завданнями, визначаючи їх потенціал до розвитку та застосування у складних багатокористувацьких середовищах [6].

Системи управління завданнями можна визначити як комплексні інформаційні рішення, що поєднують широке коло інструментів для планування, моніторингу, комунікації та аналітичної підтримки організаційної діяльності. Вони забезпечують структурований підхід до організації робочих процесів, дозволяють формалізувати послідовність виконання завдань, визначити їх взаємозв’язки, відповідальних осіб та пріоритети. Завдяки використанню таких систем підприємства та команди отримують можливість централізовано управляти інформаційними потоками, своєчасно реагувати на зміни у проєктах та підтримувати високий рівень дисципліни виконання.

Важливу роль системи управління завданнями відіграють у підвищенні продуктивності командної роботи. Вони сприяють оптимізації взаємодії між учасниками проєкту, забезпечуючи прозорість процесів та можливість фіксації результатів на кожному етапі. Чітке визначення відповідальних осіб та автоматизоване відстеження прогресу дозволяють уникати дублювання роботи, зменшувати кількість помилок та своєчасно виявляти потенційні затримки. Крім того, інформаційні системи такого типу створюють єдиний простір для комунікації та обміну даними, що особливо важливо для розподілених команд та організацій з великою кількістю паралельних проєктів.

З погляду економічної ефективності системи управління завданнями сприяють зниженню операційних витрат, оскільки автоматизують виконання рутинних процесів, зменшують потребу у ручному контролі та скорочують час, необхідний на координацію дій між учасниками команд. Систематизоване представлення даних дозволяє керівникам оперативно аналізувати завантаженість персоналу, приймати обґрунтовані управлінські рішення та оптимізувати розподіл ресурсів. Аналітичні модулі, інтегровані в сучасні інформаційні платформи, надають можливість прогнозувати ризики, оцінювати

ефективність виконання задач та виявляти «вузькі місця» у робочих процесах, що сприяє покращенню стратегічного планування й підвищенню конкурентоспроможності організацій.

Крім того, класифікація систем управління завданнями за рівнем застосування, функціональністю та ступенем автоматизації дозволяє ефективно обирати оптимальний інструмент відповідно до потреб користувачів і специфіки проєкту. Індивідуальні користувачі, невеликі команди та великі підприємства мають різні вимоги до масштабованості, доступності, інтеграції та рівня контролю над даними, тому правильний вибір системи суттєво впливає на результативність роботи. Зважаючи на швидке зростання складності сучасних бізнес-процесів, системи управління завданнями стають фундаментальним елементом інформаційної інфраструктури підприємств, забезпечуючи основу для цифрової трансформації та стійкого розвитку.

1.2 Аналіз існуючих рішень

Ринок систем управління завданнями сьогодні характеризується високим рівнем конкуренції та різноманітністю функціональних можливостей. Існуючі рішення відрізняються підходами до організації робочих процесів, архітектурою, рівнем інтеграції з іншими сервісами та орієнтацією на певний тип користувачів. Основними критеріями оцінювання таких систем є зручність інтерфейсу, гнучкість у налаштуванні робочих процесів, можливість інтеграції з іншими інструментами, а також продуктивність і безпека [7].

Одним із найпоширеніших рішень є **Trello** (рис. 2.1) – система управління завданнями, що базується на візуальній концепції Kanban-дошок. Її ключовою перевагою є інтуїтивно зрозумілий інтерфейс і простота у використанні. Кожне завдання представлено у вигляді картки, яку можна переміщати між стовпцями відповідно до етапів виконання. Trello активно використовується як для

особистого, так і для командного планування. Серед недоліків відзначають обмежені можливості у сфері аналітики, відсутність розгалуженої системи звітності та складність масштабування для великих проєктів [8].

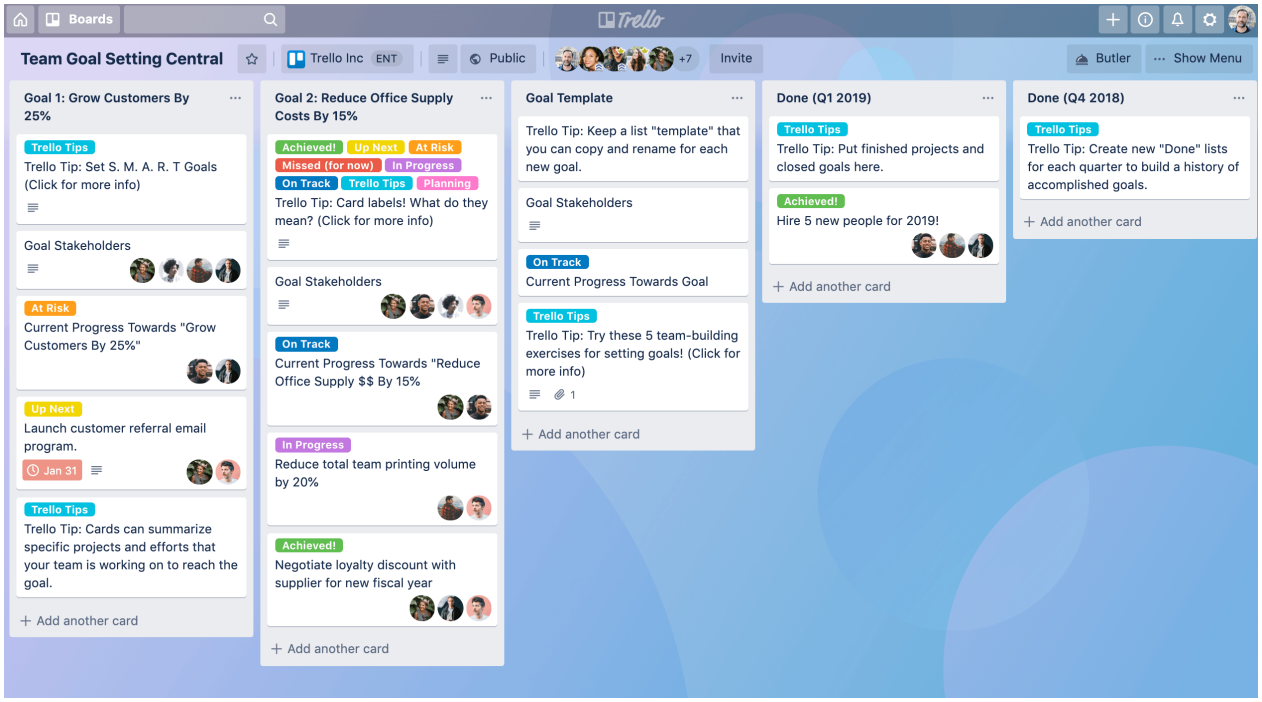

Рисунок 1.1 – Інтерфейс Trello

Інше популярне рішення – **Asana** (рис. 2.2), яке пропонує гнучкий підхід до управління завданнями у форматі списків, дошок або часових ліній. Asana орієнтована на корпоративне використання та підтримує інтеграцію з численними сторонніми сервісами, такими як Slack, Google Workspace чи Jira. Основними перевагами є підтримка розподілу ролей, автоматизація повторюваних завдань і розширена система звітів. Разом із тим, Asana вимагає певного навчання користувачів, оскільки система має складну структуру налаштувань [9].

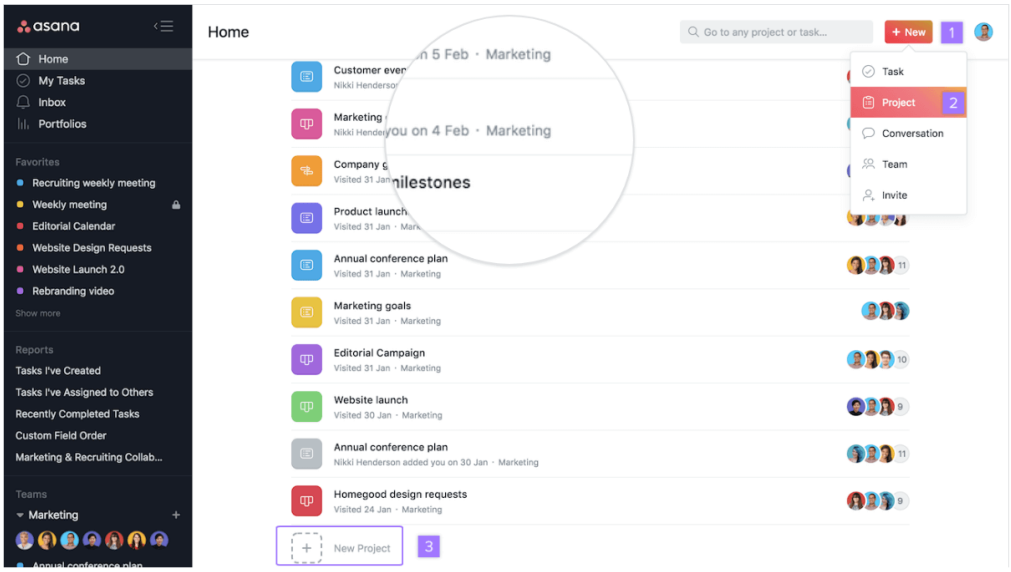

Рисунок 1.2 – Інтерфейс Asana

Серед корпоративних рішень особливе місце займає **Jira Software** (рис. 2.3), розроблена компанією Atlassian. Цей інструмент є стандартом де-факто для управління завданнями у сфері розробки програмного забезпечення. Jira підтримує методології Agile та Scrum, забезпечує потужні можливості для моніторингу прогресу, формування звітності та управління спринтами. Основною перевагою є гнучка система конфігурацій та масштабованість, однак серед недоліків зазначається складність інтерфейсу для нових користувачів і висока вартість ліцензії для великих команд [10].

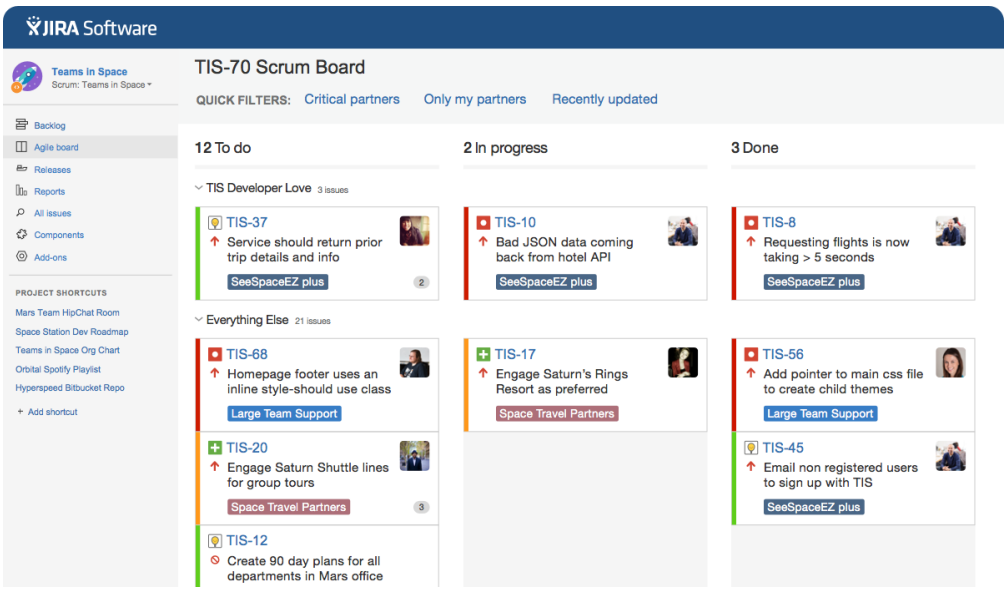

Рисунок 1.3 – Інтерфейс Jira Software

Ще одним прикладом є **ClickUp** (рис. 2.4), який поєднує функціональність управління завданнями, документами, календарями та аналітикою. ClickUp відзначається високим рівнем кастомізації, що дозволяє адаптувати систему до потреб конкретного бізнесу. Платформа активно використовує автоматизацію, включаючи створення тригерів для повторюваних дій, і надає можливість інтеграції з понад 100 сервісами. За результатами дослідження [11], ClickUp демонструє одне з найкращих співвідношень між функціональністю та ціною, однак має дещо перевантажений інтерфейс для новачків.

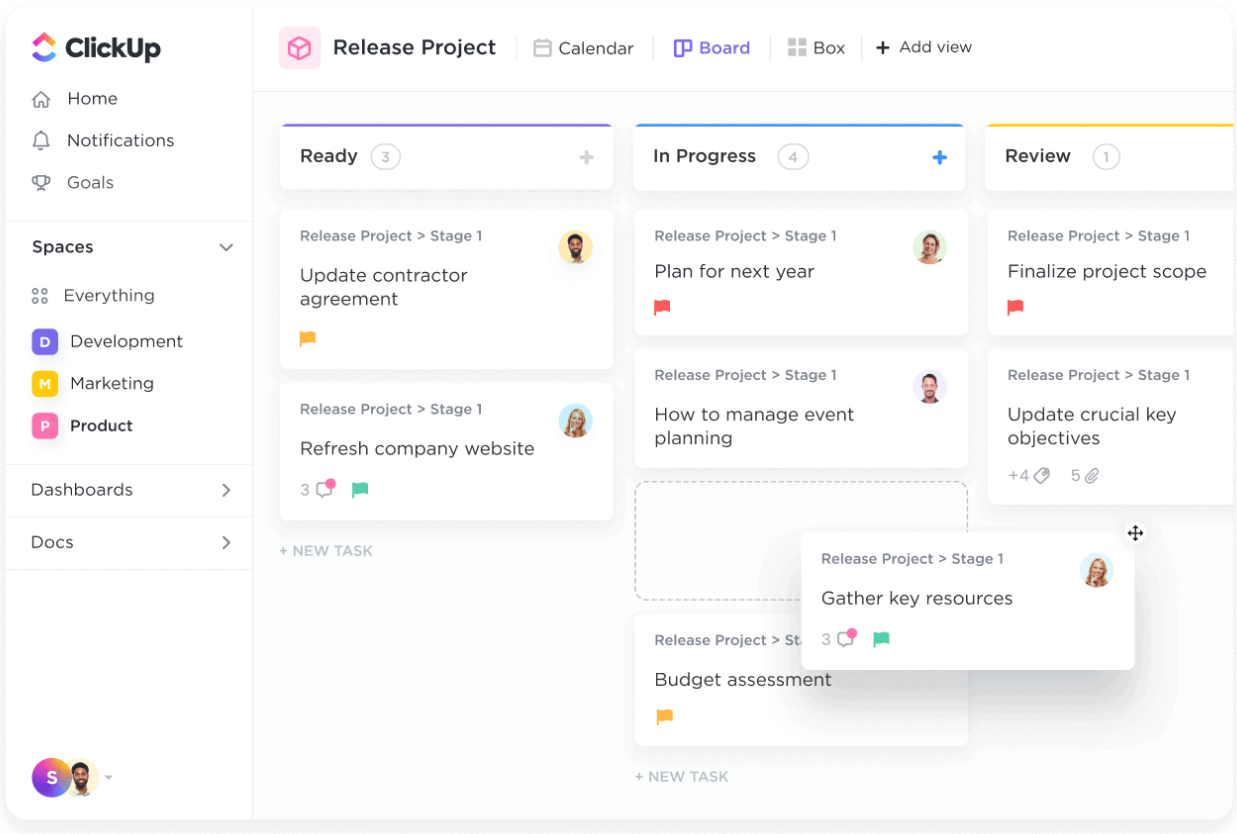

Рисунок 1.4 – Інтерфейс ClickUp

Аналіз існуючих систем показує, що ринок управління завданнями динамічно розвивається у напрямку розширення можливостей автоматизації, інтеграції та аналітики. Водночас спостерігається тенденція до спрощення інтерфейсів і збільшення мобільності користувачів. Проте більшість рішень орієнтовані на певні сценарії використання, що ускладнює універсальне

застосування. Це підтверджує доцільність створення власного API-рішення, яке забезпечить гнучкість, розширюваність і простоту інтеграції в різні бізнес-процеси. Порівняльний аналіз систем представлено в таблиці 1.1

Таблиця 1.1 – Порівняльна характеристика систем управління завданнями

Критерій	Trello	Asana	Jira	ClickUp	API-рішення
Гнучкість та кастомізація	2	4	5	5	5
Можливість розширення	3	5	5	5	5
Простота використання	5	3	2	3	3
Масштабова-ність, проду-ктивність	2	4	5	4	5
Безпека	2	3	5	3	5
Сума	**14**	**19**	**22**	**20**	**23**

Порівняльний аналіз за бальною системою демонструє, що найбільшу кількість балів отримує розроблене API-рішення – 23 з 25 можливих. Найвищі оцінки система отримує за масштабованість, безпеку, інтеграцію та гнучкість, можливістю повної кастомізації бізнес-логіки. На відміну від готових комерційних продуктів, API не обмежений за сценаріями використання та може бути адаптований під будь-який тип проєкту або корпоративного середовища.

Рівень безпеки більшості популярних систем управління завданнями обмежений їхньою орієнтацією на масового користувача: Trello, Asana та ClickUp забезпечують лише базові механізми захисту та не допускають глибокої кастомізації моделі доступу. На цьому тлі розроблене API демонструє вищий рівень безпеки завдяки використанню JWT-аутентифікації, ролей доступу, BCrypt-хешування та валідації даних, що забезпечує гнучкість і надійність, необхідні для корпоративних систем.

Серед популярних систем найближчим конкурентом є Jira Software, яка має високі показники продуктивності та інтеграцій, але складніша у використанні та має високу вартість. Інші сервіси, такі як Trello чи Asana, орієнтовані на зручність і візуальність, але програють у масштабованості й безпеці. Таким чином, власне API-рішення поєднує сильні сторони відомих продуктів та забезпечує максимальну гнучкість при мінімальних обмеженнях.

1.3 Порівняння підходів до створення API для управління завданнями

Створення API (Application Programming Interface) для систем управління завданнями є ключовим аспектом, що визначає ефективність інтеграції, масштабованість і взаємодію між різними компонентами системи. API виступає посередником між клієнтськими додатками, базою даних та сервісною логікою, забезпечуючи стандартизований спосіб обміну даними. Вибір архітектурного підходу до побудови API безпосередньо впливає на продуктивність, гнучкість і рівень безпеки програмного продукту [12].

Найбільш поширеним підходом у розробці API є архітектура **REST (Representational State Transfer)**. REST базується на протоколі HTTP та використовує стандартні методи – GET, POST, PUT, DELETE – для взаємодії з ресурсами системи. Основною перевагою REST є простота реалізації, кросплатформність та легкість інтеграції з веб- та мобільними додатками. REST API забезпечує масштабованість і незалежність клієнтської частини від серверної, що особливо важливо у розподілених системах. За даними дослідження [13], понад 80% сучасних корпоративних додатків у сфері управління завданнями використовують REST через його гнучкість і високу сумісність із фронтенд-фреймворками, такими як React, Angular та Vue.

Іншим підходом є використання **GraphQL**, розробленого компанією Facebook. На відміну від REST, GraphQL дозволяє клієнту самостійно визначати,

які саме дані необхідно отримати, що зменшує обсяг переданих даних і підвищує ефективність запитів. GraphQL має єдину кінцеву точку для всіх операцій і забезпечує точнішу взаємодію між клієнтом та сервером. Однак складність розробки та необхідність додаткового рівня валідації роблять цей підхід менш придатним для невеликих систем. За результатами дослідження [14], GraphQL демонструє переваги в продуктивності при роботі з великими наборами даних, однак потребує ретельного проєктування схеми, що збільшує початкові витрати часу на реалізацію.

Також значного поширення набувають підходи, засновані на **gRPC (Google Remote Procedure Call)**. Цей підхід базується на протоколі HTTP/2 і використовує формат обміну даними Protocol Buffers (protobuf), що дозволяє досягати високої швидкодії та зменшення навантаження на мережу. gRPC ідеально підходить для мікросервісної архітектури, де важлива ефективна взаємодія між внутрішніми сервісами системи. Проте через складність налаштування та обмежену підтримку браузерами, цей підхід рідше використовується у публічних веб-додатках [15].

У контексті систем управління завданнями, REST залишається найпопулярнішим стандартом завдяки своїй простоті, широкій підтримці та легкості документування через інструменти, такі як Swagger або OpenAPI. GraphQL найчастіше застосовується у складних багаторівневих додатках, де користувач потребує отримання даних із різних джерел в одному запиті. Натомість gRPC доцільно використовувати у високонавантажених корпоративних рішеннях із великою кількістю внутрішніх сервісів, що взаємодіють між собою з мінімальними затримками.

Вибір архітектури API залежить від вимог до продуктивності, масштабу системи, частоти оновлення даних і типу клієнтів. Для системи управління завданнями, орієнтованої на веб-доступ і просту інтеграцію, найбільш раціональним рішенням є використання REST API, який забезпечує баланс між продуктивністю, простотою реалізації та гнучкістю масштабування.

1.4 Обґрунтування вибору технологій та інструментів

Розробка API для системи управління завданнями потребує ретельного та обґрунтованого вибору технологічного стеку, оскільки саме він визначає надійність, продуктивність, безпеку та перспективи масштабування створеного рішення. У сучасних умовах розробники висувають високі вимоги до серверних систем, зокрема можливість обробляти значну кількість запитів, гнучко адаптувати бізнес-логіку, забезпечувати швидку інтеграцію з клієнтськими застосунками та сторонніми сервісами. Враховуючи ці фактори, у межах даної роботи було обрано три ключові технологічні компоненти – Java, Spring Framework та REST API. Java виступає базовою мовою програмування, яка забезпечує високу стабільність, портативність та підтримку багатопотоковості, що є критично важливим для серверних застосунків корпоративного рівня. Spring Framework доповнює можливості Java, надаючи потужний інструментарій для створення модульної архітектури, інверсії керування (IoC), контексту залежностей, а також вбудованих механізмів безпеки та взаємодії з базами даних. REST API, у свою чергу, забезпечує стандартизований і легковаговий спосіб комунікації між клієнтом і сервером, підтримуючи ідеологію безстановості, уніфікованого інтерфейсу та ресурсно-орієнтованої моделі. Поєднання цих технологій формує архітектуру, що відповідає сучасним вимогам до корпоративних вебсистем, забезпечує простоту супроводу та високу готовність до розширення функціоналу в майбутньому.

Мова програмування Java була обрана як основа для реалізації серверної частини проєкту завдяки своїй універсальності, стабільності та високому рівню безпеки, що є критично важливим для систем, орієнтованих на багатокористувацьку взаємодію та зберігання даних. Java є однією з найбільш зрілих мов у корпоративному секторі, що підтверджується широким використанням у фінансових, телекомунікаційних та банківських системах, де вимоги до надійності та відмовостійкості надзвичайно високі. Мова забезпечує

повноцінну підтримку багатопоточності, дозволяючи одночасно обробляти десятки тисяч паралельних запитів без втрати продуктивності, що є необхідним для ефективної роботи REST API у реальному часі. Розвинена система керування пам’яттю, зокрема автоматичне очищення через Garbage Collector, мінімізує людський фактор і знижує ймовірність критичних помилок, таких як memory leak чи некоректне звільнення ресурсів.

Ще однією ключовою перевагою Java є її платформонезалежність: застосунки, розроблені під JVM, можуть бути запущені на будь-якій операційній системі без зміни коду, що спрощує розгортання, міграцію та підтримку системи. Мова також має одну з найпотужніших екосистем бібліотек і фреймворків, які дозволяють розробляти масштабовані, модульні та безпечні системи з мінімальними зусиллями. Постійний розвиток стандартів, регулярні оновлення та активна підтримка спільноти забезпечують Java актуальність і сумісність із сучасними вимогами індустрії. Саме тому Java залишається домінуючою мовою для створення корпоративних API-рішень та є оптимальним вибором для реалізації системи управління завданнями в межах даної магістерської роботи [16].

Основою серверної частини розробленого API є Spring Framework – один із найпотужніших і найбільш поширених фреймворків для Java, який забезпечує гнучкий, масштабований та структурований підхід до створення веборієнтованих застосунків. Його архітектура побудована на принципі інверсії керування (IoC) та механізмі впровадження залежностей (Dependency Injection), що дозволяє значно зменшити зв’язність компонентів і спростити керування їхнім життєвим циклом. Завдяки модульній структурі Spring підтримує чітке розділення системи на ключові логічні рівні – контролери, сервіси, репозиторії, конфігураційні модулі та засоби безпеки. Це підвищує зрозумілість коду, прискорює розробку та робить проєкт більш придатним до масштабування та модифікацій.

Важливою складовою є Spring Boot – розширення Spring Framework, що автоматизує більшість конфігураційних процесів і надає готові до використання

шаблони, завдяки яким створення REST API стає максимально швидким та передбачуваним. Spring Boot пропонує механізми автоконфігурації, інтегрує вбудований сервер (Tomcat, Jetty або Undertow) та забезпечує стандартизовану структуру проєкту, що мінімізує витрати часу на початкове налаштування інфраструктури. Використання Spring Boot дозволяє реалізувати трирівневу архітектуру (Controller – Service – Repository), яка є промисловим стандартом у розробці корпоративних застосунків та забезпечує чітке розмежування відповідальності між частинами системи.

Крім того, фреймворк має розвинені засоби підтримки безпеки (Spring Security), взаємодії з базами даних (Spring Data JPA), обробки HTTP-запитів (Spring MVC) та модульного тестування (Spring Test). Завдяки цьому забезпечується висока стабільність, легкість у тестуванні та можливість швидкого розширення функціоналу. Саме поєднання багатошарової архітектури, IoC, автоматизації конфігурацій і широкої екосистеми робить Spring Framework оптимальним вибором для реалізації серверної частини системи управління завданнями [1].

Архітектурною основою створеного API є REST (Representational State Transfer) – поширений стиль побудови вебсервісів, який ґрунтується на стандартах протоколу HTTP та принципах ресурсноорієнтованої взаємодії. REST визначає, що всі сутності доменної області (наприклад, користувачі, завдання, списки To-Do) мають представлятися у вигляді ресурсів, доступних за унікальними URI, такими як /api/users, /api/todos або /api/tasks. Для керування цими ресурсами REST передбачає використання уніфікованого набору методів: GET для отримання даних, POST для створення, PUT або PATCH для оновлення та DELETE для видалення. Це забезпечує логічну, передбачувану та стандартизовану модель взаємодії, що істотно спрощує інтеграцію сторонніх клієнтів і зменшує ймовірність помилок у роботі з API.

Однією з ключових ознак REST є безстановість (stateless), що означає – сервер не зберігає інформацію про стан клієнта між запитами. Усі необхідні дані передаються в кожному запиті, що значно полегшує масштабування системи,

дозволяючи обробляти більшу кількість одночасних клієнтів без необхідності підтримувати серверні сесії. Такий підхід є критично важливим для систем управління завданнями, де велика кількість користувачів може надсилати паралельні запити з різних пристроїв або клієнтських платформ.

REST також визначає стандартизований формат обміну даними – найчастіше JSON, який є легким, зручним для читання та сумісним із більшістю мов програмування. Завдяки цьому клієнтська частина (вебінтерфейс, мобільний застосунок чи зовнішня інтеграція) може швидко взаємодіяти з сервером без додаткових перетворень. Крім того, REST-сервіси легко документувати за допомогою OpenAPI/Swagger, що спрощує підтримку та розвиток API.

Важливою перевагою REST є його здатність підтримувати різні типи клієнтів одночасно. У контексті таск-менеджера це дозволяє будувати як класичний вебінтерфейс, так і мобільні додатки, а також надавати можливість інтеграції третьої сторони – наприклад, чат-ботів або корпоративних систем. Саме така незалежність клієнтської та серверної частин формує гнучку архітектуру, здатну розширюватися без суттєвих змін базової платформи [17]

Важливою складовою обраної архітектури є впровадження механізму безпеки на основі JWT (JSON Web Token), який сьогодні вважається одним із найбільш оптимальних та ефективних способів автентифікації для REST-орієнтованих систем. На відміну від традиційного підходу, який передбачає зберігання серверних сесій, JWT працює за принципом повної безстановості – після успішного входу сервер генерує токен, який містить цифровий підпис та набір закодованих даних про користувача. Це можуть бути його унікальний ідентифікатор, роль у системі, час створення і термін дії токена. Завдяки криптографічному підпису сервер може перевірити токен без звернення до бази даних, що значно скорочує час обробки запитів і підвищує продуктивність системи.

JWT-архітектура також забезпечує високий рівень безпеки, оскільки токен неможливо підробити без секретного ключа, а дані всередині нього можуть бути як підписаними (JWS), так і зашифрованими (JWE). Це дозволяє визначати точні

права доступу користувача та забезпечує суворе розмежування ресурсів між різними ролями – наприклад, адміністратором та звичайним користувачем. Крім того, обмежений час дії токена мінімізує ризики, пов’язані з його викраденням, а можливість відкликання або ротації ключів дає змогу підвищити рівень контролю за автентифікаційними процесами.

Однією з головних переваг JWT є його незалежність від серверної інфраструктури: усі сервери, що обробляють запити, можуть перевіряти токени однаково, без спільного використання сесій чи кешу. Це робить підхід ідеальним для розподілених систем та мікросервісної архітектури, де важливо забезпечити масштабованість, рівномірний розподіл навантаження та відмовостійкість. У контексті серверної частини таск-менеджера такий механізм дає змогу гнучко керувати доступом до ресурсів, захищати персональні дані користувачів і спрощувати взаємодію клієнтських застосунків із сервером, гарантуючи при цьому безпеку та стабільність роботи API [1].

Отже, вибір стеку **Java, Spring Framework, REST API** обґрунтований вимогами до створення сучасного вебзастосунку, який має бути стабільним, безпечним, розширюваним і придатним до інтеграції. Ці технології забезпечують узгоджену архітектуру, підтримку стандартів і простоту подальшого супроводу системи, що є необхідними умовами для реалізації високоякісного API системи управління завданнями.

1.5 Постановка задачі розробки API

В умовах активного розвитку цифрових технологій, автоматизації бізнес-процесів і переходу організацій на дистанційні формати роботи, постає потреба у створенні ефективних систем управління завданнями. Такі системи повинні забезпечувати не лише зручне планування роботи, але й можливість командної взаємодії, контролю прогресу, аналітики продуктивності та інтеграції з іншими

інструментами управління проєктами. Враховуючи ці вимоги, постає завдання розробки серверної частини системи управління завданнями у вигляді REST API, яка забезпечить централізовану логіку, взаємодію між користувачами, збереження даних і безпечний доступ до інформації.

Основною метою розробки є створення REST API, який забезпечуватиме повноцінну взаємодію клієнтських додатків із базою даних та бізнес-логікою. Такий інтерфейс має дозволяти створення, редагування, призначення, моніторинг і завершення завдань, підтримувати автентифікацію користувачів, обмеження доступу за ролями та можливість масштабування. У контексті проєкту передбачено реалізацію архітектури, що базується на принципах REST і використовує багаторівневу структуру, де кожен шар системи виконує чітко визначену функцію [1].

Відповідно до поставленої мети, необхідно вирішити такі основні завдання:

– провести аналіз вимог до системи управління завданнями та визначити набір функціональних можливостей, необхідних для реалізації REST API;

– розробити архітектуру серверної частини застосунку із застосуванням принципів трирівневої моделі (Controller – Service – Repository), що забезпечує модульність і розділення відповідальностей;

– реалізувати REST API з використанням фреймворку Spring Boot, який забезпечить швидке налаштування середовища, обробку HTTP-запитів і інтеграцію з базою даних;

– впровадити систему безпеки на основі JWT (JSON Web Token) для автентифікації та авторизації користувачів без збереження сесій на сервері;

– забезпечити роботу з реляційною базою даних PostgreSQL для ефективного зберігання інформації про користувачів, завдання, проєкти та коментарі [18];

– протестувати роботу API за допомогою PostgreSQL, перевіривши коректність обробки запитів, помилок і відповідей сервера.

Очікуваним результатом реалізації є створення повнофункціонального серверного API, який можна легко інтегрувати з різними клієнтськими інтерфейсами – веб, мобільними або десктопними. Застосування принципів REST і використання Spring Boot дасть змогу досягти високої продуктивності, розширюваності та безпеки системи. Також важливим є забезпечення можливості подальшого масштабування рішення – як вертикального (покращення продуктивності окремих компонентів), так і горизонтального (розподіл навантаження між серверами).

Постановка задачі розробки API зводиться до створення надійної, безпечної та масштабованої серверної платформи, що реалізує повний цикл управління завданнями. Система має бути адаптивною до зміни функціональних вимог і придатною для подальшої інтеграції в корпоративну інфраструктуру підприємства.

1.6 Висновки до розділу

У першому розділі було проведено аналіз предметної області систем управління завданнями, визначено їх основні характеристики, класифікацію та сучасні тенденції розвитку. Проаналізовано наявні рішення, серед яких Jira, Trello, Asana, ClickUp, що дозволило сформувати розуміння ключових функціональних вимог до сучасних таск-менеджерів. Було виявлено, що більшість комерційних систем мають спільні риси – підтримку командної роботи, інтеграцію з іншими сервісами, можливість відстеження прогресу завдань і автоматизацію процесів. Водночас спостерігається проблема складності налаштування, високої вартості корпоративних ліцензій і недостатньої гнучкості інтерфейсів. Це підтвердило актуальність розробки власного API, який поєднує універсальність і простоту інтеграції.

Було досліджено й порівняно архітектурні підходи до побудови API, зокрема REST, GraphQL та gRPC. На основі цього аналізу обґрунтовано вибір REST як оптимальної архітектурної моделі для реалізації API системи управління завданнями. REST забезпечує зрозумілу ієрархію ресурсів, підтримує стандартні HTTP-методи, має високу сумісність із фронтенд-технологіями та спрощує масштабування. Це робить його доцільним вибором для побудови вебсервісів, що потребують стабільності, гнучкості та легкості інтеграції [1].

Важливою складовою розробки стало обґрунтування вибору технологій – **Java**, **Spring Framework** і **REST API**. Java забезпечує надійність, кросплатформність і підтримку великої кількості бібліотек, що прискорює розробку серверної частини. Spring Boot, у свою чергу, дає змогу швидко створювати прототипи, автоматизує конфігурацію та інтеграцію компонентів, забезпечуючи модульність і легку масштабованість системи. Поєднання цих технологій дозволяє створити трирівневу архітектуру з чітким розмежуванням логіки контролерів, сервісів і репозиторіїв, що позитивно впливає на підтримуваність і розширюваність проєкту [1].

Під час постановки задачі розробки API сформульовано основні вимоги до системи: реалізація CRUD-операцій (створення, читання, оновлення, видалення завдань), автентифікація та авторизація користувачів за допомогою JWT, обробка запитів у реальному часі, можливість інтеграції з іншими системами та адаптація до зростання навантаження. Запропонована архітектура забезпечує можливість поступового розширення функціоналу без порушення стабільності роботи системи.

Результати даного етапу дозволили сформувати концептуальні основи майбутньої реалізації API. Вибрані технології та архітектурні рішення забезпечують поєднання надійності, безпеки й гнучкості, необхідних для створення сучасної системи управління завданнями. Отримані висновки стануть базою для подальшого етапу – **проєктування API системи управління завданнями**, де будуть детально розроблені структура даних, архітектура компонентів і механізми безпеки.

2 ПРОЄКТУВАННЯ API ДЛЯ СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ

2.1 Вибір архітектури для системи управління завданнями

Для серверної частини таск-менеджера обрано REST-архітектуру, реалізовану на основі Spring Boot, що дозволяє побудувати гнучку, масштабовану та стандартизовану інфраструктуру взаємодії між клієнтом і сервером. REST виступає домінуючим підходом у сучасній веброзробці завдяки своїй простоті, ресурсно-орієнтованій моделі, використанню стандартних HTTP-методів (GET, POST, PUT, DELETE) та широкій підтримці з боку інструментарію і фреймворків. Застосування формату JSON як універсального способу передачі даних забезпечує сумісність з більшістю клієнтських застосунків, мобільних платформ та фронтенд-фреймворків. Таким чином, REST створює умови для прозорої, передбачуваної та ефективної комунікації в межах розподіленої системи.

У розробці серверної частини суттєву увагу приділено архітектурному поділу відповідно до шаблону MVC та принципів багатошаровості (Controller → Service → Repository). Кожен із шарів виконує чітко визначені функції: контролери обробляють HTTP-запити та відповіді, сервіси інкапсулюють бізнес-логіку, а репозиторії забезпечують взаємодію з базою даних через JPA. Такий підхід мінімізує зв’язність модулів, підвищує тестованість, забезпечує простоту модифікацій та полегшує впровадження нових функцій. Дотримання SOLID-принципів у бізнес-логіці гарантує стабільність і передбачуваність системи під час розвитку проєкту, що особливо важливо для довготривалих і масштабованих рішень.

Використання токенів, на відміну від серверних сесій, дозволяє забезпечити повну безстановість (stateless) сервера, що відповідає вимогам REST-архітектури та суттєво спрощує масштабування застосунку. Токени передаються в кожному запиті, не потребують збереження на стороні сервера,

забезпечують ізоляцію прав доступу й можуть бути легко інтегровані в мобільні та вебклієнти. Крім того, JWT створює умови для реалізації ролей і гнучких політик доступу, що є ключовим аспектом у системах з багатокористувацьким середовищем.

Spring Boot, який використано як основний технологічний стек, надає потужні інструменти для конфігурації, автоматизації та підвищення продуктивності розробки. Його вбудовані механізми autoconfiguration, підтримка стартових залежностей, інтеграція зі Spring Security та Spring Data JPA дозволяють суттєво скоротити час на налаштування та зосередитися на реалізації бізнес-функцій. Крім того, Spring Boot є природною платформою для побудови мікросервісів, що забезпечує легкий перехід до більш складної архітектури в майбутньому. У роботі підкреслено, що саме поєднання REST, JWT та шарової структури на Spring Boot є оптимальним для створення сучасних серверних рішень у домені систем управління завданнями [1].

REST (Representational State Transfer) є одним із найбільш поширених архітектурних стилів побудови веб-сервісів, що задає правила організації взаємодії між клієнтом і сервером у розподілених системах. Основою REST є представлення сутностей предметної області у вигляді ресурсів, кожен з яких має унікальний URI-ідентифікатор (наприклад, /api/tasks, /api/projects, /api/users). Доступ до ресурсів здійснюється за допомогою стандартних HTTP-методів, де кожен метод однозначно відповідає певній операції: GET використовується для отримання даних, POST – для створення нових об’єктів, PUT або PATCH – для оновлення існуючих ресурсів, DELETE – для їх видалення. Такий підхід забезпечує природну відповідність між операціями над даними та протоколом передачі, що сприяє узгодженості й передбачуваності поведінки API.

Одним із ключових принципів REST є безстановість (stateless): сервер не зберігає інформацію про попередні взаємодії з клієнтом, а кожен запит повинен містити всю необхідну інформацію для його обробки. Це значно спрощує масштабування системи, оскільки відсутня потреба синхронізувати сесійні дані між серверами, а також підвищується надійність і гнучкість архітектури. Таке

рішення є особливо актуальним у хмарних середовищах, де навантаження може динамічно змінюватися, а сервери – масштабуватися горизонтально.

Уніфікований інтерфейс REST визначає спільні правила представлення ресурсів, структуру відповідей, коди станів HTTP і формат переданих даних (найчастіше JSON). Це означає, що незалежно від складності системи чи доменної логіки, взаємодія клієнтів із сервером здійснюється стандартизованим способом. Уніфікація знижує поріг входу для розробників, спрощує тестування API за допомогою стандартних інструментів (Postman, cURL) і зменшує ризик появи непередбачуваних поведінкових сценаріїв.

Ще однією важливою властивістю REST є підтримка кешування – механізму, що дозволяє суттєво знизити навантаження на сервер і прискорити обробку повторюваних запитів. Кешування відповідає принципам оптимізації мережевих взаємодій і допомагає забезпечити кращу продуктивність при роботі з великою кількістю користувачів або ресурсів, що часто запитуються. Для систем управління завданнями, де значна частина операцій пов’язана з читанням даних (перегляд To-Do, списків задач, станів), це має особливу практичну цінність.

Дотримання принципів REST сприяє побудові логічної, модульної та передбачуваної структури API. Чітке визначення ресурсів і їх взаємодій спрощує документацію, супровід, рефакторинг та розширення функціональності в майбутньому. Завдяки REST-підходу клієнтські застосунки – вебінтерфейси, мобільні програми чи зовнішні інтеграційні сервіси – можуть взаємодіяти з сервером уніфікованим способом, що забезпечує широкі можливості для масштабування системи та підвищує її цінність у корпоративному середовищі [19].

MVC / багатошарова побудова. Усередині сервера приймаємо структурний поділ:

- **Controller** – приймає HTTP-запити, виконує валідацію вводу/виводу й мапінг DTO;

- **Service** – містить бізнес-правила (пріоритезація, зміна статусів, перевірка прав доступу на рівні домену);
- **Repository** – інкапсулює доступ до БД (через Spring Data JPA).

Такий поділ підвищує модульність і дає можливість незалежно змінювати представлення, логіку та зберігання даних, що прямо відзначено в тезі [1] як очікуваний ефект модульності та зручності розширення.

SOLID у доменній логіці.

- **S (Single Responsibility):** кожен сервіс відповідає за одну доменну підзадачу (наприклад, TaskService не займається автентифікацією чи нотифікаціями).
- **O (Open/Closed):** розширюємо поведінку через інтерфейси/стратегії (наприклад, різні політики призначення виконавців), не змінюючи існуючий код.
- **L (Liskov):** коректні підстановки інтерфейсів (наприклад, різні реалізації NotificationSender).
- **I (Interface Segregation):** дрібні інтерфейси замість «важких» універсальних (окремі порти для читання/запису).
- **D (Dependency Inversion):** сервіси залежать від абстракцій (портів/репозиторіїв), а не від конкретних реалізацій – це спрощує тестування й заміни компонентів [20].

Дотримання SOLID робить код передбачуваним і стійким до змін, що критично для API, яке розвивається.

Для наочності структури архітектури застосунку побудовано діаграму пакетів, що відображає основні компоненти системи та зв’язки між ними (рис. 2.1).

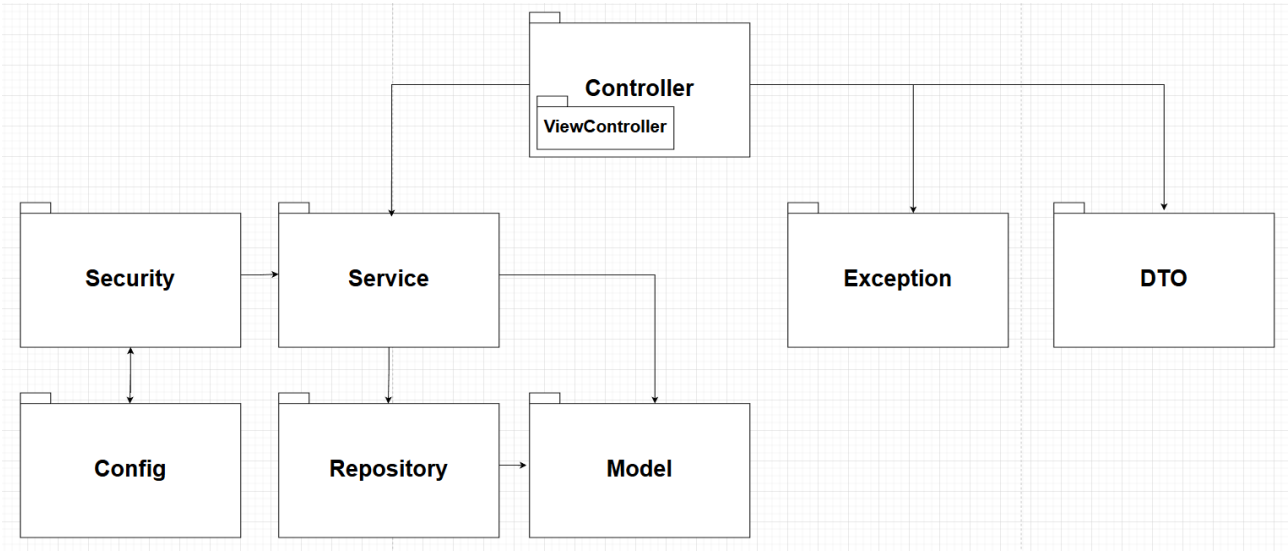

Рисунок 2.1 – Діаграма пакетів

На діаграмі пакетів відображено логічну структуру серверної частини системи управління завданнями, що побудована відповідно до принципів багатошарової архітектури та розділення відповідальності.

Пакет **controller** відповідає за обробку HTTP-запитів клієнтів та визначає REST-ендпоінти, через які зовнішні системи або фронтенд звертаються до API. Контролери не містять бізнес-логіки – їх завдання лише отримати запит, валідовано прийняти дані, передати їх до відповідного сервісу та повернути результат. Для передачі інформації між клієнтом і сервером використовуються об’єкти пакета **DTO**, що забезпечують безпечну та контрольовану серіалізацію/десеріалізацію даних. У разі виникнення помилок контролери можуть ініціювати винятки, які обробляються централізовано в пакеті **exception**.

Пакет **service** реалізує основну бізнес-логіку застосунку. Саме тут визначаються правила створення, редагування та видалення завдань і списків To-Do, а також механізми керування користувачами та їхніми правами. Сервіси взаємодіють із доменною моделлю, представленою в пакеті **model**, і отримують доступ до даних через пакет **repository**, який інкапсулює роботу з базою даних за допомогою Spring Data JPA.

Пакет **security** відповідає за автентифікацію та авторизацію користувачів. Тут реалізовано JWT-фільтри, механізм обробки токенів, завантаження

користувачів та перевірку ролей. Пакет тісно взаємодіє з **service**, оскільки використовує його для завантаження даних про користувачів під час входу в систему. Налаштування безпеки зосереджені у пакеті **config**, де визначаються правила доступу до ендпоінтів, конфігурація SecurityFilterChain, політика сесій та вмикаються/вимикаються необхідні механізми Spring Security.

Пакет **repository** абстрагує доступ до бази даних. Він містить інтерфейси, що автоматично реалізуються Spring Data JPA та дозволяють виконувати CRUD-операції над сутностями пакета **model**. У пакеті model розташовані доменні класи, які відображають структуру таблиць у PostgreSQL та визначають зв'язки між об'єктами – користувачами, завданнями, пріоритетами, станами та списками To-Do.

Уся система об'єднується головним класом додатка – **Application**, який виконує роль точки входу, ініціалізує Spring Context, завантажує всі конфігурації та формує повністю працездатний сервер. Така структура забезпечує модульність, розширюваність та легкість у тестуванні, що є критично важливим для сучасних REST-орієнтованих рішень.

Обрана архітектура – **REST + MVC (Controller–Service–Repository) на Spring Boot** із **SOLID-дисципліною** в домені та **JWT-авторизацією** – узгоджується з висновками та рекомендаціями власної наукової роботи і промисловою практикою. Це забезпечує прозорі контракти API, ізоляцію шарів, тестованість та готовність до подальшого масштабування і розширень (нові ендпоінти, ролі, інтеграції) [21-22].

2.2 Проєктування структури даних

Проєктування структури даних є одним із ключових етапів створення системи управління завданнями, адже саме від моделі бази даних залежить узгодженість інформації, швидкість доступу до неї та можливість подальшого

масштабування. Для розробки REST API таск-менеджера використано класичний підхід ER-моделювання, який дозволяє описати взаємозв'язки між основними сутностями: користувачем, проєктом (списком завдань), завданням, статусом і співавтором. Модель є реляційною, узгодженою з принципами третьої нормальної форми, що виключає дублювання даних і забезпечує цілісність зв'язків [23].

Основними сутностями системи є **users**, **todos**, **tasks**, **states** та **todo_collaborator**. Таблиця **users** містить інформацію про користувачів, включно з іменем, електронною адресою, паролем і роллю в системі. Кожен користувач може бути власником одного або декількох списків завдань, що зберігаються в таблиці **todos**, яка включає поля title, created_at, owner_id і первинний ключ id. Таблиця **tasks** відповідає за зберігання окремих завдань, які належать до певного списку (todo_id) і мають власний статус (state_id). Для статусів передбачено окрему таблицю **states**, у якій зберігаються такі значення, як "Відкрите", "У процесі" чи "Завершене".

Взаємодія між користувачами реалізується через таблицю **todo_collaborator**, яка створює зв'язок «багато-до-багатьох» між користувачами та списками завдань. Це дозволяє кільком користувачам спільно працювати над одним проєктом, водночас зберігаючи інформацію про власника кожного списку. Подібна структура забезпечує гнучкість у призначенні доступів, що є необхідною умовою для систем командної роботи.

Взаємозв'язки між сутностями виглядають таким чином: один користувач може володіти багатьма списками завдань; кожен список містить багато завдань; кожне завдання має один стан; а через проміжну таблицю todo_collaborator реалізується спільна участь користувачів у різних списках. Такий підхід дає змогу підтримувати як індивідуальну роботу з особистими завданнями, так і командну взаємодію.

Для забезпечення цілісності даних використано зовнішні ключі (FOREIGN KEY), що пов'язують між собою таблиці. Наприклад, todos.owner_id посилається на users.id, tasks.todo_id – на todos.id, а tasks.state_id – на states.id. Це дозволяє

зберігати логічну зв’язаність усіх об’єктів у системі. Крім того, в таблицях передбачено індекси для прискорення фільтрації за найчастіше використовуваними полями, такими як email, todo_id чи state_id.

На рисунку 2.2 подано **ER-діаграму**, що відображає повну структуру зв’язків між сутностями бази даних.

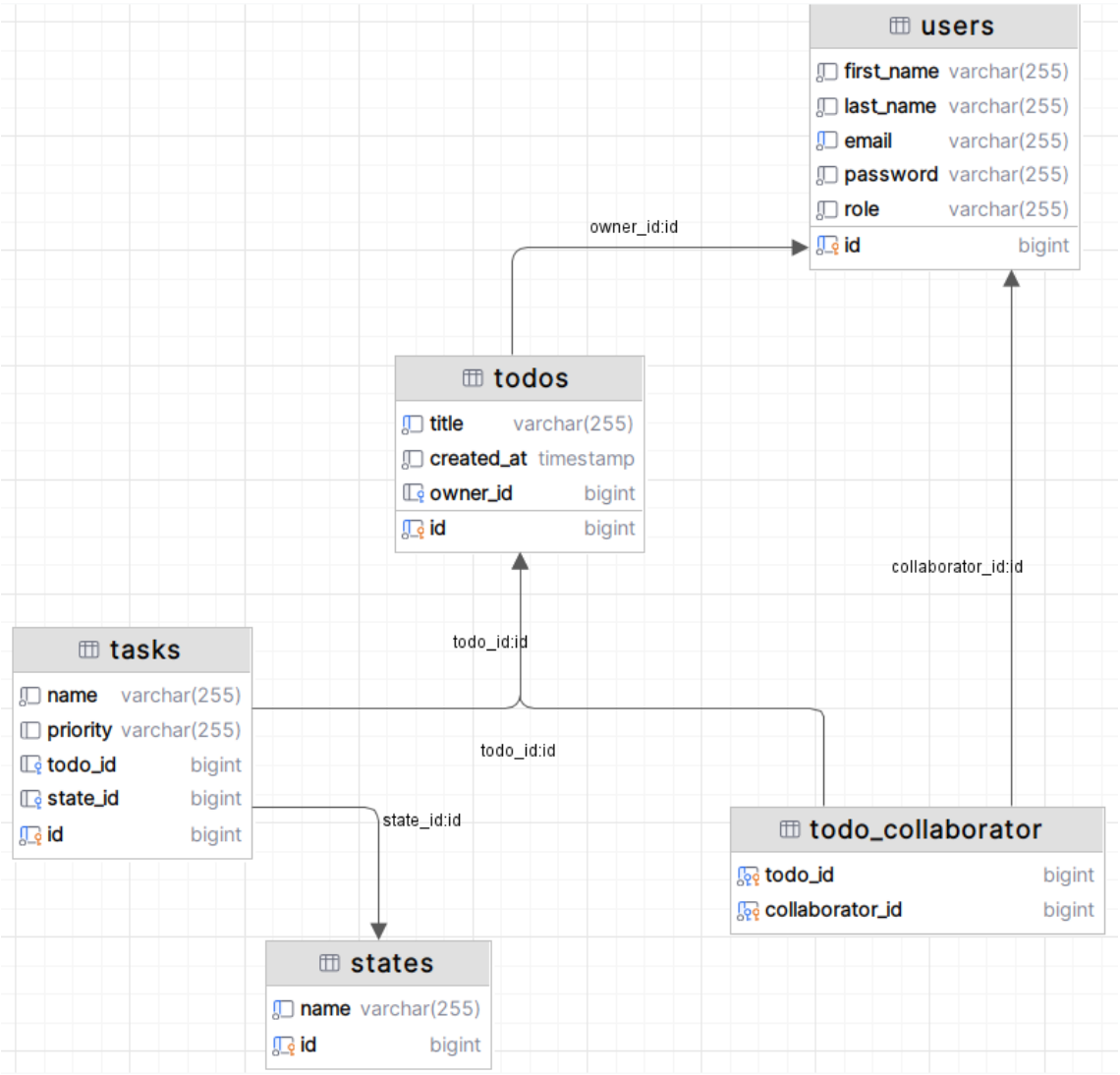

Рисунок 2.2 – ER-діаграма

Запропонована структура забезпечує розширюваність системи, дає можливість легко додавати нові сутності (наприклад, коментарі або теги) без суттєвих змін у базовій логіці. Завдяки логічному поділу таблиць за ролями та зв’язкам між ними забезпечується зручна реалізація CRUD-операцій через REST API, що повністю відповідає архітектурним принципам системи.

2.3 Специфікація API

REST API системи управління завданнями побудовано відповідно до принципів REST, описаних у розділі 2.1. Його головною метою є забезпечення зручного, стандартизованого та безпечного способу взаємодії між клієнтом і сервером. API розроблено на базі Spring Boot із використанням контролерів, які реалізують CRUD-операції (Create, Read, Update, Delete) для основних сутностей: користувачів, списків завдань, завдань, статусів і співробітників.

Кожен запит до сервера обробляється через HTTP-протокол і відповідає одному з основних методів: GET, POST, PUT, PATCH, DELETE. Обмін даними здійснюється у форматі JSON, що забезпечує сумісність із більшістю клієнтських застосунків, зокрема вебта мобільних інтерфейсів. Запити до API структуровано за логікою ресурсів, де кожна таблиця бази даних представлена як REST-ендпоінт.

Для зручності взаємодії всі ендпоінти мають префікс /api/, а доступ до захищених ресурсів здійснюється за допомогою JWT-токена, який передається в заголовку запиту Authorization: Bearer <token>.

Нижче наведено основні ендпоінти API та їхнє призначення.

Користувачі (/api/users)

- GET /api/users – отримання списку всіх користувачів (доступно лише адміністратору).
- GET /api/users/{id} – перегляд інформації про конкретного користувача.
- POST /api/users/register – реєстрація нового користувача з передачею імені, електронної пошти та пароля.
- POST /api/users/login – аутентифікація користувача й отримання JWT-токена.

Списки завдань (/api/todos)

- GET /api/todos – отримання всіх списків, створених користувачем або доступних йому як співробітнику.

- POST /api/todos – створення нового списку завдань (параметри: title, created_at).
- PUT /api/todos/{id} – оновлення назви списку або додавання співавторів.
- DELETE /api/todos/{id} – видалення списку разом із пов’язаними завданнями.

Завдання (/api/tasks)

- GET /api/tasks – перегляд усіх завдань користувача або за фільтром (наприклад, за статусом чи пріоритетом).
- POST /api/tasks – створення нового завдання (параметри: name, priority, todo_id, state_id).
- PUT /api/tasks/{id} – оновлення властивостей завдання (наприклад, зміна статусу або пріоритету).
- DELETE /api/tasks/{id} – видалення завдання.

Статуси (/api/states)

- GET /api/states – отримання довідника усіх статусів завдань.
- POST /api/states – створення нового статусу (наприклад, “Завершено”).
- DELETE /api/states/{id} – видалення статусу (за умови, що він не використовується у завданнях).

Співавтори (/api/collaborators)

- GET /api/todos/{id}/collaborators – перегляд усіх користувачів, які мають доступ до списку.
- POST /api/todos/{id}/collaborators – додавання нового співавтора за його user_id.
- DELETE /api/todos/{id}/collaborators/{user_id} – видалення користувача зі співавторів списку.

Для кожного запиту сервер повертає відповідь у форматі JSON. Наприклад, у разі успішного створення нового завдання API повертає код 201 Created і об’єкт виду:

{
 "id": 25,

"name": "Implement JWT authentication",
"priority": "HIGH",
"state": "In Progress",
"todo_id": 8,
"created_at": "2025-10-11T13:55:00Z"
}

У разі помилки клієнт отримує відповідь з кодом 400 Bad Request або 404 Not Found і повідомленням про причину, наприклад:

{
"error": "Todo not found",
"timestamp": "2025-10-11T14:01:00Z"
}

Такий підхід робить API зрозумілим і передбачуваним, що спрощує тестування через інструменти Postman або Swagger UI. Крім того, у Spring Boot можна інтегрувати **OpenAPI Specification** (Swagger) для автоматичної генерації документації, що полегшує підтримку системи й взаємодію між командами розробників [24].

Обрана специфікація ендпоінтів дозволяє гнучко керувати завданнями, користувачами, ролями та правами доступу, що є основою для побудови надійної системи управління завданнями на основі REST API.

2.4 Безпека API

Безпека є одним із найважливіших аспектів розробки серверної частини системи управління завданнями, адже саме від неї залежить захист персональних даних користувачів і цілісність інформації у базі даних. У межах цього проєкту реалізацію безпеки побудовано на основі **Spring Security** з використанням **JWT (JSON Web Token)** як механізму автентифікації та авторизації користувачів.

Такий підхід забезпечує відмову від традиційного зберігання сесій на сервері, що підвищує масштабованість і знижує навантаження на систему [1].

Spring Security виконує роль основного фреймворку контролю доступу в межах Spring Boot-застосунку. Він інтегрується з фільтрами HTTP-запитів, перехоплюючи звернення до захищених ресурсів і перевіряючи наявність коректного JWT-токена у заголовку Authorization. При успішній перевірці користувач вважається автентифікованим, і контекст безпеки (SecurityContext) зберігає інформацію про його роль – наприклад, ROLE_USER або ROLE_ADMIN. Для кожного контролера в системі визначено рівень доступу відповідно до ролі користувача.

JWT-токен формується під час входу користувача в систему через ендпоінт /api/users/login. Після успішної перевірки облікових даних сервер створює токен, який містить зашифровану інформацію про користувача: його ідентифікатор, електронну адресу, роль та час дії токена. Цей токен підписується за допомогою секретного ключа (алгоритмом HMAC SHA256 або RSA) і передається клієнту у відповіді. Далі клієнт додає його до кожного запиту у заголовок Authorization: Bearer <token>, що дає змогу серверу ідентифікувати користувача без повторного входу.

JWT складається з трьох частин: **Header**, **Payload** та **Signature**. Header визначає тип токена і алгоритм підпису, Payload містить основну інформацію (claims), а Signature гарантує цілісність даних. Приклад розшифрованого токена для користувача з роллю адміністратора може виглядати так:

```
{
  "sub": "admin@tasksystem.com",
  "role": "ROLE_ADMIN",
  "iat": 1731258000,
  "exp": 1731261600
}
```

Spring Security перевіряє токен на валідність, а у разі його відсутності або завершення терміну дії повертає код 401 Unauthorized. Це забезпечує базовий рівень захисту від несанкціонованого доступу до API.

У системі передбачено два основні рівні доступу:

1. **Загальний користувач (ROLE_USER)** – може створювати та редагувати власні списки і завдання, а також переглядати проєкти, до яких його додано як співробітника.
2. **Адміністратор (ROLE_ADMIN)** – має повний доступ до всіх користувачів, проєктів і статусів системи.

Контроль доступу реалізується за допомогою анотацій @PreAuthorize та @Secured, які визначають дозволені ролі для кожного ендпоінта [25].

Крім авторизації, Spring Security підтримує механізми захисту від атак типу **CSRF**, **Brute Force** та **Session Fixation**. У межах REST API, де використовується JWT, CSRF-захист зазвичай вимикається, оскільки сервер не зберігає стан сесії. Для збереження конфіденційності паролі користувачів хешуються за допомогою **BCrypt**, що гарантує безпеку навіть у разі витоку бази даних.

У тезі [1] зазначено, що використання JWT є ефективним способом побудови безпечного REST API, оскільки токен не потребує зберігання на сервері та може перевірятись на валідність без додаткових запитів до бази даних. Крім того, підхід дозволяє легко інтегрувати систему з іншими сервісами або клієнтами без зміни архітектури безпеки.

Обрана модель безпеки – комбінація Spring Security, JWT і ролей користувачів – забезпечує необхідний баланс між зручністю, масштабованістю та захистом даних. Вона повністю відповідає архітектурі REST, оскільки не порушує принцип безстановості запитів і дозволяє системі обробляти велику кількість одночасних звернень без надмірного навантаження на сервер.

2.5 Висновки до розділу

У другому розділі було розглянуто основні етапи проєктування архітектури та реалізації API системи управління завданнями. На основі аналізу обґрунтовано вибір **REST-архітектури**, яка забезпечує стандартизовану взаємодію клієнта і сервера через HTTP-протокол, підтримує масштабованість і легку інтеграцію з вебта мобільними застосунками. У межах обраної архітектури визначено використання **MVC-підходу** (Controller–Service–Repository), що дозволяє розділити відповідальність між рівнями системи, підвищити читабельність коду і спростити його супровід. Крім того, у процесі проєктування враховано **SOLID-принципи**, які гарантують гнучкість і стійкість коду до змін у бізнес-логіці.

Створена структура бази даних відображає основні сутності предметної області – користувачів, списки завдань, самі завдання, статуси та співробітників – і визначає чіткі зв’язки між ними. ER-модель системи забезпечує цілісність даних, уникає дублювання інформації та дозволяє реалізувати необхідні CRUD-операції в межах REST API. Така модель є оптимальною для подальшої реалізації системи на базі Spring Data JPA.

Розроблена специфікація API описує основні ендпоінти для взаємодії з системою, включаючи операції створення, оновлення, перегляду й видалення завдань, користувачів і списків. Вона забезпечує узгодженість запитів і відповідей у форматі JSON, що спрощує тестування та документування за допомогою OpenAPI (Swagger).

Окрему увагу приділено питанням **безпеки**. У системі реалізовано авторизацію й автентифікацію користувачів за допомогою **Spring Security** та **JWT**, що дозволяє перевіряти користувача без необхідності зберігання сесій на сервері. Такий підхід відповідає принципу безстановості REST і забезпечує високий рівень захисту персональних даних. Крім того, реалізація ролей

користувачів (ROLE_USER, ROLE_ADMIN) дає змогу гнучко керувати доступом до ресурсів системи.

Результати другого розділу формують технічну основу для реалізації REST API системи управління завданнями. Обрані архітектурні рішення, структура даних, логіка взаємодії та механізми безпеки створюють умови для побудови масштабованої, надійної та захищеної серверної частини застосунку, що відповідає сучасним стандартам розробки.

3 РЕАЛІЗАЦІЯ API НА БАЗІ SPRING FRAMEWORK

3.1 Середовище розробки та інструменти

Реалізація REST API системи управління завданнями здійснювалась у сучасному програмному середовищі, яке забезпечує повний цикл розробки — від написання коду до тестування та розгортання серверного застосунку. Основним інструментом виступило середовище IntelliJ IDEA, що є одним із найпотужніших IDE для Java-розробки й надає широкий спектр можливостей, цілком орієнтованих на проєкти, реалізовані на Spring Framework. Середовище забезпечує підтримку інтелектуального автодоповнення, аналізу коду в реальному часі, автоматичного рефакторингу та навігації між класами, що значно зменшує ризик виникнення помилок і підвищує продуктивність розробника.

Важливою перевагою IntelliJ IDEA є тісна інтеграція з Maven — системою керування залежностями, яка використовується у проєкті. IDE автоматично завантажує необхідні бібліотеки, відслідковує зміни у файлі pom.xml та синхронізує структуру застосунку без додаткових налаштувань. Також це середовище містить вбудовані засоби роботи з базами даних, що дозволяє розробнику переглядати структуру таблиць, виконувати SQL-запити, тестувати взаємодію з PostgreSQL і відлагоджувати репозиторії без необхідності перемикання на сторонні інструменти.

Крім того, IntelliJ IDEA забезпечує глибоку інтеграцію зі Spring Boot. IDE автоматично визначає точки запуску, конфігураційні класи, біни та залежності контексту. Це дозволяє значно пришвидшити процес створення REST-ендпоінтів, налаштування безпеки, тестування контролерів та сервісів. Інструменти дебагінгу та профілювання дають змогу простежити логіку виконання запитів, оцінити продуктивність методів та виявити потенційні вузькі місця. Завдяки своїй стабільності, інтелектуальним можливостям і широкій

екосистемі плагінів IntelliJ IDEA стала оптимальним середовищем для реалізації серверної частини таск-менеджера та забезпечила високу швидкість розробки і якість кінцевого рішення.

Для керування залежностями та збирання серверної частини застосунку було використано Apache Maven — один із найбільш поширених інструментів у Java-екосистемі, що забезпечує стандартизовану структуру проєктів і автоматизує всі основні етапи їхнього життєвого циклу. Maven дозволяє уникнути проблем із сумісністю бібліотек завдяки централізованому визначенню залежностей у файлі *pom.xml*, що забезпечує однозначність конфігурації та прозорість процесу збирання. Такий підхід мінімізує ручну роботу розробника, автоматизує завантаження потрібних артефактів і значно спрощує підтримку проєкту у довгостроковій перспективі.

Використання Maven дало змогу без труднощів інтегрувати Spring Boot Starter-и, які містять оптимальні набори бібліотек для REST-контролерів, роботи з JPA, валідацією даних, Spring Security та JWT-аутентифікацією. Завдяки цьому налаштування базових модулів системи управління завданнями було зведено до мінімуму, оскільки більшість конфігурацій виконується автоматично. Maven також забезпечив можливість використання сучасних плагінів, зокрема для запуску тестів (Surefire, JUnit), створення виконуваних артефактів, генерації документації та аналізу коду.

Принцип *“configuration over convention”*, на якому базується Maven, дозволив зберегти структуру проєкту зрозумілою та передбачуваною, а сам процес збирання — повторюваним та стабільним. Це особливо важливо під час командної роботи або розгортання на різних середовищах, оскільки забезпечує однакову поведінку застосунку незалежно від локальної конфігурації розробників. У результаті використання Maven стало ключовим фактором у забезпеченні надійності, масштабованості та простоти супроводу створеного REST API [26].

У якості системи керування базами даних для серверної частини застосунку було обрано PostgreSQL, що є однією з провідних реляційних СУБД

з відкритим вихідним кодом. PostgreSQL інтенсивно використовується у промислових та корпоративних рішеннях завдяки своїй високій продуктивності, стійкості до збоїв, підтримці транзакційності рівня ACID та здатності обробляти складні SQL-запити. Її архітектура забезпечує можливість горизонтального та вертикального масштабування, що робить систему придатною для сценаріїв із великими обсягами даних або значною кількістю одночасних користувачів. Крім того, PostgreSQL підтримує розширення функціональності через модулі та користувацькі типи даних, що дозволяє адаптувати її до специфічних вимог інформаційної системи.

Вибір PostgreSQL також обумовлений її тісною інтеграцією зі Spring Boot та Spring Data JPA, що надає зручні засоби для автоматичного генерування SQL-операцій, управління транзакціями та мапування об’єктів доменної моделі на реляційну структуру. Механізм ORM, реалізований за допомогою JPA та Hibernate, дає змогу працювати з даними на рівні Java-об’єктів, мінімізуючи необхідність ручного написання SQL-запитів та знижуючи ризик помилок. У проєкті було сформовано реляційну модель, яка включає таблиці користувачів, ролей, списків завдань, окремих задач і статусів задач — така структура повністю відображає логіку предметної області й забезпечує цілісність та зв’язність даних.

Для забезпечення коректної роботи застосунку було налаштовано драйвер PostgreSQL, конфігураційні параметри у файлі *application.properties*, а також створено набір JPA-ентіті, що автоматично мапувалися на відповідні таблиці за допомогою анотацій. Це забезпечило прозорий зв’язок між об’єктною моделлю застосунку та фізичним рівнем зберігання даних. Крім того, Spring Boot дозволив автоматизувати створення таблиць, підтримку схем та оновлення структури бази даних при зміні моделі, що значно спростило розробку та подальший супровід системи [27].

Поєднання таких інструментів, як IntelliJ IDEA, Maven і PostgreSQL, сформувало комплексне та надійне програмне середовище для реалізації REST API, забезпечивши високу ефективність розробки та стабільність роботи всієї системи. IntelliJ IDEA надала широкі можливості для автоматизації

конфігурацій, швидкої навігації по класах, інтеграції зі Spring Boot та налагодження коду, що значно скоротило час розробки та зменшило кількість технічних помилок. Maven, у свою чергу, забезпечив централізоване керування залежностями, стандартизацію структури проєкту та автоматизацію процесів збирання й тестування, що зробило розробку передбачуваною та масштабованою. PostgreSQL як основна СУБД надала можливість надійного зберігання даних, виконання складних транзакцій та гнучкого структурування інформації, що є ключовим для систем управління завданнями.

Усі ці інструменти органічно поєднуються завдяки Spring Boot, який забезпечує автоматичне налаштування компонентів, інтегрує ORM-рівень через Spring Data JPA та оптимізує взаємодію з базою даних без необхідності складних ручних конфігурацій. Завдяки такій інтеграції розробник може зосередитися на створенні бізнес-логіки й функціональних компонентів системи, не витрачаючи час на низькорівневі деталі середовища виконання. У результаті сформована архітектура забезпечує можливість подальшого масштабування, розширення функціоналу та впровадження нових сервісів без суттєвих змін у структурі застосунку.

3.2 Реалізація моделі даних

Модель даних у системі управління завданнями реалізована за допомогою механізму **JPA (Jakarta Persistence API)** у поєднанні з **ORM-фреймворком Hibernate**, який інтегрований у Spring Boot. Основною метою створення моделі даних є відображення структури таблиць PostgreSQL у вигляді Java-класів (**Entity**) та забезпечення автоматичного мапінгу між об’єктами програми та записами в реляційній базі даних. Модель відтворює структуру предметної області, подану у вигляді ER-діаграми в попередньому розділі, включаючи сутності користувача, списку завдань, завдання, статусу та співпраці.

Реалізація **Entity-класів** передбачає використання анотацій @Entity, @Table, @Id, @GeneratedValue, які визначають таблицю, первинний ключ і стратегію генерації ключів. Наприклад, класи User, Task, ToDo, State та проміжна таблиця TodoCollaborator відповідають окремим сутностям бази даних. Для кожного класу передбачені відповідні поля, що точно мапуються на стовпці таблиць (@Column), а також типи даних, узгоджені з PostgreSQL. Паролі користувачів зберігаються у вигляді хешу, що відповідає вимогам безпеки та рекомендаціям Spring Security.

У моделі даних реалізовано різні типи зв’язків між сутностями:

- **один-до-багатьох** (@OneToMany) між User і ToDo, що відображає можливість одного користувача створювати декілька списків;
- **багато-до-одного** (@ManyToOne) між Task та ToDo, що забезпечує належність кожного завдання до конкретного списку;
- **багато-до-одного** між Task і State, що визначає стан виконання завдання;
- **багато-до-багатьох** (@ManyToMany) між User і ToDo через проміжну таблицю todo_collaborator, яка реалізована у вигляді окремої сутності.

Усі зв’язки визначено з використанням @JoinColumn та @JoinTable, що забезпечує правильність мапінгу та автоматичну генерацію SQL-запитів. Для підвищення стабільності моделі використано каскадні операції (cascade = CascadeType.ALL або CascadeType.REMOVE) там, де це виправдано, а також ліниве завантаження (FetchType.LAZY) для запобігання зайвому навантаженню на базу даних.

Концепція JPA дозволила мінімізувати кількість ручних SQL-операцій та надати можливість працювати з даними у вигляді об’єктів Java. Використання ORM-підходу також зменшує кількість помилок, пов’язаних із неправильними SQL-інструкціями, і забезпечує простіший супровід системи. Крім того, модульність та чітка структура Entity-класів забезпечують гнучкість у подальшій розробці та можливість легко додавати нові сутності або розширювати існуючі без порушення загальної архітектури [28-29].

Реалізована модель даних повністю відповідає логіці предметної області та вимогам системи управління завданнями. Вона забезпечує узгодженість, коректність і масштабованість даних, що є критично важливими для правильної роботи бізнес-логіки та REST API.

3.3 Реалізація DTO та сервісного рівня

Розглянемо роботу основних компонентів системи: контролера, сервісного рівня, DTO-класів та репозиторію – на прикладі сутності користувача **(User)**, оскільки цей приклад найбільш повно відображає взаємодію між різними шарами архітектури. Реалізація моделі роботи з користувачем демонструє принципи побудови REST API в межах Spring Boot, включно з отриманням даних через репозиторій, обробкою бізнес-логіки у сервісах, формуванням відповідей через DTO та контролем доступу через Spring Security.

Для передачі даних використовується DTO-клас UserResponse, який містить тільки необхідні поля – ідентифікатор, ім’я, прізвище, електронну пошту та роль. Використання анотації Lombok @Value забезпечує незмінність об’єктів цього класу, а @JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class) перетворює назви полів у формат snake_case, що відповідає поширеним практикам побудови REST API. Конструктор UserResponse(User user) вибірково копіює дані з сутності, не включаючи чутливу інформацію, таку як пароль, що підвищує безпеку й узгодженість відповідей API [30-31].

Сервісний рівень визначено інтерфейсом UserService, який окреслює загальні операції – створення, оновлення, читання за id або email, видалення та отримання списку користувачів. Його реалізація UserServiceImpl інкапсулює бізнес-логіку: перевіряє коректність вхідних даних, обробляє випадки відсутності користувача, викликає методи репозиторію та забезпечує узгодженість операцій. Наприклад, перед оновленням користувача сервіс

викликає readById() для перевірки, що об’єкт існує; при пошуку за email – кидає EntityNotFoundException, якщо запис відсутній. Сервіс виступає центральною частиною, яка поєднує поведінку програми та доступ до даних, залишаючи контролеру тільки завдання маршрутизації запитів і формування відповідей.

Важливу роль у роботі з користувачами відіграє CustomUserDetailsService, який реалізує механізм автентифікації Spring Security. Він завантажує користувача через UserRepository, створює список прав доступу на основі ролі та повертає об’єкт типу UserDetails, який використовується у JWT-фільтрах та під час входу до системи. Це забезпечує коректну інтеграцію бізнес-моделі користувача з механізмами безпеки та дозволяє застосунку визначати права доступу до API-ендпоінтів.

3.4 Реалізація контролерів і REST API

Реалізація REST API у проєкті здійснюється через набір контролерів, що відповідають за обробку HTTP-запитів та взаємодію користувача із системою. Контролери становлять верхній рівень архітектури та реалізують REST-ендпоінти для роботи з основними сутностями – користувачами, списками завдань (ToDo) та завданнями (Task). Вони використовують сервіси для виконання бізнес-логіки та DTO-класи для формування структурованих і безпечних відповідей. Такий підхід відповідає принципу розділення відповідальностей і забезпечує чисту архітектуру програми.

Усі контролери в проєкті анотовані за допомогою @RestController, що дозволяє Spring автоматично серіалізувати об'єкти у формат JSON. Базові маршрути налаштовуються через @RequestMapping, а конкретні операції – через методи @GetMapping, @PostMapping, @PutMapping та @DeleteMapping, що узгоджено з REST-підходом. Кожен контролер орієнтований на CRUD-функціональність: контролер користувачів (UserController) дозволяє переглядати

профілі, контролер завдань (TaskController) керує створенням, оновленням і видаленням завдань, а контролер списків (ToDoController) реалізує управління проєктами й співучасниками.

Взаємодія з клієнтом здійснюється через DTO-класи, що гарантує безпечність та узгодженість відповідей API. Наприклад, відповіді на запити щодо користувача формуються через UserResponse, який містить лише ті поля, що мають бути доступні клієнту. Це дозволяє уникнути витоку внутрішніх полів моделі, таких як хеші паролів або технічні службові параметри. Подібні DTO-класи (TaskResponseDto, ToDoResponseDto) застосовуються для всіх основних сутностей.

Усі контролери функціонують у тісній взаємодії зі службами безпеки. Перед виконанням бізнес-операції REST-ендпоінти перевіряють автентифікацію та авторизацію користувача через механізм Spring Security і JWT. Наприклад, доступ до /api/users залежить від ролі користувача: адміністратор може переглядати всі профілі, а звичайний користувач – лише власний. Аналогічно реалізовано контроль доступу до списків завдань та окремих задач, що забезпечує коректний рівень безпеки та дотримання приватності інформації.

Для кожної операції передбачено повернення HTTP-кодів відповідей, що полегшує інтеграцію з клієнтськими застосунками. У разі успіху повертаються коди 200 OK, 201 Created, 204 No Content; у разі помилок система формує стандартизовані відповіді через глобальний обробник винятків (GlobalExceptionHandler), що забезпечує уніфікований формат повідомлень про помилки.

Реалізація контролерів у системі забезпечує повну REST-функціональність, чітку структуру ендпоінтів, відокремлену бізнес-логіку через сервісний рівень, узгоджені DTO-класи та інтегровану модель безпеки. Це дозволяє створити гнучкий, передбачуваний і розширюваний API, що відповідає сучасним вимогам до вебсервісів.

3.5 Реалізація автентифікації та авторизації

У системі управління завданнями автентифікація та авторизація реалізовані за допомогою комбінації механізмів **Spring Security**, **JWT-токенів** та призначення ролей користувачам. Такий підхід дозволяє забезпечити безпечний доступ до REST API без використання серверних сесій, дотримуючись принципу безстановості (stateless), що є ключовим для REST-архітектури. Усі компоненти безпеки взаємодіють між собою через конфігурацію, фільтри та сервіс для завантаження користувачів.

Основним центром налаштувань є клас SecurityConfig, у якому визначається поведінка HTTP-запитів, правила доступу та фільтрація токенів. Через компонент SecurityFilterChain вимикається CSRF-захист (оскільки система не використовує сесії), задаються дозволені маршрути (/api/auth/login, /api/auth/register) та маршрути, доступні лише адміністраторам (/api/users). Усі інші запити вимагають наявності коректного JWT-токена. Конфігурація встановлює політику сесій як STATELESS, що означає – сервер не зберігає ніякого стану між запитами, покладаючись виключно на токен, переданий клієнтом [32].

JWT-токени обробляються спеціальним фільтром JwtAuthenticationFilter, який виконується перед стандартним UsernamePasswordAuthenticationFilter. Фільтр перехоплює кожен запит, перевіряє заголовок Authorization, витягує токен, валідує його за допомогою утиліти (JwtUtil) і при успішній перевірці встановлює дані користувача у контекст безпеки. Завдяки цьому всі подальші рівні – контролери, сервіси – знають, який користувач здійснює запит, і можуть перевіряти доступ відповідно до ролей або належності об’єктів.

Завантаження даних користувача для автентифікації здійснюється через сервіс CustomUserDetailsService, який реалізує інтерфейс UserDetailsService. Метод loadUserByUsername() отримує користувача з репозиторію UserRepository, а потім формує об’єкт UserDetails із зашифрованим паролем та

роллю у форматі Spring Security (ROLE_ADMIN, ROLE_USER). Саме цей об’єкт використовується під час перевірки токена або входу користувача до системи. Такий підхід гарантує централізований контроль автентифікації та є стандартною практикою у Spring Security [33].

Для захисту паролів використовується BCryptPasswordEncoder, що забезпечує адаптивне хешування, стійке до brute-force атак. Компонент authenticationProvider() створює DaoAuthenticationProvider, налаштовуючи його на використання сервісу користувачів та парольного енкодера. Таким чином, усі спроби входу виконуються через єдиний механізм, що забезпечує узгодженість перевірок та коректне використання хешованих паролів.

Ролі користувачів відіграють критичну роль у визначенні доступу до ресурсів. Конфігурація у SecurityConfig визначає такі правила:

- **Адміністратор (ROLE_ADMIN)** має доступ до всіх користувачів (/api/users).
- **Звичайний користувач (ROLE_USER)** може переглядати лише власний профіль і взаємодіяти з власними проєктами та завданнями.

Поєднання JWT, Spring Security та ролей дає змогу створити безпечний, масштабований та передбачуваний механізм захисту REST API. Такий підхід забезпечує захист персональних даних користувачів та ресурсів системи.

3.6 Тестування компонентів

Тестування компонентів є важливою частиною розробки REST API, оскільки воно забезпечує перевірку коректності бізнес-логіки, поведінки контролерів, взаємодії з базою даних та роботи сервісів у різних сценаріях. У проєкті застосовано три основні підходи до тестування: **модульне тестування сервісів**, **тестування контролерів на основі Spring MVC**, а також **інтеграційні тести для репозиторіїв**. Використання таких інструментів, як JUnit 5, Mockito,

MockMvc і DataJpaTest, дозволило комплексно протестувати головну функціональність системи.

Тестування репозиторіїв та частини сервісної логіки виконується не на основній базі PostgreSQL, а у спеціально піднятій **тестовій in-memory базі H2**, яка автоматично конфігурується завдяки анотації @DataJpaTest. Використання H2 дозволяє здійснювати інтеграційні тести значно швидше, без необхідності запускати зовнішню СУБД, а також гарантує повну ізоляцію тестового середовища від реальної бази. Усі операції – створення таблиць, збереження сутностей, виконання JPA-запитів – відбуваються в оперативній пам’яті й видаляються після завершення тестів, що забезпечує повторюваність результатів та безпеку продуктивних даних.

Тестування контролерів реалізовано за допомогою анотацій @WebMvcTest та MockMvc. Наприклад, у класі ToDoControllerTest контролер перевіряється у відриві від повної конфігурації Spring Boot, що дає змогу імітувати HTTP-запити типу GET або POST та перевіряти статус відповіді, назву відображення, модель та поведінку сервісів. У тестах використовуються мок-об’єкти (@MockBean), які замінюють залежності ToDoService, TaskService та UserService. Це дозволяє визначати очікувану поведінку методів сервісів та перевіряти логіку контролера без доступу до реальної бази даних. У тестах перевіряється правильність обробки створення, оновлення, перегляду та видалення списків завдань, а також взаємодія з механізмом безпеки через анотацію @WithMockCustomUser, що імітує автентифікованого користувача.

Сервісний рівень перевіряється у класі ToDoServiceTest за допомогою бібліотеки Mockito. Анотація @Mock дозволяє підмінити ToDoRepository та повністю контролювати його поведінку. Методи тестування включають: перевірку успішного створення об’єкта, реакцію на некоректні дані (наприклад, null), обробку винятків EntityNotFoundException, правильність роботи методів оновлення, видалення та пошуку. Для цього використовується функціонал Mockito: when(...).thenReturn(...), verify(...), assertThrows(...). Завдяки такому

підходу гарантується, що сервіс правильно обробляє валідні та невалідні сценарії без прив’язки до бази даних [34-35].

Інтеграційне тестування репозиторіїв реалізовано в класах RepositoryTest за допомогою анотації @DataJpaTest, яка піднімає мінімальний контекст Spring Boot і працює з тестовою базою даних (H2). Тести перевіряють повну логіку взаємодії з JPA: збереження об’єктів, пошук за користувачем, роботу кастомних SQL-запитів, а також поведінку бази у випадку кількох пов’язаних сутностей. Завдяки цьому можна підтвердити, що структура Entity-класів, зв’язки між ними та запити в репозиторії працюють правильно в реальному середовищі JPA.

Для зовнішнього тестування REST API використовується **Postman**, який дозволяє перевіряти роботу ендпоінтів у взаємодії з JWT-токенами, авторизацією, форматом запитів і відповідей. Це дає змогу протестувати роботу API з точки зору кінцевого користувача та підтвердити коректність поведінки всієї системи.

Завдяки поєднанню модульних, інтеграційних і REST-тестів проєкт забезпечує високу якість коду, стійкість до помилок та прогнозовану поведінку системи при різних сценаріях взаємодії.

3.7 Висновки до розділу

У даному розділі було реалізовано повноцінний серверний функціонал системи управління завданнями на основі Spring Framework. Було визначено та налаштовано середовище розробки, інтегровано інструменти для роботи з залежностями та базою даних, а також створено модель даних із використанням JPA та ORM-підходу. Реалізовано сервісний рівень і логіку взаємодії з репозиторіями, описано роботу контролерів, що забезпечують REST-ендпоінти для виконання CRUD-операцій із ключовими сутностями – користувачами,

списками завдань та окремими задачами. Окрему увагу приділено механізмам автентифікації та авторизації: інтегровано JWT-токени, побудовано структуру ролей та налаштовано фільтри безпеки для захисту API.

Крім цього, у розділі описано комплексний підхід до тестування: використано модульні тести з Mockito, MVC-тести контролерів із MockMvc та інтеграційні тести репозиторіїв у середовищі H2, що дозволило верифікувати коректність роботи застосунку без взаємодії з продуктивною базою PostgreSQL. Такий підхід забезпечив перевірку системи на різних рівнях, гарантував надійність бізнес-логіки та стабільність основних функцій API.

У межах розділу було побудовано повноцінну інфраструктуру back-end частини системи, що відповідає сучасним вимогам до безпеки, структурованості та тестованості серверних застосунків. Це створює основу для подальшої інтеграції API з клієнтською частиною, масштабування системи та впровадження нових функціональних можливостей у наступних етапах дослідження.

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Опис сценаріїв використання системи управління завданнями

У даній підсистемі реалізовано низку ключових сценаріїв взаємодії користувача з API, які відтворюють основні робочі процеси, характерні для сучасних систем управління завданнями. Кожен сценарій демонструє, яким чином клієнтська частина може взаємодіяти із сервером, починаючи від автентифікації користувача і закінчуючи створенням, редагуванням та спільною роботою над списками завдань і окремими задачами. Процеси побудовані на основі REST-підходу та включають послідовність запитів до відповідних ендпоінтів, формування стандартних відповідей, опрацювання можливих помилок, а також валідацію отриманих даних.

Сценарії враховують використання JWT-токенів як основного механізму підтвердження особи користувача. Кожен запит, що вимагає авторизації, супроводжується передачею токена у заголовку Authorization, після чого система перевіряє його валідність, термін дії та відповідність ролі запитуваним ресурсам. Такий підхід забезпечує безстанову взаємодію, характерну для REST-систем, та підвищує продуктивність завдяки відсутності потреби зберігати сесії на сервері. Додатково у сценаріях враховано рольову модель доступу: звичайний користувач може керувати лише власними ресурсами, тоді як адміністратор має розширені можливості взаємодії з іншими обліковими записами.

Описані сценарії охоплюють повний життєвий цикл об’єктів системи: від створення облікового запису та виконання входу до формування списків ToDo, керування задачами, додавання колабораторів та організації спільної роботи. Такі сценарії дозволяють продемонструвати усю функціональність API, перевірити коректність реалізації бізнес-логіки та підтвердити відповідність системи вимогам до багатокористувацьких застосунків. Завдяки деталізованому

опису вони також можуть використовуватися як основа для побудови клієнтських застосунків або документації для зовнішніх інтеграцій.

Першим і базовим сценарієм взаємодії з API є **автентифікація користувача та отримання JWT-токена** (додаток Б.2). На цьому етапі клієнтська частина надсилає POST-запит на ендпоінт **/api/auth/login**, передаючи email та пароль у тілі запиту. Контролер спрямовує ці дані до механізму автентифікації Spring Security, де вони перевіряються за допомогою **AuthenticationManager** і сервісу користувачів. У випадку успішної перевірки система генерує JWT-токен, що містить у собі зашифровану інформацію про користувача, його роль та час дії. Отриманий токен повертається клієнту та надалі використовується в усіх захищених запитах у вигляді заголовка **Authorization: Bearer <token>**, забезпечуючи коректну ідентифікацію користувача та доступ до приватних ресурсів API без необхідності підтримки серверних сесій.

Другим важливим сценарієм є **отримання інформації про користувача** (додаток Б.3), що демонструє роботу ролевої моделі доступу в системі. Звернення до ендпоінта **/api/users** дозволяє адміністраторам отримати повний список зареєстрованих користувачів, тоді як звичайний користувач може переглядати лише дані власного профілю. Це забезпечується перевіркою ролей та ідентифікатора користувача через **SecurityContextHolder**, де зберігається інформація, отримана з JWT-токена. Аналогічно працює ендпоінт **/api/users/{id}**, який повертає детальну інформацію про конкретного користувача, але лише за умови, що запитувач має відповідні права доступу — або є адміністратором, або запитує дані власного профілю. Таким чином, сценарій забезпечує безпечний та контрольований доступ до персональної інформації, відповідаючи принципам ролевої авторизації.

Третій сценарій охоплює процес створення та подальшого управління списками завдань (додаток Б.4), що є ключовим функціональним блоком системи. Користувач із роллю **USER** або **ADMIN** може ініціювати створення нового списку, надіславши POST-запит на **/api/todos** разом із DTO-об’єктом,

який містить основні параметри ToDo. Після створення списку користувач отримує можливість виконувати над ним усі необхідні операції через ендпоінти **/api/todos/{id}**, зокрема перегляд інформації, оновлення назви або опису, а також повне видалення списку. На кожному етапі виконується перевірка прав доступу: система визначає, чи є поточний автентифікований користувач його власником, і лише в такому випадку дозволяє змінювати або видаляти ToDo. Це забезпечує захищеність даних та унеможливлює некоректне втручання в чужі списки завдань.

Четвертий сценарій описує повний цикл роботи користувача із задачами (Task) у межах певного списку ToDo (додаток Б.5). За допомогою ендпоінта **/api/tasks** звичайний користувач отримує доступ лише до власних задач, тоді як адміністратор має можливість переглядати всі записи системи, що відповідає його розширеним правам. Створення нової задачі виконується через **POST /api/tasks**, де контролер здійснює кілька важливих перевірок: чи існує відповідний ToDo, чи належить він поточному користувачу та чи є вказаний стан задачі (State) коректним. Подальші операції, такі як оновлення (**PUT /api/tasks/{id}**) або видалення (**DELETE /api/tasks/{id}**), суворо контролюються системою безпеки: виконувати їх може лише власник списку або адміністратор. Такий підхід забезпечує цілісність даних, запобігає несанкціонованим змінам та формує передбачувану й безпечну модель роботи з задачами в системі.

Реалізовані сценарії демонструють повний і логічно завершений цикл взаємодії користувача із системою: від початкової автентифікації та отримання JWT-токена, що відкриває доступ до захищених ресурсів, до перегляду та модифікації власного профілю, управління списками ToDo та створення, редагування й видалення окремих завдань. Кожен зі сценаріїв супроводжується перевіркою ролей користувачів, валідацією даних і контролем доступу на основі приналежності ресурсів, що підтверджує коректність реалізації бізнес-логіки та відповідність API принципам безпеки. Сукупність описаних процесів демонструє функціональну завершеність розробленої системи та повну

відповідність її можливостей вимогам, характерним для сучасних систем управління завданнями.

4.2 Візуальне представлення роботи сценаріїв

У цьому підрозділі подано візуальне представлення основних сценаріїв використання системи управління завданнями. На основі інтерфейсів, зображених на скріншотах, нижче наведено опис ключових етапів взаємодії користувача із системою – від авторизації та роботи зі списками завдань до управління користувачами, станами задач і правами доступу. Кожен сценарій демонструє фактичну поведінку системи

Вхід до системи (рис. 4.1) здійснюється через окрему сторінку логіну, де користувач вводить електронну пошту та пароль. У разі успішної автентифікації система повідомляє про вхід і надає доступ до персонального кабінету. Додатково передбачено можливість реєстрації нового користувача, що забезпечує повний життєвий цикл роботи з обліковим записом.

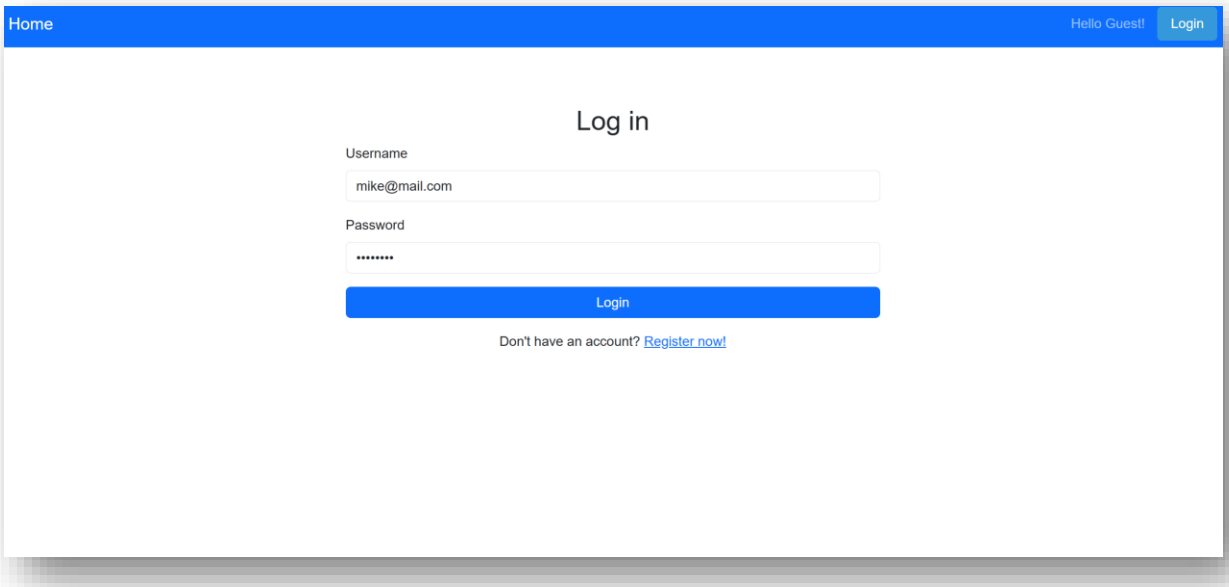

Рисунок 4.1 – Вхід до системи

Перегляд списку To-Do (рис. 4.2) дає змогу користувачу побачити всі власні списки завдань. Інтерфейс містить назву кожного To-Do, дату створення, інформацію про власника та можливість переходу до редагування чи видалення. Також доступна кнопка створення нового To-Do, що скорочує час на організацію робочих списків.

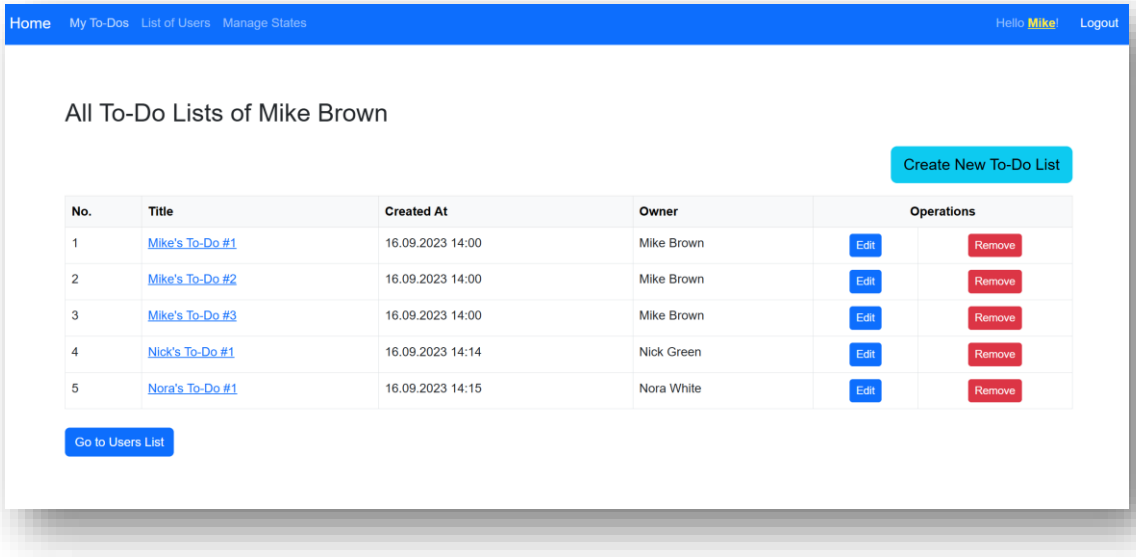

Рисунок 4.2 – Перегляд списку To-Do

Оновлення To-Do (рис. 4.3) виконується у формі редагування, де користувач може змінити назву вибраного списку. Інтерфейс побудований так, щоб відобразити лише необхідні поля, а після збереження змін користувач повертається до загального списку To-Do.

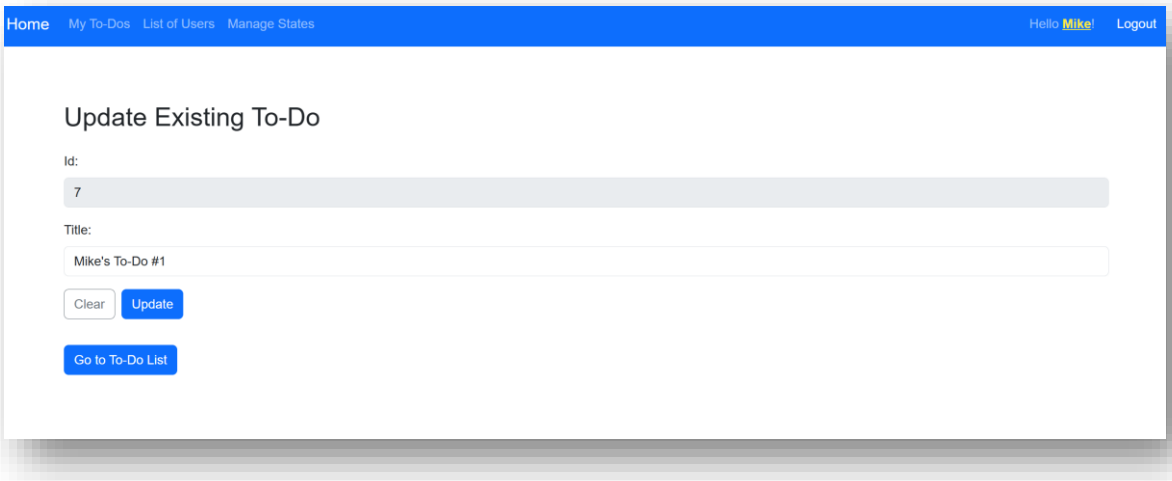

Рисунок 4.3 – Оновлення To-Do

Перегляд завдань у To-Do (рис. 4.4) дозволяє користувачу керувати всіма задачами у межах конкретного списку. На сторінці відображаються назва задачі, її пріоритет, стан та можливість виконати операції «Edit» або «Remove». Нижче розташований блок колаборатoрів, де можна побачити користувачів, які мають доступ до цього To-Do, а також додати або видалити колаборатора.

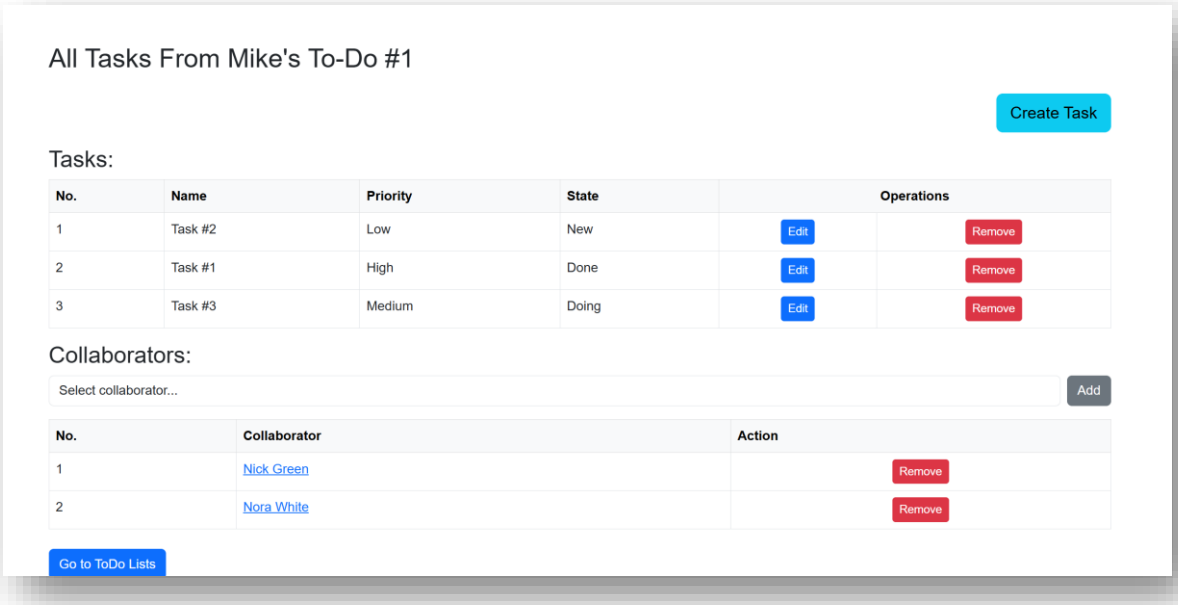

Рисунок 4.4 – Перегляд завдань у To-Do

Оновлення завдання (рис. 4.5) відбувається у формі редагування, де користувач може змінити назву, пріоритет і стан задачі. Такий підхід дозволяє оперативно вносити зміни до робочого процесу, а система автоматично зберігає оновлені дані.

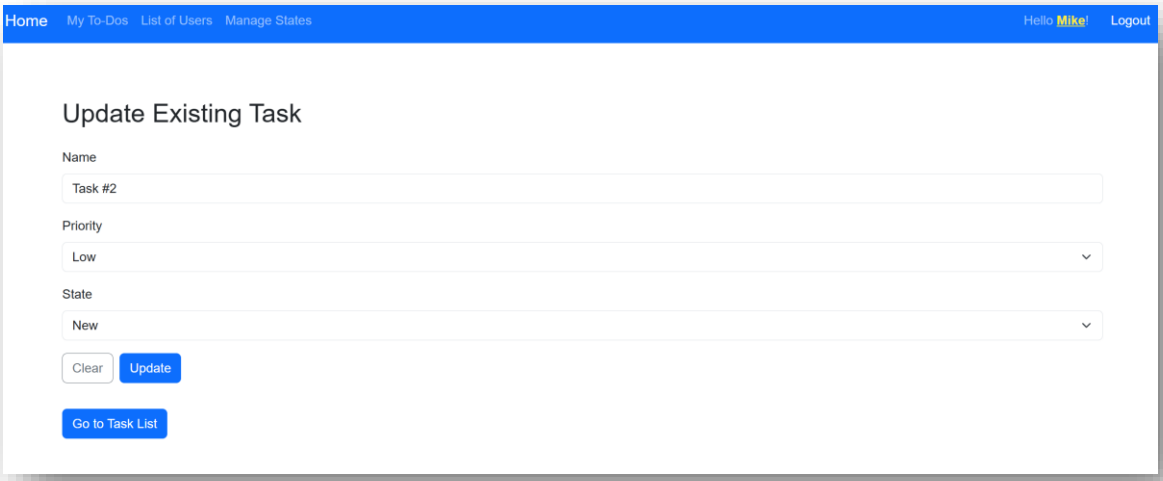

Рисунок 4.5 – Оновлення завдання

Перегляд списку користувачів (рис. 4.6) доступний лише адміністратору. Інтерфейс дає змогу бачити всіх зареєстрованих користувачів, їхній email, ім’я та прізвище. Також передбачено редагування користувачів і видалення облікових записів, що дає адміністратору можливість контролювати систему.

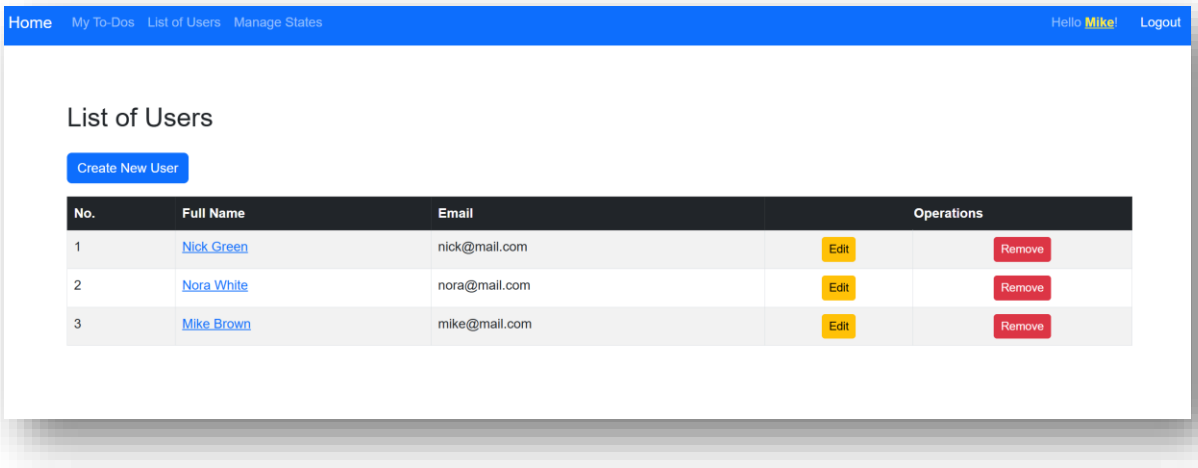

Рисунок 4.6 – Перегляд списку користувачів

Оновлення даних користувача (рис. 4.7) включає зміну імені, прізвища, електронної пошти та ролі. Ця функція доступна адміністраторам і забезпечує правильне розмежування прав доступу. Інтерфейс містить поля для майбутнього функціоналу зміни пароля, що відображає потенціал розширення системи.

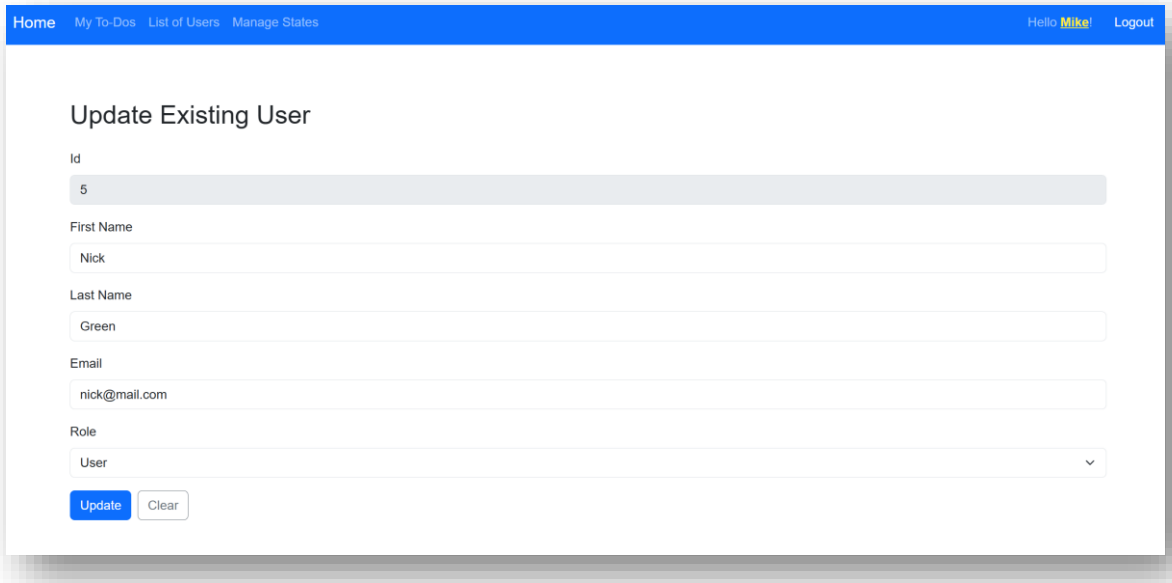

Рисунок 4.7 – Оновлення даних користувача

Управління станами задач (рис. 4.8) також доступне лише адміністраторам. На відповідній сторінці відображено перелік станів, які може мати завдання, з можливістю додавання нових, видалення або перейменування

існуючих. Це дозволяє гнучко адаптувати робочі процеси під потреби організації.

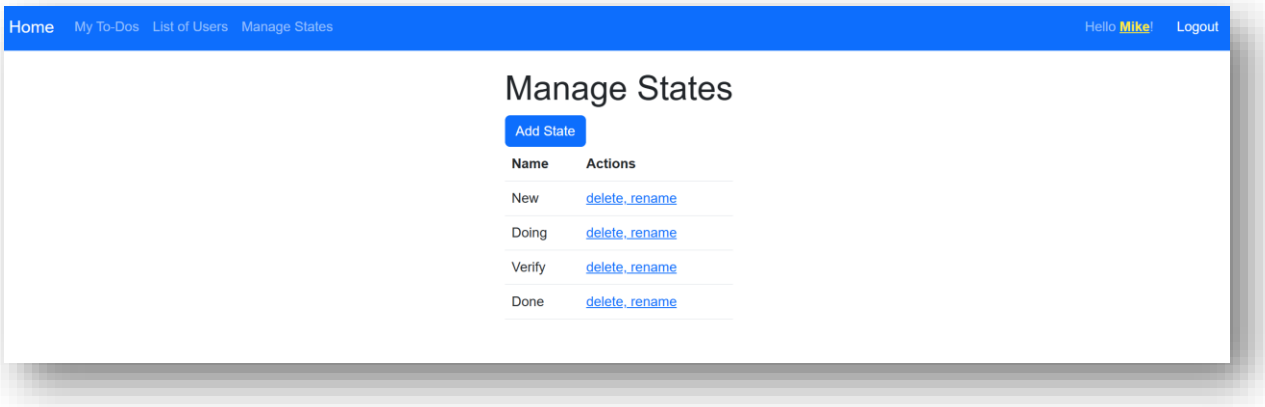

Рисунок 4.8 – Управління станами задач

Меню звичайного користувача (рис. 4.9) обмежена власними To-Do та задачами. Він не має доступу до списку користувачів або сторінки управління станами. Це відповідає реалізованій моделі ролей і забезпечує захист даних інших користувачів та елементів системи.

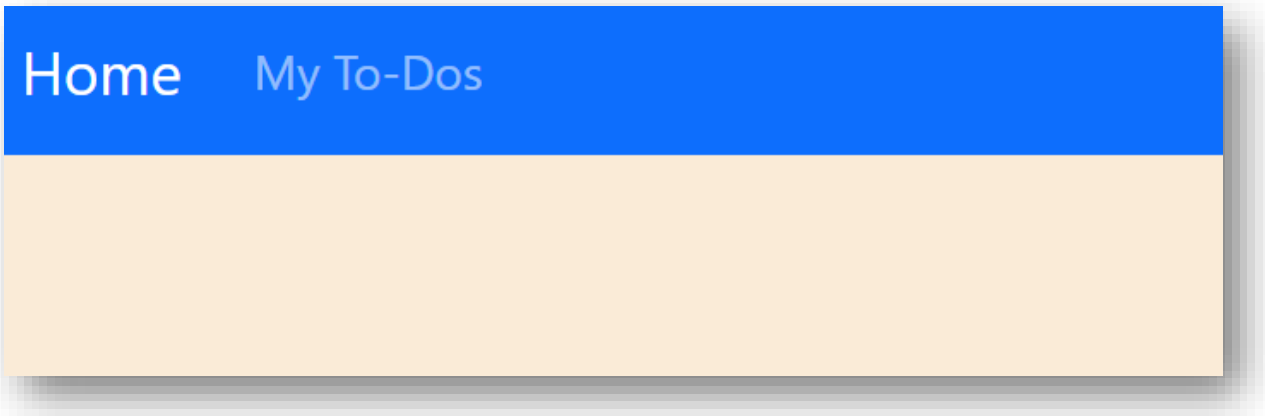

Рисунок 4.9 – Меню звичайного користувача

Робота із задачами звичайного користувача (рис. 4.10) не має обмежень щодо створення, редагування чи видалення, однак користувач не може змінювати склад колаборатів у To-Do. Таким чином забезпечується чітке

розмежування відповідальності: власник To-Do керує доступом, а учасники – виконують задачі.

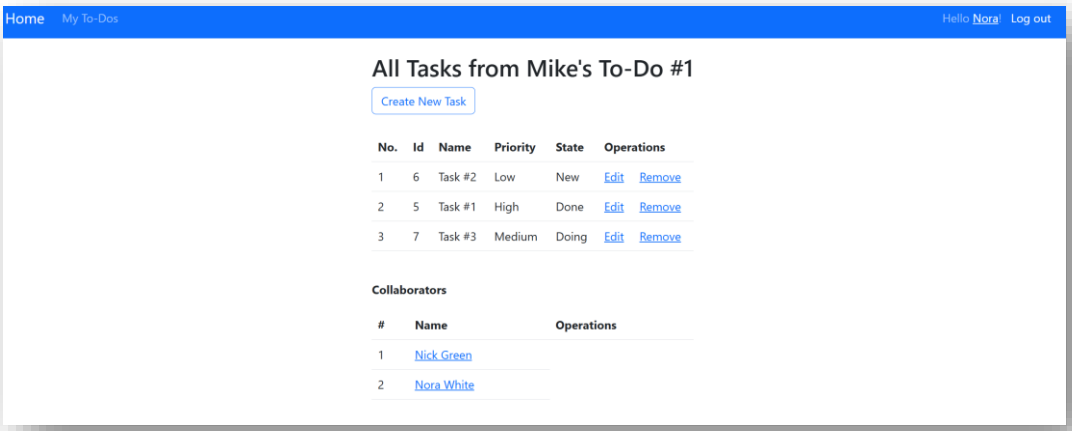

Рисунок 4.10 – Робота із задачами звичайного користувача

4.3 Перевірка роботи API через приклади запитів

Для підтвердження коректності функціонування реалізованого API було проведено експериментальну перевірку роботи його основних ендпоінтів. Перевірка охоплює процес автентифікації, виконання операцій над задачами, списками To-Do та користувачами, а також демонструє механізм контролю доступу відповідно до ролей користувачів. Усі запити здійснювалися у форматі HTTP із використанням JWT-токена, який клієнт отримує під час входу в систему.

Першим кроком є **автентифікація користувача**, що виконується через запит *POST /api/auth/login*. Контролер приймає email та пароль, після чого перевіряє їх через AuthenticationManager. У разі успіху формується JWT-токен, який повертається у відповіді. Саме цей токен необхідно передавати в усіх наступних запитах у заголовку Authorization: Bearer <token>. Таким чином токен

виступає ключовим елементом аутентифікації та дозволяє клієнту отримати доступ до захищених ресурсів API (рис. 4.11).

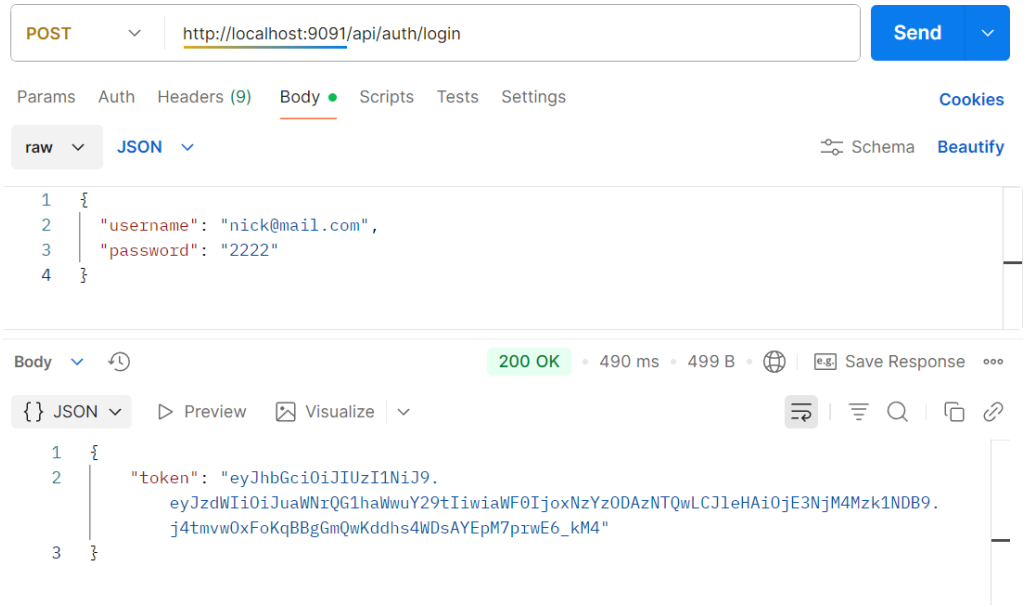

Рисунок 4.11 – Логін та отримання токена

Наступним кроком є перевірка роботи ендпоінтів керування To-Do списками. Запит *GET /api/todos* повертає перелік усіх To-Do, доступних поточному користувачу. Якщо запит виконує адміністратор, він отримує всі записи, а звичайний користувач – лише власні. Створення нового To-Do здійснюється через запит *POST /api/todos*, у якому користувач передає DTO-об’єкт із полями списку. У відповідь система повертає створений To-Do у форматі ToDoResponseDTO. Аналогічно працюють ендпоінти оновлення та видалення: *PUT /api/todos/{id}* та *DELETE /api/todos/{id}*, що дає змогу перевірити логіку редагування та дотримання політик доступу (рис. 4.12).

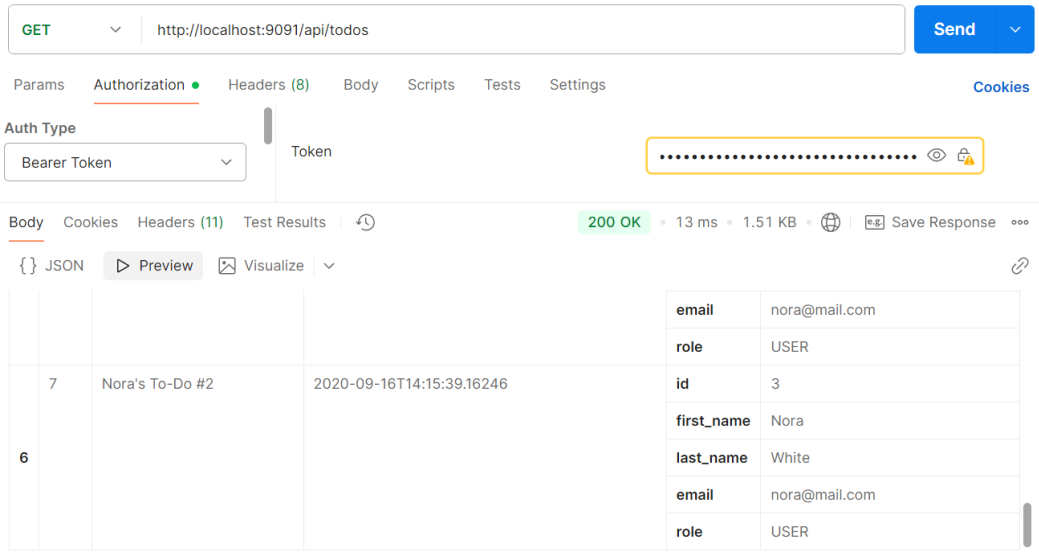

Рисунок 4.12 – Перегляд усіх To-Do адміністратором

При виведенні даних важливо виводити їх обмежену кількість, оскільки кожне To-Do містить інформацію про колаборatorів, кожен колаборатор про список To-Do і так програма буде збиватись через нескінечний цикл виведення даних.

Робота з задачами перевіряється за допомогою запитів до /api/tasks. Запит *GET /api/tasks* повертає список задач: адміністратор бачить усі, а звичайний користувач – лише ті, що належать до його власних To-Do. Створення задачі здійснюється через *POST /api/tasks*, де необхідно вказати ідентифікатор списку To-Do і стан задачі. Контролер перевіряє, чи має користувач право додавати завдання саме до цього To-Do. Оновлення задачі (*PUT /api/tasks/{id}*) та її видалення (*DELETE /api/tasks/{id}*) також супроводжуються перевірками: звичайний користувач не може редагувати або видаляти завдання в чужому To-Do.

Також було протестовано отримання інформації про користувачів. Ендпоінт *GET /api/users* повертає список усіх користувачів лише адміністраторам. Запит *GET /api/users/{id},* де користувач може переглядати лише свій профіль, а адміністратор – будь-який. Отже, робота цього ендпоінта підтверджує правильність інтеграції ролей та обмежень доступу (рис. 4.13-16).

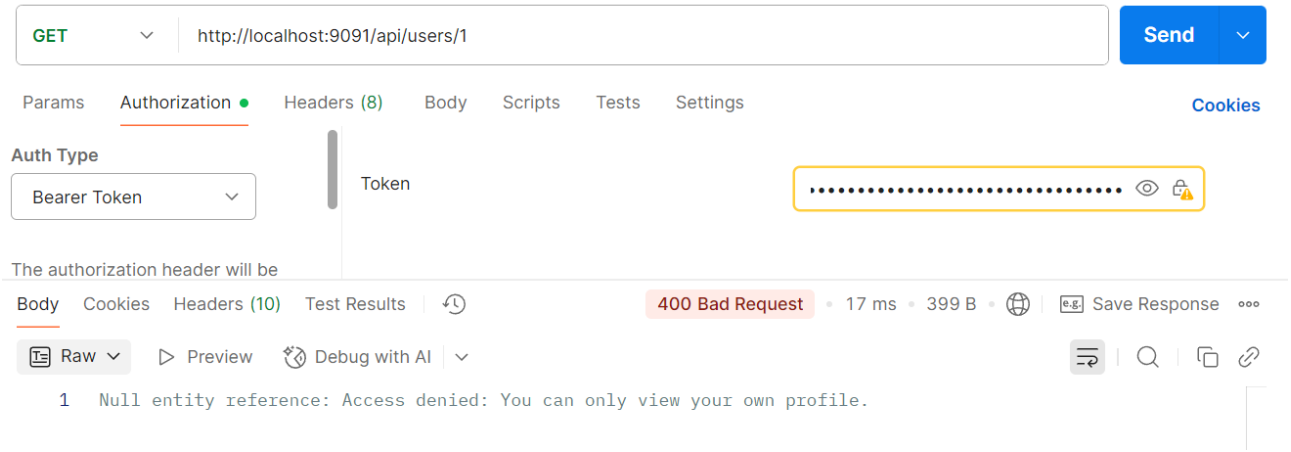

Рисунок 4.13 – Інформація про іншого користувача. Запит користувача

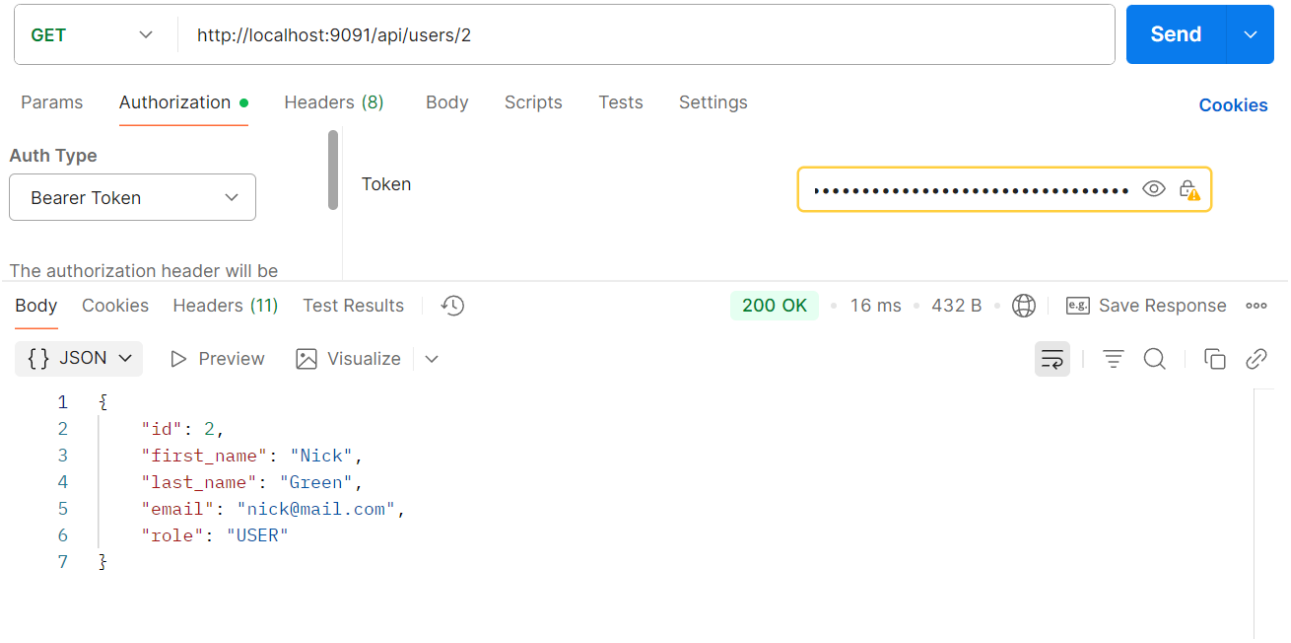

Рисунок 4.14 – Інформація про себе. Запит користувача

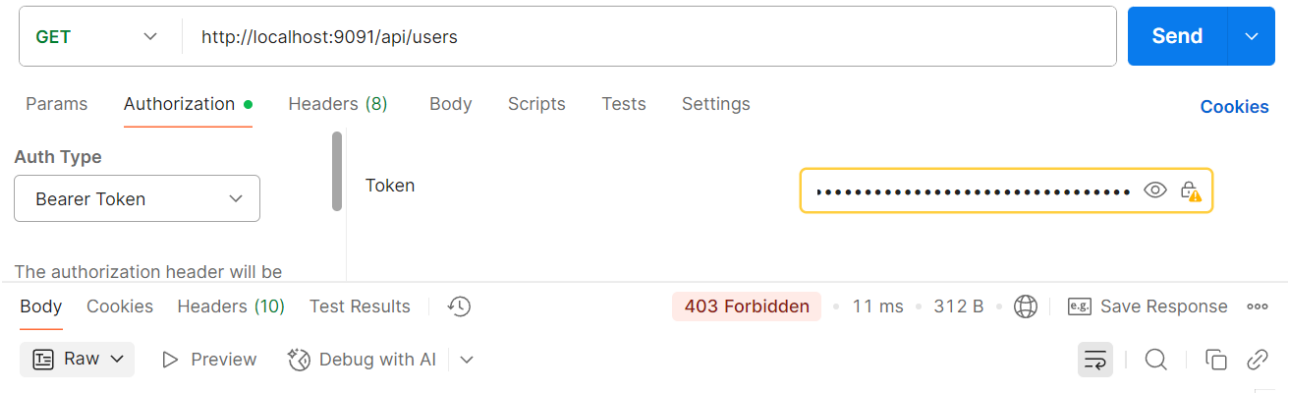

Рисунок 4.15 – Список усіх користувачів. Запит користувача

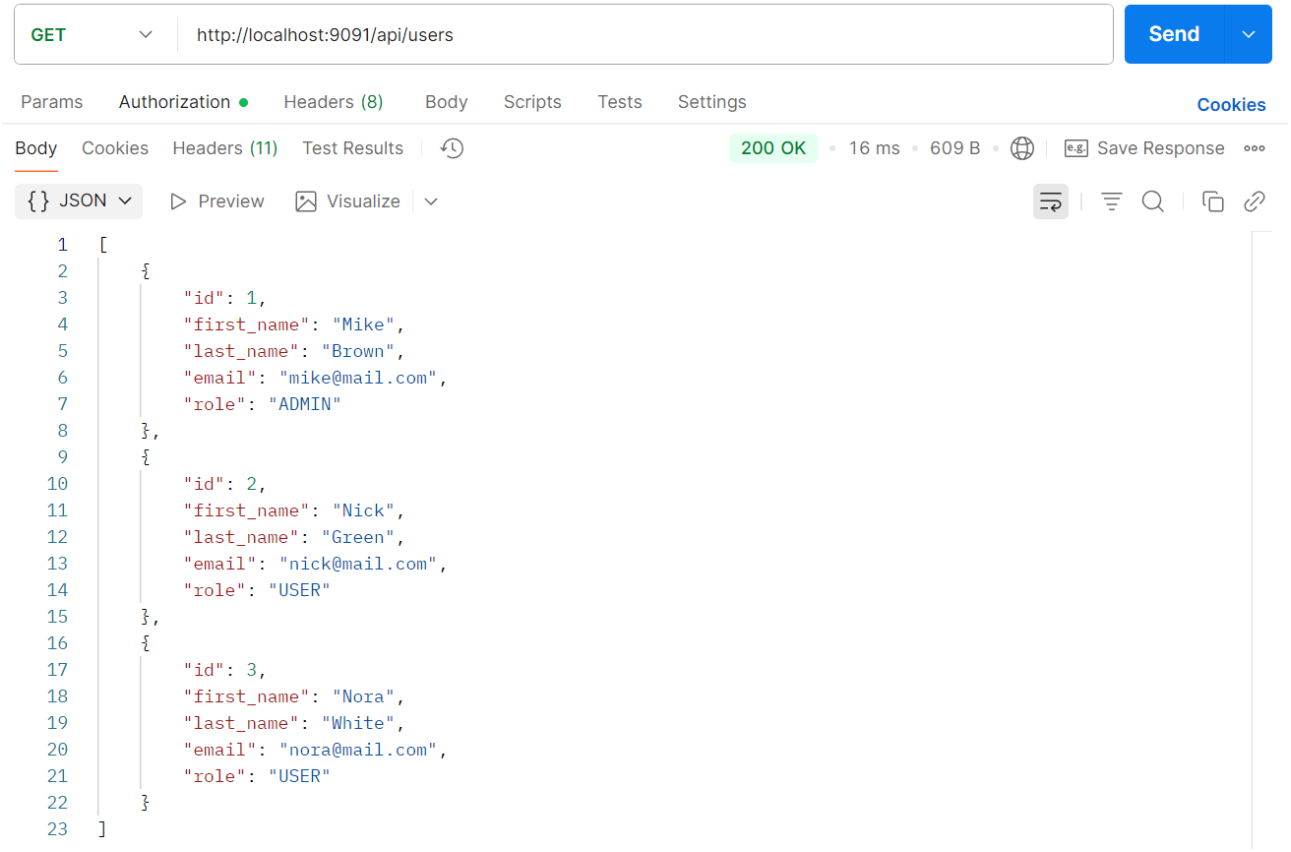

Рисунок 4.16 – Список усіх користувачів. Запит адміністратора

При спробі отримати дані іншого користувача отримуємо повідомлення, що ви як користувач можете переглянути лише свій профіль, оскільки цей ендпоінт вам доступний, але інформація – ні. Але при спробі отримати дані по всім користувачам отримуємо 403 помилку, оскільки цей ендпоінт заблокований до перегляду з правами користувача.

Отримані результати демонструють передбачувану та стабільну роботу API. Усі ендпоінти коректно реагують на наявність або відсутність прав доступу, повертають структуровані DTO-відповіді, а механізм JWT-аутентифікації забезпечує захищеність комунікації та ізоляцію ресурсів відповідно до ролі користувача. Таким чином, проведені експерименти підтвердили відповідність функціональності API вимогам системи управління завданнями та його готовність до інтеграції з клієнтськими застосунками.

4.4 Аналіз продуктивності та швидкодії API

Для оцінки ефективності роботи реалізованого API було проведено серію вимірювань часу відповіді основних ендпоінтів, що відображають ключові операції системи управління завданнями. Тестування здійснювалося шляхом п’яти повторних запитів для кожної операції, після чого обчислено середнє значення часу відповіді. Це дозволило отримати об’єктивну оцінку продуктивності системи та порівняти швидкодію різних типів запитів – читання, створення, оновлення та аутентифікації. Результати представлені на таблиці 4.1

Таблиця 4.1 – Продуктивність та швидкодія запитів

Назва запиту	1, мс	2, мс	3, мс	4, мс	5, мс	Середній час, мс
Логін	257	243	247	241	257	249
Отримання списку To-Do	16	14	13	13	13	13.8
Створення нового To-Do	32	15	14	13	14	17.6
Отримання всіх задач	40	13	15	14	13	19
Створення нової задачі	16	14	14	16	13	14.6
Оновлення задачі	81	16	17	16	14	28.8
Отримання списку користувачів	20	11	11	10	12	12.8

Отримані результати демонструють, що найвищий час відповіді спостерігається під час виконання операції логіну. Середній час у 249 мс пояснюється тим, що у процесі автентифікації відбувається не лише перевірка користувача в базі даних, а й виконання криптографічної операції хешування пароля за допомогою BCrypt та генерація JWT-токена. Це очікувано є найбільш ресурсомісткою операцією серед протестованих.

Операції, пов’язані з читанням даних, демонструють значно кращі показники. Отримання списку To-Do показало середній час відповіді 13.8 мс, що

свідчить про оптимальність реалізованої логіки доступу до даних та мінімальні накладні витрати на обробку запиту. Аналогічно, отримання списку користувачів також показало низький середній час – 12.8 мс, що характерно для невеликого обсягу вибірки та ефективної реалізації репозиторію. Запит отримання всіх задач є більш навантаженим, оскільки включає обробку пов’язаних сутностей і додаткові фільтри, що пояснює його вищий середній час – 19 мс.

Операції створення нових ресурсів займають дещо більше часу, ніж читання, що зумовлено необхідністю запису даних до бази та виконанням внутрішньої валідації. Створення To-Do має середній час 17.6 мс, тоді як створення задачі – 14.6 мс. В обох випадках продуктивність залишається на високому рівні та свідчить про оптимальну роботу сервісного шару.

Найбільш ресурсоємною CRUD-операцією виявилося оновлення задачі, середній час якого становив 28.8 мс. Це пов’язано з додатковими перевірками доступу, пошуком пов’язаних сутностей (To-Do та State), трансформацією DTO у сутність та виконанням транзакційного оновлення. Попри збільшений час у порівнянні з іншими CRUD-операціями, значення залишається в межах норми для подібних задач.

Загалом проведений аналіз підтверджує, що продуктивність API відповідає вимогам до сучасних веб-сервісів. Час відповіді для всіх протестованих операцій знаходиться у межах, що забезпечують комфортну взаємодію клієнта з системою, а реалізована архітектура дозволяє ефективно масштабувати проєкт. Отримані результати демонструють збалансовану роботу сервісного шару, оптимальну організацію бази даних та коректну інтеграцію механізмів автентифікації.

4.5 Оцінка відповідності поставленим вимогам

На основі проведених експериментальних досліджень, аналізу продуктивності та функціонального тестування можна зробити висновок, що розроблений API системи управління завданнями у повному обсязі відповідає поставленим вимогам. Реалізовані модулі забезпечують необхідний набір функціональних можливостей, включаючи авторизацію та автентифікацію користувачів на основі JWT, управління сутностями To-Do, завданнями, станами та користувачами, а також механізми контролю доступу відповідно до ролей.

Функціональні вимоги були виконані шляхом впровадження REST-архітектури, що забезпечує логічне структурування ендпоінтів, підтримку CRUD-операцій та коректну роботу з клієнтськими запитами через уніфіковані HTTP-методи. Кожен контролер API перевірений тестами, що гарантує стабільність роботи при різних сценаріях використання та дозволяє впевнено оцінювати надійність системи. Результати тестування засвідчили, що всі основні операції виконуються коректно та відповідають очікуваній поведінці.

Нефункціональні вимоги також були враховані. Проведена оцінка швидкодії продемонструвала, що час відповіді для основних операцій знаходиться у допустимих межах, а найпоширеніші запити виконуються з мінімальною затримкою. Це свідчить про ефективність реалізованого сервісного шару та оптимальну взаємодію з базою даних. Використання H2-середовища для тестування дозволило імітувати роботу основної БД та забезпечити повну ізоляцію тестів, що є важливим аспектом для перевірки коректності логіки без впливу зовнішніх факторів.

Розроблена система відповідає функціональним, архітектурним і продуктивним вимогам, визначеним на етапі постановки задачі. API продемонстрував стабільну роботу, коректну обробку даних та відповідність стандартам розробки, що підтверджує доцільність подальшого розгортання, розширення функціональності та інтеграції системи у більш масштабні рішення.

4.6 Висновки до розділу

У даному розділі було проведено комплексне експериментальне дослідження роботи API системи управління завданнями, що дало змогу оцінити його функціональну коректність, стабільність та продуктивність у реальних сценаріях використання. На основі аналізу виконання основних операцій – автентифікації, роботи зі списками To-Do, управління задачами та обробки запитів користувачів – встановлено, що система демонструє передбачувану й узгоджену поведінку відповідно до вимог, визначених на етапі проєктування.

Візуальне представлення сценаріїв показало, що інтерфейс і логіка роботи API є інтуїтивними, а рольові обмеження коректно розмежовують доступ користувачів, підвищуючи безпеку взаємодії з даними. Аналіз прикладів запитів підтвердив правильність реалізації механізму JWT-аутентифікації, що забезпечує захист ендпоінтів та дозволяє використовувати отриманий токен у подальших запитах, відповідно до сучасних стандартів безпеки веб-систем.

Дослідження продуктивності показало, що середній час відповіді для всіх ключових запитів залишається на низькому рівні, а найбільше навантажені операції – такі як логін та оновлення задач – також демонструють стабільні результати з прийнятною затримкою. Це свідчить про ефективну організацію сервісного шару та оптимальну взаємодію з базою даних.

Проведені експерименти підтвердили, що реалізований API є працездатним, відповідає функціональним і нефункціональним вимогам та може бути використаний як основа для подальшого розвитку системи управління завданнями.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки програмного забезпечення

Комерційний потенціал програмного забезпечення визначає можливість його практичного використання, здатність забезпечити економічний ефект та перспективи подальшого впровадження на ринку. Для оцінювання ефективності розробленого API для системи управління завданнями було застосовано метод експертних оцінок. Такий підхід дозволяє отримати об’єктивну характеристику перспективності проєкту на основі сукупності вагомих критеріїв, що впливають на його конкурентоздатність та можливість комерціалізації.

Оцінювання проводилось за дванадцятьма критеріями, рекомендованими методичними вказівками, за п’ятибальною шкалою. До експертного оцінювання було залучено фахівців, компетентних у галузях програмної інженерії, інформаційних систем та управління проєктами.

Таблиця 5.1 – Критерії оцінювання комерційного потенціалу ПЗ

№	Критерій оцінювання	0	1	2	3	4	5
Технічна здійсненність концепції							
1	Достовірність концепції	Концепція не підтверджена	Концепція підтверджена теоретично	Концепція підтверджена розрахунками	Реалізовано прототип	Проведено тестування в лабораторних умовах	Реалізовано тестування в реальних умовах
2	Захист від сучасних кіберзагроз	Захист не забезпечено	Базовий рівень захисту	Часткový захист від поширених атак	Високий рівень захисту	Захист від передових атак забезпечено	Постійний моніторинг і оновлення забезпечено

Ринкові переваги (недоліки)							
3	Унікальність розробки	Аналогів багато	Є часткові унікальні елементи	Унікальність обмежена	Значна частина функцій унікальна	Високий рівень унікальності	Унікальна технологія
4	Потенціал впровадження на ринок	Дуже малий ринок	Ринок малий	Середній потенціал	Високий попит на ринку	Широкий потенціал	Стабільний попит і розширення ринку
5	Конкуренція	Висока конкуренція	Часткова конкуренція	Конкуренція обмежена	Низька конкуренція	Мало конкурентів	Конкуренції немає
Економічна доцільність							
6	Витрати на реалізацію	Дуже високі витрати	Високі витрати	Середні витрати	Низькі витрати	Мінімальні витрати	Інвестиції непотрібні
7	Термін окупності	Окупність >10 років	5–10 років	3–5 років	<3 років	Менше 1 року	Окупність відразу після впровадження
Практична здійсненність							
8	Потреба у фахівцях	Висококваліфіковані фахівці	Потрібні досвідчені спеціалісти	Фахівці середнього рівня	Можна без спеціальної підготовки	Кваліфікація не потрібна	Повна автоматизація
9	Час реалізації	Більше 5 років	3–5 років	1–3 роки	<1 року	До 6 місяців	Можна реалізувати за 3 місяці
10	Ризики впровадження	Ризики дуже високі	Ризики високі	Помірні ризики	Низькі ризики	Ризики мінімальні	Ризики відсутні
11	Вимоги до сертифікації	Складна процедура сертифікації	Потрібно доопрацьовувати	Сертифікація можлива за певних умов	Сертифікація можлива без ускладнень	Сертифікація вже проведена	Сертифікація не потрібна

Таблиця 5.2 – Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-11	Низький
12-22	Нижче середнього
23-33	Середній
34-44	Вище середнього
45-55	Високий

В таблиці 5.3 наведено результати оцінювання експертами комерційного потенціалу розробки.

Таблиця 5.3 – Результати оцінювання комерційного потенціалу розробки

Критерії	Експерт 1	Експер 2	Експерт 3
	Бали, виставлені експертами		
1. Достовірність концепції	4	5	5
2. Захист від сучасних кіберзагроз	5	5	5
3. Унікальність розробки	4	4	4
4. Потенціал впровадження на ринок	5	4	5
5. Конкуренція	4	3	4
6. Витрати на реалізацію	4	4	4
7. Термін окупності	4	5	4
8. Потреба у фахівцях	4	4	5
9. Час реалізації	5	5	5
10. Ризики впровадження	4	4	4
11. Вимоги до сертифікації	5	5	5
Сума балів	48	48	50
Середнє арифметичне	48.67		

На основі проведеного експертного оцінювання було визначено інтегральний показник комерційного потенціалу розробленого програмного забезпечення, який становить 48.67 бали. Відповідно до шкали рівнів комерційної привабливості, це значення належить до категорії “високий”. Такий результат свідчить про перспективність впровадження створеного API у

практичну діяльність підприємств та організацій, що використовують системи управління завданнями. Високі бали за критеріями технічної здійсненності, захищеності, короткого часу реалізації та відсутності значних ризиків підтверджують, що розробка має значні конкурентні переваги та може бути комерційно успішною. Отже, результати аналізу підтверджують економічну доцільність подальшої розробки та впровадження системи.

5.2 Прогнозування витрат на виконання наукової роботи та впровадження її результатів

Витрати, пов’язані з виконанням науково-дослідної роботи зі створення API для системи управління завданнями, включають такі статті: оплата праці, нарахування на заробітну плату, витрати на матеріали, витрати на енергію, програмне забезпечення, амортизацію та накладні витрати. Усі розрахунки виконані відповідно до методичних рекомендацій.

1. Розрахунок основної заробітної плати дослідників $З_о$

Основна заробітна плата визначається за формулою:

$$З_0 = \frac{М}{Т_р} * t \text{ (грн)}, \quad (5.1)$$

де M – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

T_P – число робочих днів в місяці; приблизно 20-24 дні;

t – число робочих днів роботи дослідника.

Для розробки API були залучені два фахівці: інженер-програміст та backend-розробник.

Таблиця 5.3 – Основна заробітна плата дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Кількість робочих днів	Витрати на заробітну плату, грн
Інженер-програміст	22000	1100	20	22000
Backend-розробник	28000	1400	20	28000
Всього	-	-	-	50000

Основна заробітна плата розробників становить **50 000 грн**.

2. Додаткова заробітна плата $З_д$

Додаткова зарплата становить 10% від основної заробітної плати дослідникам та робітникам. Оскільки у працівниках немає потреби, то у цьому випадку це 10% від основна заробітна плата дослідників:

$$З_д = (З_о + З_р) * \frac{Н_{дод}}{100\%} \quad (5.2)$$

$$З_д = 0.1 \cdot 50000 = 5000 \text{ (грн)}.$$

Додаткова заробітна плата становить **5 000 грн**.

3. Нарахування на заробітну плату $Н_{ЗП}$

Розрахунок нарахувань на заробітну плату для дослідників і працівників, які були залучені до виконання цього етапу робіт, здійснюється за такою формулою:

$$Н_{ЗП} = (З_о + З_д) * \frac{\beta}{100}, \quad (5.3)$$

де $З_о$ – основна заробітна плата розробників, грн.;

$З_д$ – додаткова заробітна плата всіх розробників та робітників, грн.;

β – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, %.

Ця діяльність належить до бюджетної сфери, тому розмір єдиного внеску на загальнообов'язкове державне соціальне страхування становитиме 22%. У такому разі:

$$Н_{ЗП} = (50000 + 5000) * 0.22 = 12100 \text{ грн}$$

Нарахування становлять **12 100 грн**.

4. Витрати на матеріали К

Матеріали використовуються мінімально (канцелярія, носії даних, стендові матеріали). Розрахунок виконується за формулою:

$$К = \sum_{i=1}^{n} Н_i * Ц_i * К_i , \quad (5.4)$$

де $Н_i$ – кількість комплектуючих *i*-го виду, шт.;

$Ц_i$ – покупна ціна комплектуючих *i*-го найменування, грн.;

$К_i$ – коефіцієнт транспортних витрат (1,1…1,15).

Таблиця 5.4 – Комплектуючі, що використані на розробку

Найменування матеріалу	Ціна за одиницю, грн	Витрачено, шт.	Вартість витраченого матеріалу, грн
Папір	150	1	150
Маркери/ручки	15	4	60
Flash-носій	180	1	180
Всього	-	-	390
З урахуванням коефіцієнта транспортних витрат (1.15)	-	-	449

Витрати на матеріали становлять **449 грн**.

5. Програмне забезпечення для виконання наукової роботи включає витрати на розробку та придбання спеціальних програмних засобів і

програмного забезпечення, необхідного для проведення досліджень. Для написання магістерської роботи використовувались:

IntelliJ IDEA Community Edition (безкоштовно)

PostgreSQL (безкоштовно)

Postman (безкоштовно)

GitHub (безкоштовно

Витрати на ПЗ 0 грн.

6. Амортизація обладнання А.

Дані відрахування розраховують по кожному виду обладнання, приміщенням тощо.

$$A = \frac{Ц * Т}{Т_{кор} * 12}, \quad (5.5)$$

де Ц – балансова вартість даного виду обладнання (приміщень), грн.;

$Т_{кор}$ – час користування;

Т – термін використання обладнання (приміщень), цілі місяці.

Згідно пункта 137.3.3 Податкового кодекса амортизація нараховується на основні засоби вартістю понад 2500 грн. Задіяний робочий комп'ютер вартістю 45 000 грн, строк служби – 2 роки.

$$А = \frac{45000 \cdot 1}{2 \cdot 12} = 1875 \text{ (грн)}$$

Амортизація за час виконання дослідження становить **1875 грн**.

7. Витрати на електроенергію $В_е$

До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

$$В_е = \sum_{i=1}^{n} \frac{W_{yt} \cdot t_i \cdot Ц_е \cdot К_{впі}}{\eta_і}, \quad (5.6)$$

де W_{yt} – встановлена потужність обладнання на певному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_е$ – вартість 1 кВт-години електроенергії, грн;

$К_{впі}$ – коефіцієнт, що враховує використання потужності, $К_{впі}$ <1;

$\eta_і$ – коефіцієнт корисної дії обладнання, $\eta_і$ <1.

Для написання магістерської роботи використовується персональний комп'ютер для якого розрахуємо витрати на електроенергію.

$$В_е = \frac{0.25 \cdot 160 \cdot 6{,}158 \cdot 0.8}{0{,}9} = 218.95 \text{ (грн)}.$$

Витрати на електроенергію становлять **218.95 грн**.

8. Накладні витрати $В_{нзв}$

Накладні витрати охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати Внзв можна прийняти як (100…150) % від суми основної заробітної плати розробників та робітників, які виконували дану МКР, тобто:

$$В_{нзв} = \left(З_о + З_р\right) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.7)$$

де $Н_{нзв}$ – норма нарахування за статтею «Інші витрати».

Приймаємо як 100% від основної зарплати:

$$В_{нзв} = 50000 \cdot \frac{100}{100\%} = 50000 (\text{грн}).$$

Накладні витрати становлять **50000 грн**

9. Загальні прямі витрати ЗВ

Сума всіх попередніх статей витрат дає витрати, які безпосередньо стосуються даного розділу МКР

В = 50000+5000+12100+449+1875+218.95+50000=119642.95 (грн).

Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР здійснюється за формулою:

$$ЗВ = \frac{В}{\eta}, \quad (5.8)$$

де η – коефіцієнт, який характеризує етап (стадію) виконання науководослідної роботи. Так, якщо науково-технічна розробка знаходиться на стадії: науково-дослідних робіт, то η =0,1; технічного проектування, то η =0,2; розробки конструкторської документації, то η =0,3; розробки технологій, то η =0,4; розробки дослідного зразка, то η =0,5; розробки промислового зразка, то η =0,7; впровадження, то η =0,9.

$$ЗВ = \frac{119642.95}{0,7} = 170918.5 \text{ грн}$$

Загальні прямі витрати становлять **170918.5 грн**.

5.3 Прогнозування комерційних ефектів від реалізації результатів розробки

У ринкових умовах основним позитивним результатом для потенційного інвестора від впровадження нового програмного забезпечення є збільшення чистого прибутку. Розроблене API для системи управління завданнями може бути інтегроване у роботу підприємств різних масштабів, що створює підстави для комерціалізації протягом кількох років після впровадження. У цьому підрозділі здійснено кількісний прогноз комерційної вигоди, яку може отримати підприємство внаслідок використання результатів наукової роботи.

Передбачається, що модернізований продукт дозволить підприємству збільшити кількість користувачів завдяки покращенню функціональності, зниженню операційних витрат та покращенні автоматизації. Основні вихідні дані для розрахунку такі:

Збільшення кількості споживачів продукту в аналізовані періоди часу завдяки покращенню його характеристик (ΔN):

1-й рік – 40 користувачів;

2-й рік – 60 користувачів;

3-й рік – 85 користувачів.

Базова кількість користувачів до впровадження (N) – 300;

Ціна продукту до впровадження ($Ц_0$) – 15 000 грн;

Покращення вартості продукту ($ΔЦ_0$) – +700 грн;

Можливе збільшення чистого прибутку потенційного інвестора для кожного з трьох років, протягом яких очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, розраховується за формулою:

$$\Delta П_i = (\pm \Delta Ц_б * N + Ц_б * \Delta N)_i * \lambda * \rho * (1 - \frac{\vartheta}{100}) \qquad (5.9)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2025 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0,8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту. Приймемо $\rho = 30\%$;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2025 році $\vartheta = 18\%$;

1-й рік: $\Delta П_1 = (700 \times 300 + 15000 \times 40) \times 0,83 \times 0,3 \times (1 - 0.18) = 166191.0$ грн

2-й рік: $\Delta П_2 = (700 \times 300 + 15000 \times (40 + 60)) \times 0,83 \times 0,3 \times (1 - 0.18) = 350049,2$ грн

3-й рік: $\Delta П_3 = (700 \times 300 + 15000 \times (40 + 60 + 85)) \times 0,83 \times 0,3 \times (1 - 0.18) = 613452,3$ грн

$$\Delta П_{заг} = 166191.0 + 350049.2 + 613452.3 = 1129692.5 \text{ грн}$$

Проведені розрахунки демонструють, що впровадження розробленого API для системи управління завданнями забезпечує значне зростання чистого прибутку підприємства. Прогнозований сумарний комерційний ефект за три роки становить **1 129 692,5 грн**, що значно перевищує прогнозовані витрати на створення та впровадження розробки (170 691 грн). Це свідчить про високу економічну доцільність реалізації проєкту та підтверджує його інвестиційний потенціал.

5.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Основними критеріями, що визначають доцільність фінансування науково-технічної розробки, є абсолютна ефективність інвестицій, відносна ефективність та термін їх окупності. На першому етапі визначається теперішня вартість інвестицій (PV), які необхідно вкласти у впровадження та комерціалізацію програмного продукту.

Теперішня вартість інвестицій обчислюється за формулою:

$$\text{PV} = k_{\text{інв}} * \text{ЗВ} \tag{5.10}$$

$k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$=3;

ЗВ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо 170918.5 грн.

$$\text{PV} = 3 * 170918.5 \ = 512755.5 \text{ грн.}$$

Тоді абсолютний економічний ефект $\text{Е}_{\text{абс}}$ або чистий приведений дохід для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$\text{Е}_{\text{абс}} = (\text{ПП} - \text{PV}) \tag{5.11}$$

де ПП – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, грн;

PV – теперішня вартість початкових інвестицій, грн.

Приведена вартість всіх чистих прибутків ПП розраховується за формулою:

$$ПП = \sum_1^T \frac{\Delta П_1}{(1+\tau)^t} \quad (5.12)$$

де ΔПі – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

Т – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,05…0,15;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$$ПП = \frac{166191}{(1+0{,}1)^1} + \frac{350049.2}{(1+0{,}1)^2} + \frac{613452.3}{(1+0{,}1)^3} = 901713.5 \text{ грн.}$$

$$Е_{абс} = 901713.5 - 512073 = 389640.5 \text{грн.}$$

Оскільки $E_{абс} > 0$, впровадження розробки є економічно обґрунтованим і генерує чистий прибуток.

Внутрішня економічна дохідність інвестицій $Е_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки, розраховується за формулою:

$$Е_в = \sqrt[Т_ж]{1 + \frac{Е_{абс}}{PV}} - 1 \quad (5.13)$$

де $Е_{абс}$ – абсолютний економічний ефект вкладених інвестицій, грн;

PV – теперішня вартість початкових інвестицій, грн;

$Т_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, роки.

$$Е_в = \sqrt[3]{1 + \frac{389640.5}{512073}} - 1 = 0.205 = 20.5\%$$

Порівняємо Ев з мінімальною (бар'єрною) ставкою дисконтування τ_{min}, яка визначає мінімальну дохідність, нижче якої інвестиції не будуть здійснюватися.

У загальному вигляді мінімальна (бар'єрна) ставка дисконтування τ_{min} визначається за формулою:

$$\tau_{min} = d + f \quad (5.14)$$

d – середньозважена ставка за депозитними операціями в комерційних банках;

f – показник, що характеризує ризикованість вкладень; $f = 0,4$.

$d = 0,2$.

$$\tau_{min} = 0,2 + 0,4 = 0,6$$

Це означає, що хоча проєкт є прибутковим, його дохідність нижча за бар'єрну ставку, що робить інвестиційний ризик підвищеним. Така ситуація типова для програмних продуктів без активної маркетингової кампанії або з малою початковою аудиторією. Однак при можливому масштабуванні або залученні додаткових ринків дохідність може зрости.

Далі розраховуємо період окупності інвестицій $Т_{ок}$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$Т_{ок} = \frac{1}{Е_в} \quad (5.15)$$

де $Е_в$ – внутрішня економічна дохідність вкладених інвестицій.

$$T_{ок} = \frac{1}{0.205} = 4{,}87$$

Оскільки **Ток > 3 років**, розробка окупається повільніше за рекомендований інвестиційний горизонт. Проте довгостроковий чистий прибуток залишається значним, а очікуваний ріст ринку систем управління завданнями може покращити показники ефективності.

5.5 Висновки до розділу

У ході виконання економічної оцінки розробки було проведено комплексний аналіз комерційного потенціалу програмного продукту, прогнозування витрат на його створення та впровадження, визначення можливих комерційних ефектів, а також розрахунок ефективності інвестицій та терміну їх окупності. Зведені результати дають можливість об’єктивно оцінити доцільність фінансування та перспективи використання створеного API для системи управління завданнями.

Проведений аналіз показав, що загальна сума витрат на виконання дослідження та підготовку розробки до впровадження становить **170 918.5 грн**, а прогнозована теперішня вартість інвестицій – **512 755.5 грн**. Передбачений приріст чистого прибутку протягом трирічного періоду досягає **901 713,5 грн**, що забезпечує позитивний абсолютний економічний ефект у розмірі **389 640,5 грн**. Це означає, що навіть за консервативного сценарію проєкт здатний приносити реальний фінансовий результат.

Разом із тим, розрахована внутрішня економічна дохідність становить **20,5%**, що нижче бар’єрної ставки дисконтування на рівні 60%. Це свідчить про підвищений інвестиційний ризик і необхідність ретельної маркетингової

стратегії та подальшого розвитку продукту для розширення ринку. Термін окупності інвестицій дорівнює **4,87 року**, тобто перевищує рекомендовані три роки, однак усе ще є прийнятним для високотехнологічних IT-рішень, що орієнтовані на довгострокову комерціалізацію.

У процесі аналізу були визначені ключові переваги розробки:

– невисока собівартість впровадження порівняно з аналогічними корпоративними системами;

– гнучкість архітектури та можливість масштабування, що позитивно впливає на довгострокову рентабельність;

– повна автономність серверної частини, що дозволяє інтегрувати продукт у різні бізнес-процеси без дорогих модифікацій;

– наявність сучасних механізмів безпеки, включаючи JWT-автентифікацію та контроль доступу;

– можливість подальшого розвитку функціональності без суттєвих додаткових витрат.

До недоліків належать:

– нижча, ніж бажано, внутрішня рентабельність інвестицій через обмежений прогнозований ринок;

– порівняно тривалий термін окупності, характерний для програмних продуктів із повільним зростанням користувацької бази;

– необхідність вкладень у маркетинг для збільшення попиту та скорочення терміну повернення інвестицій.

Попри вказані недоліки, результати дослідження переконливо демонструють, що розробка API для системи управління завданнями є економічно доцільною. Проєкт приносить позитивний фінансовий ефект, має значний потенціал масштабування, низьку собівартість підтримки та відповідає сучасним технічним і безпековим вимогам. За умови подальшої комерційної оптимізації, впровадження гнучких моделей ліцензування та активного просування на ринок, продукт може стати фінансово успішним і

конкурентоспроможним рішенням у сфері управління завданнями та планування робочих процесів.

ВИСНОВКИ

У процесі виконання магістерської кваліфікаційної роботи було розроблено серверний програмний продукт API для системи управління завданнями, що реалізує REST-архітектуру та орієнтований на використання в багатокористувацьких середовищах. У роботі послідовно виконано повний цикл створення серверної частини сучасної інформаційної системи від аналізу предметної області та порівняння існуючих рішень до проєктування архітектури, реалізації функціональності, тестування та економічного обґрунтування ефективності впровадження.

Було проведено дослідження ринку систем управління завданнями, проаналізовано можливості інструментів Trello, Asana, Jira Software та ClickUp, що дозволило виявити їхні сильні сторони та недоліки. На основі цього аналізу обґрунтовано доцільність розробки власного API з акцентом на гнучкість, розширюваність, безпеку та можливість інтеграції в різні корпоративні рішення. Окремо визначено, яким чином створена система усуває обмеження існуючих продуктів завдяки відкритій архітектурі та незалежності від клієнтської частини.

У ході реалізації було створено доменну модель даних, побудовано структуру бази даних PostgreSQL та реалізовано всі основні сутності системи. Розроблено сервісний рівень із чітким розмежуванням відповідальності згідно з принципами SOLID, впроваджено DTO для забезпечення безпечної та контрольованої передачі даних. Реалізовано повноцінний набір REST-контролерів, які забезпечують CRUD-операції над користувачами, завданнями та списками ToDo.

Особлива увага приділена безпеці системи: впроваджено механізм автентифікації та авторизації на основі JWT, розроблено фільтр перехоплення запитів, налаштовано політику доступу на основі ролей, що забезпечує захищену взаємодію клієнтів із сервером. Модуль безпеки підтвердив стабільність роботи

під час тестування, а структуру взаємодії з токенами інтегровано у всі ключові сценарії користувацької діяльності.

Було проведено комплексне тестування API, включаючи модульні тести із використанням JUnit та інтеграційні перевірки за допомогою Postman. Окремо оцінено продуктивність системи за реальними показниками часу відповіді на ключові операції. Отримання списку To-Do показало середній час відповіді 13.8 мс, отримання списку користувачів також показало низький середній час – 12.8 мс, запит отримання всіх задач є більш навантаженим, що пояснює його вищий середній час – 19 мс. Операції створення нових ресурсів займають дещо більше часу, ніж читання. Створення To-Do має середній час 17.6 мс, тоді як створення задачі – 14.6 мс. Найбільш ресурсоємною CRUD-операцією виявилося оновлення задачі, середній час якого становив 28.8 мс. Результати підтвердили стабільність роботи системи та готовність до обробки навантаження.

В економічній частині роботи виконано прогнозування витрат на розробку системи, оцінено комерційний потенціал API, розраховано можливі прибутки від комерціалізації та визначено економічну ефективність упровадження продукту. Сума витрат на виконання дослідження та підготовку розробки до впровадження становить **170 918.5 грн**, а прогнозована теперішня вартість інвестицій – **512 755.5 грн**. Передбачений приріст чистого прибутку протягом трирічного періоду досягає **901 713,5 грн**, що забезпечує позитивний абсолютний економічний ефект у розмірі **389 640,5 грн**. Отримані результати свідчать, що попри тривалий термін окупності, API має позитивний абсолютний економічний ефект і може бути комерційно перспективним за умови подальшого розвитку та маркетингової підтримки.

Розроблене API має значний практичний потенціал: його можна використовувати як окремий програмний продукт, інтегрувати у корпоративні системи чи розширювати новими модулями, включаючи веб-інтерфейс, мобільний клієнт або аналітичні інструменти. Створена архітектура є гнучкою, масштабованою та відповідає сучасним стандартам розробки серверних застосунків.

Поставлені у магістерській кваліфікаційній роботі завдання виконано повністю. Результати дослідження та реалізованого програмного забезпечення підтверджують доцільність обраного підходу, ефективність розробленого API та перспективи його практичного використання в реальних умовах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Галіброда А. С., Кабачій В. В., Створення REST API для таск-менеджера: дослідження архітектурних рішень. *Молодь в науці: дослідження, проблеми, перспективи (МН-2025)*. Вінниця: Вінницький національний технічний університет, 2025.

2. Галіброда А. С., Кабачій В. В., Створення REST API для таск-менеджера: дослідження практичні результати та аналіз. *Молодь в науці: дослідження, проблеми, перспективи (МН-2026)*. Вінниця: Вінницький національний технічний університет, 2025.

3. Burke R. Project Management: Planning and Control Techniques. Wiley, 2018.

4. S. Rahman, M. Rahim, A comparative analysis of task management tools for agile software development. *International Journal of Advanced Computer Science and Applications*, 2020.

5. L. Feng, Y. Jiang, Machine learning in intelligent task scheduling systems. *IEEE Access*, 2022.

6. M. Patel, S. Singh, Comparative Study of Modern Task Management Applications. *International Journal of Computer Applications*, 2021.

7. F. Almeida, Evaluating Kanban-based tools for task management: Trello case study. *Procedia Computer Science*, 2020.

8. L. Thompson, Adoption challenges in collaborative project management systems: the Asana experience. *Information Systems Frontiers*, 2022.

9. Atlassian Corporation Plc – Atlassian Documentation. Performance and Scale Testing Overview for Jira Software. URL https://confluence.atlassian.com/adminjiraserver103/performance-and-scale-testing-1489808376.html (дата звернення 30.09.2025).

10. D. G. Morales, Automation trends in task management: An analysis of ClickUp platform efficiency. *IEEE Access*, 2023.

11. E. Ahmad, R. Mahmud, Architectural Patterns for RESTful and GraphQL APIs in Cloud-based Task Management Systems. *Journal of Cloud Computing*, 2022.

12. N. Williams, REST API Design Patterns and Best Practices. *IEEE Software Engineering Review*, 2023.

13. K. Mishra, A. Kumar, Performance Evaluation of GraphQL vs REST for Large-Scale Applications. *ACM Transactions on Web Systems*, 2022.

14. R. Oliveira, F. Costa, gRPC Communication Efficiency in Microservice Architectures. *Journal of Internet Services and Applications*, 2023.

15. RedMonk Developer Survey, Programming Language Rankings 2024. *RedMonk Research Report*, 2024.

16. G. Chatzoglou, D. Macaulay, A review of task management systems in software development. *Journal of Systems and Software*, 2021.

17. A. Yilmaz, Designing RESTful APIs for Scalable Web Applications. *IEEE Transactions on Software Engineering*, 2022.

18. HackerNoon. Using Postgres effectively in Spring Boot applications. URL: https://hackernoon.com/using-postgres-effectively-in-spring-boot-applications (дата звернення 13.11.2025).

19. R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures (REST), PhD Thesis, UC Irvine, 2000.

20. R. C. Martin. Clean Architecture, Prentice Hall, 2017.

21. G. E. Krasner, S. T. Pope. A Description of the Model–View–Controller User Interface Paradigm. *Journal of Object Oriented Programming*, 1988.

22. Fielding R. T., Taylor R. N. Architectural Styles and the Design of Network-based Software Architectures**.** *Irvine Technical Report*, University of California*,* 2000.

23. Elmasri R., Navathe S. *Fundamentals of Database Systems*, 7th ed., Pearson, 2016.

24. OpenAPI Initiative. OpenAPI Specification 3.1. URL: https://www.openapis.org/blog/2021/02/18/openapi-specification-3-1-released (дата звернення 18.11.2025).

25. Medium. Onu V., Implementing JWT authentication in a simple Spring Boot application with Java, Medium, URL: https://medium.com/@victoronu/implementing-jwt-authentication-in-a-simple-spring-boot-application-with-java-b3135dbdb17b (дата звернення 18.11.2025).

26. Apache Maven Project. Welcome to Apache Maven. URL: https://maven.apache.org/ (дата звернення 19.11.2025).

27. PostgreSQL Global Development Group. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення 19.11.2025).

28. Hibernate. Hibernate ORM. URL: https://hibernate.org/orm/ (дата звернення 19.11.2025).

29. Jakarta EE Foundation. Jakarta Persistence Specification, Version 3.1. URL: https://jakarta.ee/specifications/persistence/3.1/ (дата звернення 19.11.2025).

30. Project Lombok. Documentation. URL: https://projectlombok.org/contributing/contributing (дата звернення 19.11.2025).

31. GitHub – FasterXML. Jackson Databind & Annotations Reference. URL: https://github.com/FasterXML/jackson (дата звернення 19.11.2025).

32. Spring – Spring Security. Spring Security Architecture & JWT Integration Guide. URL: https://spring.io/projects/spring-security (дата звернення 19.11.2025).

33. Curity. JSON Web Token (JWT) Security Best Practices. URL: https://curity.io/resources/learn/jwt-best-practices/ (дата звернення 20.11.2025).

34. JUnit Team. JUnit 5 User Guide. URL: https://docs.junit.org/current/user-guide/ (дата звернення 20.11.2025).

35. Mockito Organization. Mockito Framework. URL: https://site.mockito.org/ (дата звернення 20.11.2025).

ДОДАТКИ

Додаток А (обов’язковий)

Технічне завдання

ЗАТВЕРДЖУЮ

Завідувач кафедри АІІТ

д.т.н., проф. Олег БІСІКАЛО

«17» жовтня 2025 року

ТЕХНІЧНЕ ЗАВДАННЯ

на магістерську кваліфікаційну роботу

«**РОЗРОБКА API ДЛЯ СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ІЗ ВИКОРИСТАННЯМ SPRING FRAMEWORK**»

08-31.МКР.002.02.000 ТЗ

Керівник роботи:

к.т.н., доц. каф. АІІТ

Владислав КАБАЧІЙ

«16» жовтня 2025 р.

Виконавець:

ст. гр. 1ІСТ-24м

Анатолій ГАЛІБРОДА

«16» жовтня 2025 р.

Вінниця ВНТУ – 2025

1. Назва та галузь застосування

Розробка API для системи управління завданнями із використанням Spring Framework.

Інформаційні системи та технології. Системи управління проєктами та бізнес-процесами. Розробка та інтеграція серверних REST-API для багатокористувацьких застосунків, управління задачами та планування робочих процесів.

2. Підстава для розробки

Розробку системи здійснювати на підставі наказу по університету № 313 від 24 вересня 2025 року та завдання до магістерської кваліфікаційної роботи, складеного та затвердженого кафедрою «Автоматизації та інтелектуальних інформаційних технологій»

3. Мета та призначення розробки

Метою роботи є створення серверної системи управління завданнями на основі архітектури REST, яка забезпечить:

- Надання стандартизованого API для багатокористувацької роботи з даними, включаючи управління користувачами, проєктами, списками завдань (ToDo) та окремими задачами;
- Забезпечення захищеної автентифікації та авторизації за допомогою JWT-токенів з рольовою моделлю доступу;
- Забезпечення можливості створення, оновлення, видалення та фільтрації завдань і проєктів, включаючи перевірку прав доступу до кожної операції;
- Розробку інфраструктури для взаємодії з клієнтськими застосунками через чітко визначені REST-ендпоінти;
- Проведення тестування роботи API за допомогою Postman, JUnit та Mockito, із подальшим аналізом продуктивності.

Призначення системи: забезпечення надійного, масштабованого та безпечного серверного програмного забезпечення для управління завданнями в навчальних, корпоративних або командних середовищах, а також створення інтеграційної основи для сторонніх клієнтських застосунків, що працюватимуть із цим API.

4. Джерела розробки

1. Галіброда А. С., Кабачій В. В., Створення REST API для таск-менеджера: дослідження архітектурних рішень. *Молодь в науці: дослідження, проблеми, перспективи (МН-2025)*. Вінниця: Вінницький національний технічний університет, 2025.
2. S. Rahman, M. Rahim, A comparative analysis of task management tools for agile software development. *International Journal of Advanced Computer Science and Applications*, 2020.
3. M. Patel, S. Singh, Comparative Study of Modern Task Management Applications. *International Journal of Computer Applications*, 2021.
4. F. Almeida, Evaluating Kanban-based tools for task management: Trello case study. *Procedia Computer Science*, 2020.
5. L. Thompson, Adoption challenges in collaborative project management systems: the Asana experience. *Information Systems Frontiers*, 2022.

5. Показники призначення

5.1. Основні технічні характеристики системи

Функціональні можливості:

5.1.1 Модуль управління користувачами:

- Реєстрація та автентифікація користувачів з використанням JWT-токенів;
- Отримання даних автентифікованого користувача;
- Рольова модель доступу (USER / ADMIN);
- Перегляд, оновлення та видалення профілю користувача.

5.1.2 Модуль управління ToDo (списками завдань):

- Створення нового ToDo для конкретного користувача;
- Оновлення назви, власника, списку співвиконавців;
- Видалення ToDo;
- Отримання переліку всіх ToDo;

- Перегляд ToDo одного користувача;
- Перевірка прав доступу (користувач може працювати тільки зі своїми ToDo).

5.1.3 Модуль управління задачами:

- Створення задачі з прив’язкою до ToDo;
- Оновлення даних задачі (назва, опис, дата, стан);
- Видалення задачі;
- Отримання переліку всіх задач або задач конкретного ToDo;
- Контроль доступу (користувач може редагувати лише задачі у своїх ToDo);

5.1.4 Модуль управління станами (State):

- Додавання нових станів (доступно лише ADMIN);
- Зміна стану задачі під час оновлення Task;
- Отримання всіх станів;

5.1.5 Модуль тестування та валідації:

- Unit-тести сервісного шару;
- Інтеграційне тестування;
- Ручне тестування API;
- Логування операцій.

5.2. Мінімальні системні вимоги

5.2.1 Апаратне забезпечення:

- Процесор: тактова частота не менше 2.5 GHz, рекомендовано 4+ ядра для стабільної роботи Spring Boot;
- Оперативна пам'ять: 8 GB RAM;
- Місце на диску: 2 GB для проєкту, залежностей та бази даних;
- Мережа: стабільне Інтернет-з'єднання для для роботи API (мінімум 10 Mbps).

5.2.2 Програмне забезпечення:

- Операційна система: Windows 10/11, Linux (Ubuntu 20.04+), macOS 10.15+;

- Java: версія 17 або новіша;
- СУБД**:** PostgreSQL 14/15/16;
- Сервіс побудови проєкту: Maven 3.8+;
- Додаткові інструменти: IntelliJ IDEA, Postman, H2 Database (для тестів).

5.3. Вхідні дані

Система обробляє такі типи даних:

- Дані користувачів: ім’я, прізвище, email, password (захешований), роль (USER / ADMIN);
- Дані ToDo (списків завдань): назва, власник, dateCreated, список співвиконавців;
- Дані задач (Task): назва, опис, дата створення, дедлайн, стан (State), ToDo-власник;
- Дані станів (State): назва стану (new, in progress, completed тощо).

Формат даних JSON, передається через HTTP методи (GET, POST, PUT, DELETE).

5.4 Результати роботи програми

5.4.1 Робота API:

Система забезпечує повністю функціональний набір REST-ендпоінтів:

- /api/auth/login, /api/auth/register – видача JWT-токенів;
- /api/users – управління користувачами;
- /api/todos – управління ToDo елементами;
- /api/tasks – управління задачами;
- /api/states – управління станами задач.

5.4.2 Структура відповідей:

Відповіді приходять у JSON форматі:

- { "token": "eyJhbGciOiJIUzI1NiJ9..." };
- {

  "id": 12,

  "name": "Finish API",

  "priority": "HIGH",

"todo_id": 3,

"state_id": 1

}.

6. Економічні показники

До економічних показників входять:

- витрати на розробку – до 200 тис. грн.
- узагальнений коефіцієнт якості розробки – більше 4-х
- термін окупності – до 5х років

7. Стадії розробки:

1. Розділ 1 «Аналіз предметної області та постановка задачі» має бути виконаний до 05.10.2025 р.

2. Розділ 2 «Проєктування API для системи управління завданнями» має бути виконаний до 25.10.2025 р.

3. Розділ 3 «Реалізація API на базі Spring Framework» має бути виконаний до 10.11.2025 р.

4. Розділ 4 «Експериментальне дослідження та оцінка ефективності» має бути виконаний до 20.10.2025 р.

5. Розділ 5 «Економчна частина» має бути виконаний до 01.12.2025 р

8. Порядок контролю та приймання

1. Рубіжний контроль провести до 14.11.2025.

2. Попередній захист магістерської кваліфікаційної роботи провести до 02.12.2025.

3. Захист магістерської кваліфікаційної роботи провести в період з 15.12.2025 р. до 19.12.2025 р.

Додаток Б (обов’язковий)

Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА API ДЛЯ СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ІЗ ВИКОРИСТАННЯМ SPRING FRAMEWORK

Діаграма класів

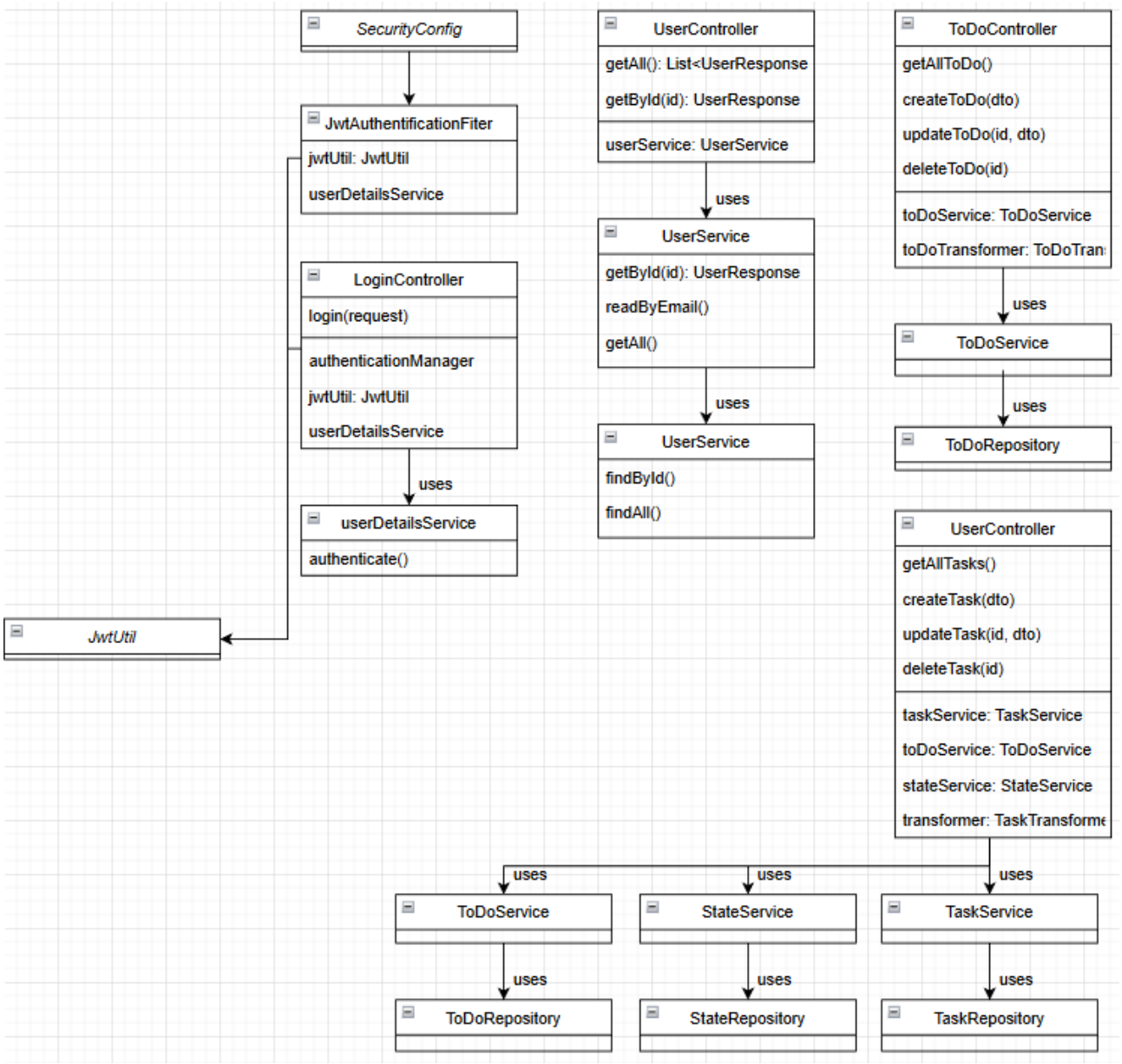

Рисунок Б.1 – UML Діаграма класів

Діаграми послідовності

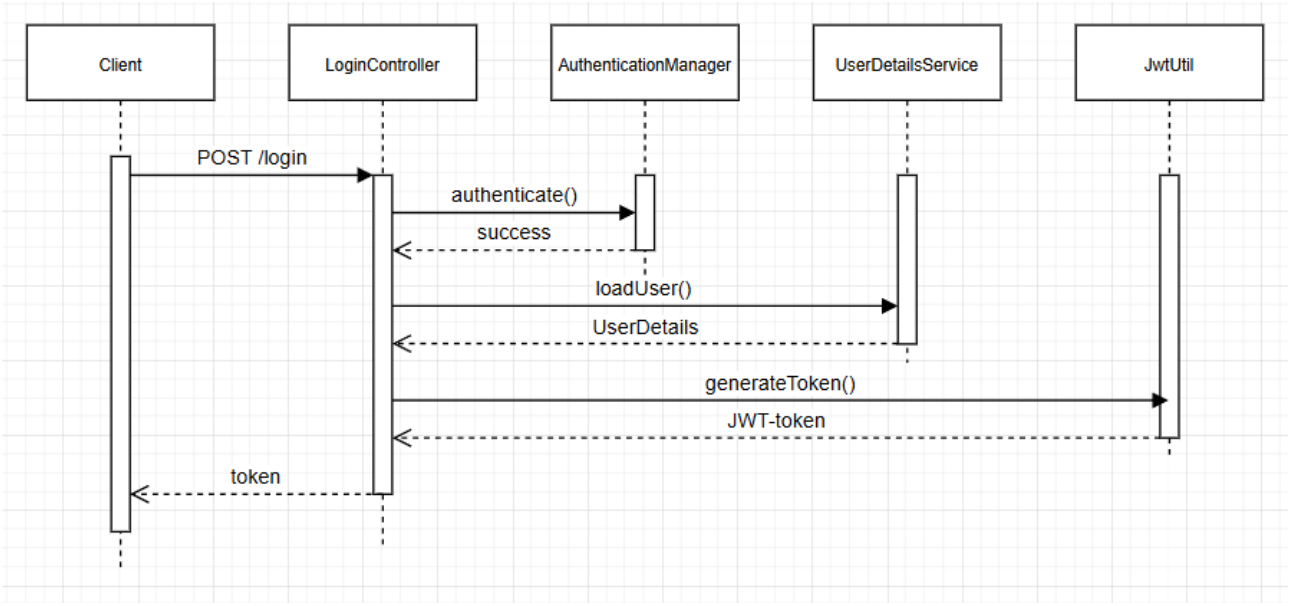

Рисунок Б.2 – UML Сценарій автентифікації та отримання JWT-токена

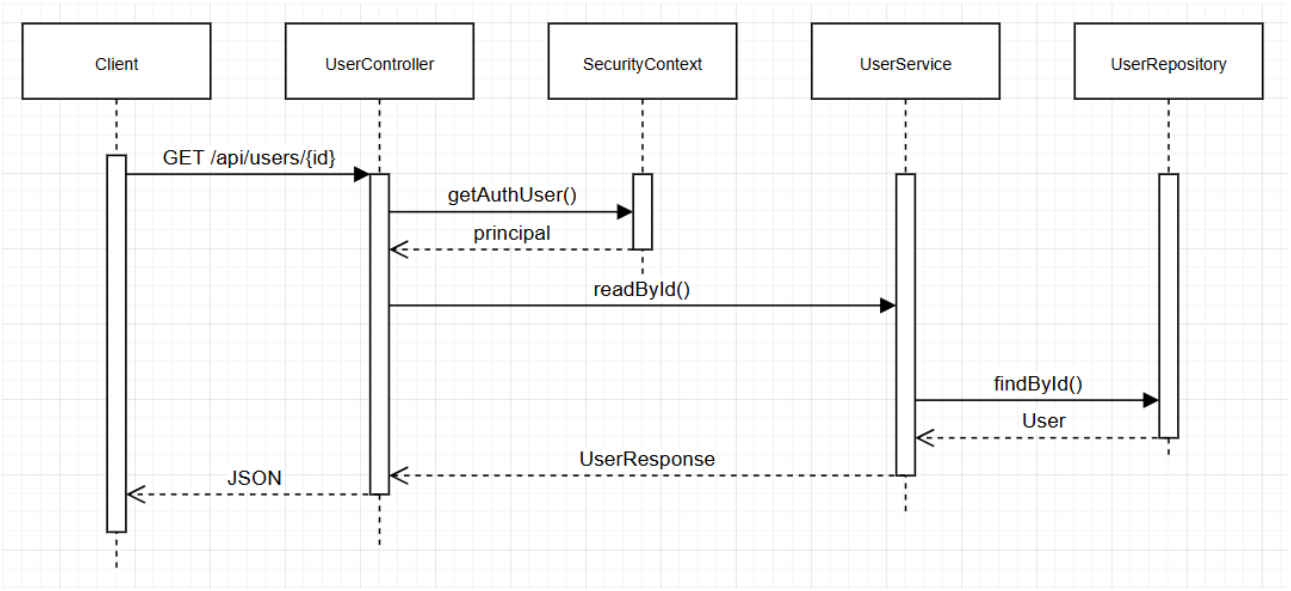

Рисунок Б.3 – UML Сценарій отримання інформації про користувача

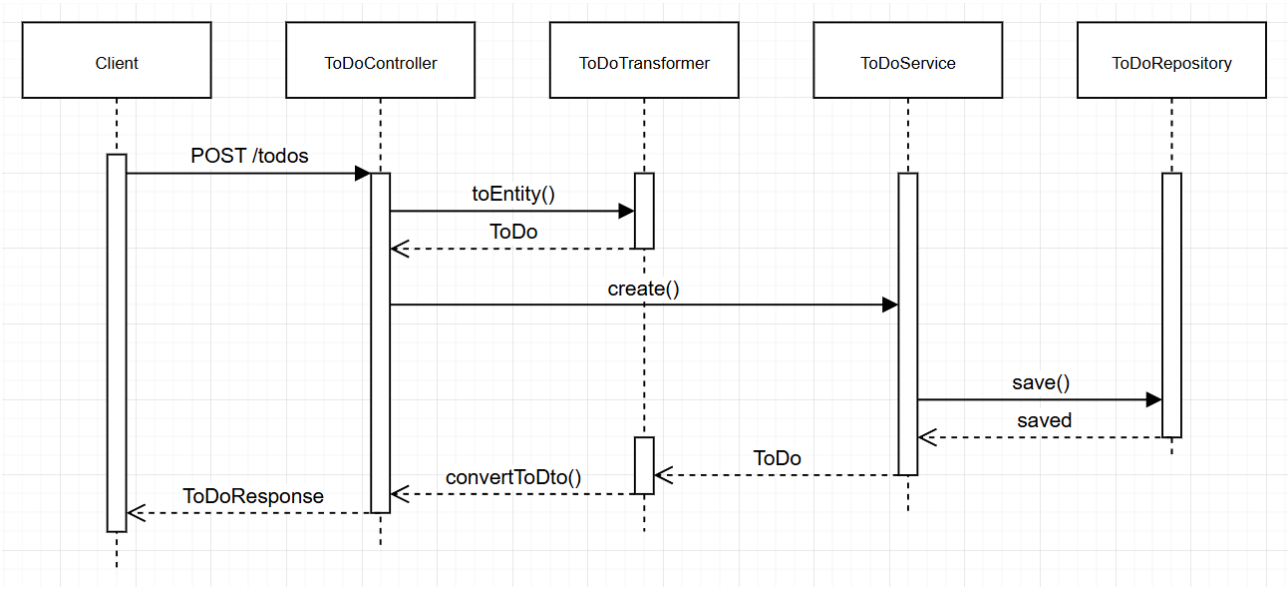

Рисунок Б.4 – UML Сценарій створення спискаму завдань

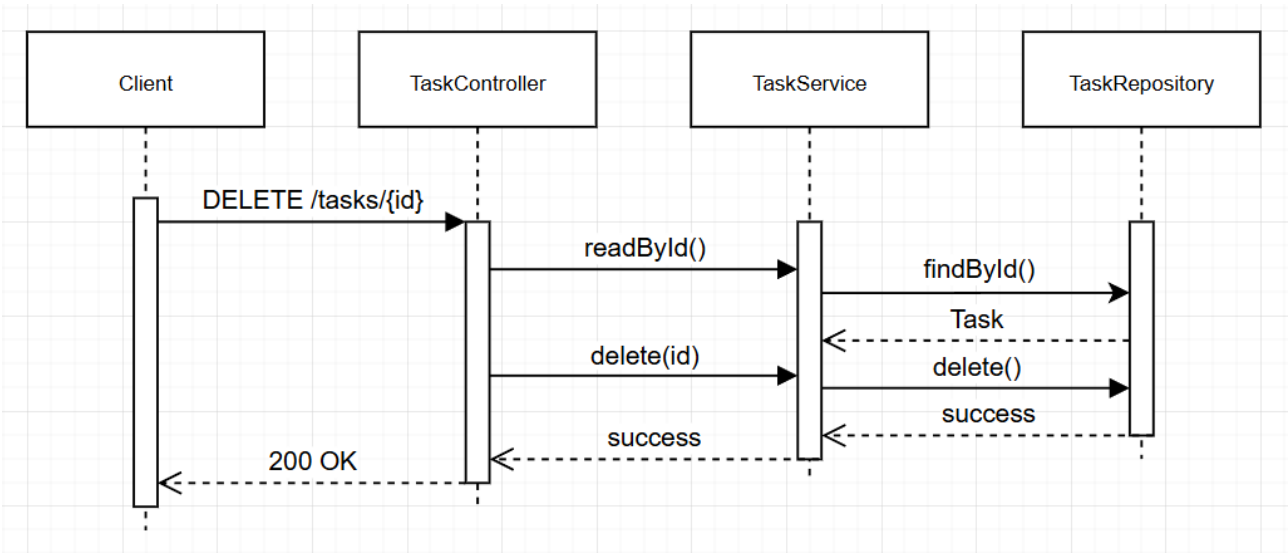

Рисунок Б.5 – UML Сценарій видалення задачі

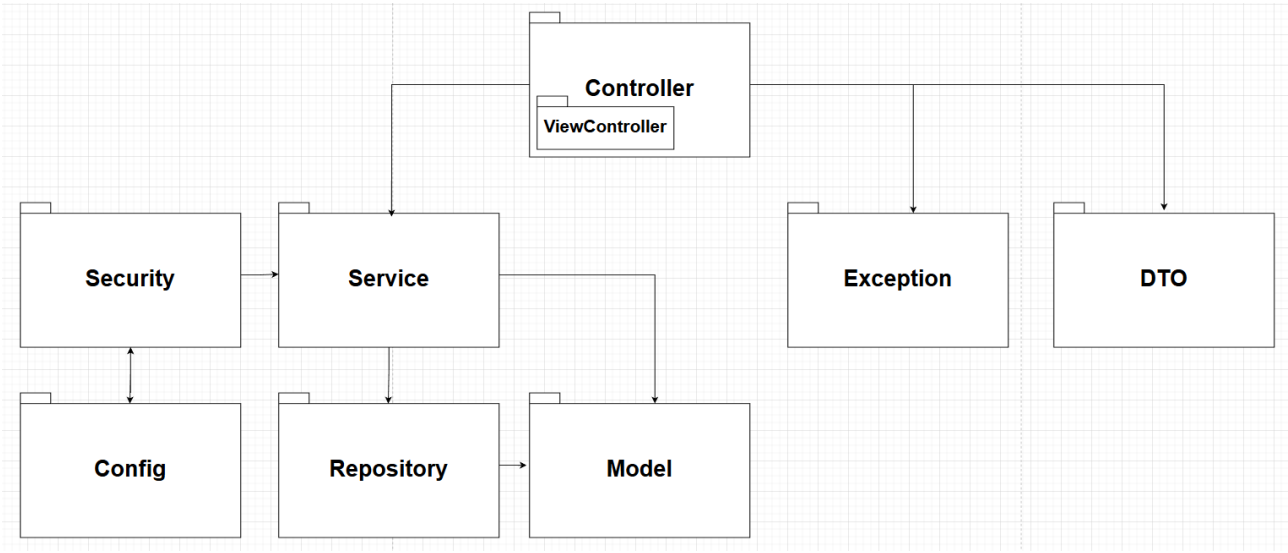

Рисунок Б.6 – UML Діаграма пакетів

Додаток В (обов’язковий)

Лістинг програми

```
package com.softserve.itacademy.todolist.config;

import com.softserve.itacademy.todolist.security.JwtAuthenticationFilter;
import com.softserve.itacademy.todolist.service.CustomUserDetailsService;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.Authent
icationConfiguration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationF
ilter;

@Configuration
@RequiredArgsConstructor
public class SecurityConfig {
```

```
    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final CustomUserDetailsService userDetailsService;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests(a -> a
                .antMatchers("/api/auth/login", "/api/auth/register").permitAll()
                .antMatchers("/api/users").hasRole("ADMIN")
                .antMatchers("/api/users/**").hasAnyRole("ADMIN", "USER")
                .anyRequest().authenticated())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authenticationProvider(authenticationProvider())
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
```

```
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

```
package com.softserve.itacademy.todolist.controller;

import com.softserve.itacademy.todolist.security.JwtUtil;
import com.softserve.itacademy.todolist.service.CustomUserDetailsService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.*;

import java.util.Map;
```

```
@Slf4j
@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class LoginController {

    private final AuthenticationManager authenticationManager;
    private final JwtUtil jwtUtil;
    private final CustomUserDetailsService userDetailsService;

    @PostMapping("/login")
    public Map<String, String> login(@RequestBody Map<String, String>
request) {
        try {
            String username = request.get("username");
            String password = request.get("password");

            log.info("Attempting login with: {}", username);
            log.info("Raw password entered: {}", password);

            authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(username, password));
            UserDetails                    userDetails                    =
userDetailsService.loadUserByUsername(username);
            String token = jwtUtil.generateToken(userDetails.getUsername());

            return Map.of("token", token);
        } catch (AuthenticationException e) {
            log.error(e.getMessage());
```

```
                return Map.of("error", "Invalid username or password");
            }
        }
    }
    package com.softserve.itacademy.todolist.controller;

    import com.softserve.itacademy.todolist.dto.TaskRequestDto;
    import com.softserve.itacademy.todolist.dto.TaskResponseDto;
    import com.softserve.itacademy.todolist.model.State;
    import com.softserve.itacademy.todolist.model.Task;
    import com.softserve.itacademy.todolist.model.ToDo;
    import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
    import com.softserve.itacademy.todolist.service.StateService;
    import com.softserve.itacademy.todolist.service.TaskService;
    import com.softserve.itacademy.todolist.service.ToDoService;
    import com.softserve.itacademy.todolist.dto.TaskTransformer;
    import lombok.RequiredArgsConstructor;
    import org.springframework.security.core.Authentication;
    import org.springframework.security.access.prepost.PreAuthorize;
    import org.springframework.web.bind.annotation.*;
    import javax.persistence.EntityNotFoundException;
    import org.springframework.security.access.AccessDeniedException;


    import javax.validation.Valid;
    import java.util.List;
    import java.util.stream.Collectors;

    @RestController
```

```
@RequestMapping("/api/tasks")
@RequiredArgsConstructor
public class TaskController {

    private final TaskService taskService;
    private final ToDoService toDoService;
    private final StateService stateService;

    @GetMapping
    public List<TaskResponseDto> getAllTasks(Authentication authentication) {
        if (isAdmin(authentication)) {
            return taskService.getAll().stream()
                    .map(TaskTransformer::toResponseDto)
                    .collect(Collectors.toList());
        }
        return taskService.getAll().stream()
                .filter(task -> isOwner(authentication, task.getTodo()))
                .map(TaskTransformer::toResponseDto)
                .collect(Collectors.toList());
    }

    @PostMapping
    @PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
    public TaskResponseDto createTask(@Valid @RequestBody
TaskRequestDto requestDto, Authentication authentication) {
        ToDo todo = toDoService.readById(requestDto.getTodoId());
        if (todo == null) {
            throw new EntityNotFoundException("ToDo with ID " +
requestDto.getTodoId() + " not found.");
        }
```

```
            if (!isAdmin(authentication) && !isOwner(authentication, todo)) {
                throw new AccessDeniedException("Access denied. You can only add tasks to your own ToDo lists.");
            }

            State state = stateService.readById(requestDto.getStateId());
            if (state == null) {
                throw new EntityNotFoundException("State with ID " + requestDto.getStateId() + " not found.");
            }

            Task task = TaskTransformer.toEntity(requestDto, todo, state);
            Task createdTask = taskService.create(task);
            return TaskTransformer.toResponseDto(createdTask);
        }

        @PutMapping("/{id}")
        @PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
        public TaskResponseDto updateTask(@PathVariable Long id, @Valid @RequestBody TaskRequestDto requestDto, Authentication authentication) {
            Task existingTask = taskService.readById(id);
            if (existingTask == null) {
                throw new EntityNotFoundException("Task with ID " + id + " not found.");
            }

            ToDo todo = existingTask.getTodo();
            if (!isAdmin(authentication) && !isOwner(authentication, todo)) {
```

```
            throw new AccessDeniedException("Access denied. You can only update tasks in your own ToDo lists.");
        }

        State state = stateService.readById(requestDto.getStateId());
        if (state == null) {
            throw new EntityNotFoundException("State with ID " + requestDto.getStateId() + " not found.");
        }

        Task updatedTask = TaskTransformer.toEntity(requestDto, todo, state);
        updatedTask.setId(id);
        Task savedTask = taskService.update(updatedTask);
        return TaskTransformer.toResponseDto(savedTask);
    }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
    public void deleteTask(@PathVariable Long id, Authentication authentication) {
        Task task = taskService.readById(id);
        if (task == null) {
            throw new EntityNotFoundException("Task with ID " + id + " not found.");
        }

        ToDo todo = task.getTodo();
        if (!isAdmin(authentication) && !isOwner(authentication, todo)) {
            throw new AccessDeniedException("Access denied. You can only delete tasks from your own ToDo lists.");
```

```
            }

            taskService.delete(id);
        }

        private boolean isAdmin(Authentication authentication) {
            return authentication.getAuthorities().stream()
                    .anyMatch(authority ->
authority.getAuthority().equals("ROLE_ADMIN"));
        }

        private boolean isOwner(Authentication authentication, ToDo todo) {
            return todo.getOwner().getEmail().equals(authentication.getName());
        }
    }
    package com.softserve.itacademy.todolist.controller;

    import com.softserve.itacademy.todolist.dto.ToDoRequestDTO;
    import com.softserve.itacademy.todolist.dto.ToDoResponseDTO;
    import com.softserve.itacademy.todolist.dto.ToDoTransformer;
    import com.softserve.itacademy.todolist.model.ToDo;
    import com.softserve.itacademy.todolist.service.ToDoService;
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.security.access.prepost.PreAuthorize;
    import org.springframework.web.bind.annotation.*;

    import javax.validation.Valid;
    import java.util.List;
    import java.util.stream.Collectors;
```

```
@RestController
@RequestMapping("/api/todos")
public class ToDoController {

    @Autowired
    private ToDoService toDoService;

    @Autowired
    private ToDoTransformer toDoTransformer;

    @PostMapping
    @PreAuthorize("hasAnyRole('ROLE_USER','ROLE_ADMIN')")
    public ToDoResponseDTO createToDo(@RequestBody ToDoRequestDTO
requestDTO) {
        ToDo toDo = toDoTransformer.convertToEntity(requestDTO);
        ToDo savedToDo = toDoService.create(toDo);
        return toDoTransformer.convertToDto(savedToDo);
    }

    @GetMapping
    @PreAuthorize("hasAnyRole('ROLE_USER','ROLE_ADMIN')")
    public List<ToDoResponseDTO> getAllToDo() {
        List<ToDo> toDos = toDoService.getAll();
        return toDos.stream()
                .map(toDoTransformer::convertToDto)
                .collect(Collectors.toList());
    }

    @GetMapping("/{id}")
    @PreAuthorize("hasAnyRole('ROLE_USER','ROLE_ADMIN')")
```

```
    public ToDoResponseDTO getToDo(@PathVariable Long id) {
        ToDo toDo = toDoService.readById(id);
        return toDoTransformer.convertToDto(toDo);
    }

    @PutMapping("/{id}")
    @PreAuthorize("hasAnyRole('ROLE_USER','ROLE_ADMIN')")
    public ToDoResponseDTO updateToDo(@PathVariable Long id, @Valid
@RequestBody ToDoRequestDTO requestDTO) {
        ToDo existingToDo = toDoService.readById(id);
        toDoTransformer.updateEntity(existingToDo, requestDTO);
        ToDo updatedToDo = toDoService.update(existingToDo,
requestDTO.getOwnerId());
        return toDoTransformer.convertToDto(updatedToDo);
    }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasAnyRole('ROLE_USER','ROLE_ADMIN')")
    public void deleteToDo(@PathVariable Long id) {
        toDoService.delete(id);
    }
}
package com.softserve.itacademy.todolist.controller;

import com.softserve.itacademy.todolist.dto.UserResponse;
import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
import com.softserve.itacademy.todolist.model.User;
import com.softserve.itacademy.todolist.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    UserService userService;

    @GetMapping
    List<UserResponse> getAll() {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

        boolean isAdmin = authentication.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .anyMatch(role -> role.equals("ROLE_ADMIN"));

        if (isAdmin) {
            return userService.getAll().stream()
                .map(UserResponse::new)
```

```
                .collect(Collectors.toList());
        }

        String authenticatedEmail = authentication.getName();
        User authenticatedUser = userService.readByEmail(authenticatedEmail);
        if (authenticatedUser == null) {
            throw new NullEntityReferenceException("Authenticated user not
found");
        }

        return List.of(new UserResponse(authenticatedUser));
    }

    @GetMapping("/{user_id}")
    UserResponse getById(@PathVariable long user_id) {
        Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
        String authenticatedEmail = authentication.getName();

        User user = userService.readById(user_id);
        if (user == null) {
            throw new NullEntityReferenceException("User not found with id: " +
user_id);
        }

        boolean isAdmin = authentication.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .anyMatch(role -> role.equals("ROLE_ADMIN"));

        if (isAdmin || user.getEmail().equals(authenticatedEmail)) {
```

```
            return new UserResponse(user);
        }

        throw new NullEntityReferenceException("Access denied: You can only view your own profile.");
    }
}
package com.softserve.itacademy.todolist.dto;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.util.Objects;

public class TaskDto {
    private long id;

    @NotBlank(message = "The 'name' cannot be empty")
    private String name;

    @NotNull
    private String priority;

    @NotNull
    private long todoId;

    @NotNull
    private long stateId;

    public TaskDto() {
    }
```

```
    public TaskDto(long id, String name, String priority, long todoId, long
stateId) {
        this.id = id;
        this.name = name;
        this.priority = priority;
        this.todoId = todoId;
        this.stateId = stateId;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPriority() {
        return priority;
    }
```

```
    public void setPriority(String priority) {
        this.priority = priority;
    }

    public long getTodoId() {
        return todoId;
    }

    public void setTodoId(long todoId) {
        this.todoId = todoId;
    }

    public long getStateId() {
        return stateId;
    }

    public void setStateId(long stateId) {
        this.stateId = stateId;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        TaskDto taskDto = (TaskDto) o;
        return id == taskDto.id && todoId == taskDto.todoId && stateId ==
taskDto.stateId && Objects.equals(name, taskDto.name) && Objects.equals(priority,
taskDto.priority);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(id, name, priority, todoId, stateId);
    }

    @Override
    public String toString() {
        return "TaskDto { " +
                "id = " + id +
                ", name = '" + name + '\'' +
                ", priority = '" + priority + '\'' +
                ", todoId = " + todoId +
                ", stateId = " + stateId +
                " }";
    }
}
package com.softserve.itacademy.todolist.dto;

import lombok.Data;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Data
public class TaskRequestDto {
    @NotBlank(message = "The 'name' cannot be empty")
    private String name;

    @NotNull(message = "Task priority is required")
    private String priority;
```

```
    @NotNull(message = "ToDo ID is required")
    private Long todoId;

    @NotNull(message = "State ID is required")
    private Long stateId;
}
package com.softserve.itacademy.todolist.dto;

import lombok.Data;

@Data
public class TaskResponseDto {
    private Long id;
    private String name;
    private String priority;
    private Long todoId;
    private String todoTitle;
    private Long stateId;
    private String stateName;
}
package com.softserve.itacademy.todolist.dto;

import com.softserve.itacademy.todolist.model.Priority;
import com.softserve.itacademy.todolist.model.State;
import com.softserve.itacademy.todolist.model.Task;
import com.softserve.itacademy.todolist.model.ToDo;

public class TaskTransformer {
    public static TaskResponseDto toResponseDto(Task task) {
```

```
        TaskResponseDto responseDto = new TaskResponseDto();
        responseDto.setId(task.getId());
        responseDto.setName(task.getName());
        responseDto.setPriority(task.getPriority().toString());
        responseDto.setTodoId(task.getTodo().getId());
        responseDto.setTodoTitle(task.getTodo().getTitle());
        responseDto.setStateId(task.getState().getId());
        responseDto.setStateName(task.getState().getName());
        return responseDto;
    }

    public static Task toEntity(TaskRequestDto requestDto, ToDo todo, State
state) {
        Task task = new Task();
        task.setName(requestDto.getName());
        task.setPriority(Priority.valueOf(requestDto.getPriority()));
        task.setTodo(todo);
        task.setState(state);
        return task;
    }
}
package com.softserve.itacademy.todolist.dto;

import com.fasterxml.jackson.databind.PropertyNamingStrategies;
import com.fasterxml.jackson.databind.annotation.JsonNaming;
import lombok.Data;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
```

```
@Data
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
public class ToDoRequestDTO {
    @NotBlank(message = "The title cannot be empty")
    private String title;
    @NotNull(message = "OwnerId is required")
    private Long ownerId;
}
package com.softserve.itacademy.todolist.dto;

import lombok.Data;

import java.time.LocalDateTime;

@Data
public class ToDoResponseDTO {
    private Long id;
    private String title;
    private LocalDateTime createdAt;
    private UserResponse owner;

}
package com.softserve.itacademy.todolist.dto;
import com.softserve.itacademy.todolist.model.ToDo;
import lombok.Data;
import org.springframework.stereotype.Component;

import java.time.LocalDateTime;

@Component
```

```
public class ToDoTransformer {
    public ToDo convertToEntity(ToDoRequestDTO dto) {
        ToDo toDo = new ToDo();
        toDo.setTitle(dto.getTitle());
        toDo.setCreatedAt(LocalDateTime.now());
        return toDo;
    }
    public void updateEntity(ToDo toDo, ToDoRequestDTO dto) {
        toDo.setTitle(dto.getTitle());
    }

    public ToDoResponseDTO convertToDto(ToDo toDo) {
        ToDoResponseDTO dto = new ToDoResponseDTO();
        dto.setId(toDo.getId());
        dto.setTitle(toDo.getTitle());
        dto.setCreatedAt(toDo.getCreatedAt());
        dto.setOwner(toDo.getOwner() != null ? new
UserResponse(toDo.getOwner()) : null);
        return dto;
    }
}
package com.softserve.itacademy.todolist.dto;

import com.fasterxml.jackson.databind.PropertyNamingStrategies;
import com.fasterxml.jackson.databind.PropertyNamingStrategy;
import com.fasterxml.jackson.databind.annotation.JsonNaming;
import com.softserve.itacademy.todolist.model.User;
import lombok.Value;
import
org.springframework.data.mapping.model.PropertyNameFieldNamingStrategy;
```

```
@Value
@JsonNaming(PropertyNamingStrategies.SnakeCaseStrategy.class)
public class UserResponse {
    Long id;
    String firstName;
    String lastName;
    String email;
    String role;

    public UserResponse(User user) {
        id = user.getId();
        firstName = user.getFirstName();
        lastName = user.getLastName();
        email = user.getEmail();
        role = user.getRole().getName();
    }
}
package com.softserve.itacademy.todolist.exception;

import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.ModelAndView;
```

```
import
org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionH
andler;

import javax.persistence.EntityNotFoundException;
import javax.servlet.http.HttpServletRequest;
import java.util.stream.Collectors;

@Slf4j
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public                                          ResponseEntity<String>
handleMethodArgumentNotValid(MethodArgumentNotValidException ex) {
        log.error("Validation error: {}", ex.getMessage());
        String validationErrors = ex.getBindingResult().getFieldErrors().stream()
                .map(error -> error.getField() + ": " + error.getDefaultMessage())
                .collect(Collectors.joining(", "));
        return    ResponseEntity.badRequest().body("Validation    failed:    "    +
validationErrors);
    }

    @ExceptionHandler(EntityNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public                                          ResponseEntity<String>
handleEntityNotFoundException(EntityNotFoundException ex) {
        log.error("Entity not found: {}", ex.getMessage());
```

```
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Error: " +
ex.getMessage());
    }

    @ExceptionHandler(NullEntityReferenceException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ResponseEntity<String>
handleNullEntityReferenceException(NullEntityReferenceException ex) {
        log.error("Null reference: {}", ex.getMessage());
        return ResponseEntity.badRequest().body("Null entity reference: " +
ex.getMessage());
    }

    @ExceptionHandler(AccessDeniedException.class)
    @ResponseStatus(HttpStatus.FORBIDDEN)
    public ResponseEntity<String>
handleAccessDeniedException(AccessDeniedException ex) {
        log.error("Access denied: {}", ex.getMessage());
        return ResponseEntity.status(HttpStatus.FORBIDDEN).body("Access
denied: " + ex.getMessage());
    }

    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ResponseEntity<String> handleGenericException(Exception ex) {
        log.error("Unexpected error: {}", ex.getMessage());
        return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An
unexpected error occurred: " + ex.getMessage());
    }
```

```
}
package com.softserve.itacademy.todolist.exception;

public class NullEntityReferenceException extends RuntimeException {
   public NullEntityReferenceException() {    }

   public NullEntityReferenceException(String message) {
      super(message);
   }
}
package com.softserve.itacademy.todolist.model;

public enum Priority {
   LOW, MEDIUM, HIGH
}
package com.softserve.itacademy.todolist.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import lombok.ToString;
import org.hibernate.Hibernate;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import java.util.List;
import java.util.Objects;
```

```
@Getter @Setter @NoArgsConstructor @ToString
@Entity @Table(name = "roles")
public class Role implements GrantedAuthority {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "The 'name' cannot be empty")
    @Column(name = "name", nullable = false, unique = true)
    private String name;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return
false;
        Role role = (Role) o;
        return getId() != null && getId().equals(role.getId());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    @Override
    public String getAuthority() {
        return "ROLE_" + name;
    }
}
```

```
package com.softserve.itacademy.todolist.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.hibernate.Hibernate;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import java.util.List;
import java.util.Objects;

@Getter @Setter @NoArgsConstructor
@Entity @Table(name = "states")
public class State {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "The 'name' cannot be empty")
    @Column(name = "name", nullable = false, unique = true)
    private String name;

    @OneToMany(mappedBy = "state")
    private List<Task> tasks;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return
false;
        State state = (State) o;
        return getId() != null && getId().equals(state.getId());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    @Override
    public String toString() {
        return "State { " +
                "id = " + id +
                ", name = '" + name + '\'' +
                " }";
    }
}
package com.softserve.itacademy.todolist.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.hibernate.Hibernate;

import javax.persistence.*;
import java.util.Objects;

@Getter @Setter @NoArgsConstructor
```

```
@Entity
@Table(name = "tasks")
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "priority")
    @Enumerated(EnumType.STRING)
    private Priority priority;

    @ManyToOne
    @JoinColumn(name = "todo_id")
    private ToDo todo;

    @ManyToOne
    @JoinColumn(name = "state_id")
    private State state;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return
false;
        Task task = (Task) o;
        return getId() != null && getId().equals(task.getId());
    }
```

```
    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    @Override
    public String toString() {
        return "Task { " +
                "id = " + id +
                ", name = '" + name + '\'' +
                ", priority = " + priority +
                ", todo = " + todo +
                ", state = " + state +
                " }";
    }
}
package com.softserve.itacademy.todolist.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.hibernate.Hibernate;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Objects;
```

```
@Getter @Setter @NoArgsConstructor
@Entity @Table(name = "todos")
public class ToDo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "The 'title' cannot be empty")
    @Column(name = "title", nullable = false, unique = true)
    private String title;

    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private User owner;

    @OneToMany(mappedBy = "todo", cascade = CascadeType.REMOVE)
    private List<Task> tasks;

    @ManyToMany
    @JoinTable(name = "todo_collaborator",
            joinColumns = @JoinColumn(name = "todo_id"),
            inverseJoinColumns = @JoinColumn(name = "collaborator_id"))
    private List<User> collaborators;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return
false;
        ToDo toDo = (ToDo) o;
        return getId() != null && getId().equals(toDo.getId());
    }

    @Override
    public int hashCode() {
        return getClass().hashCode();
    }

    @Override
    public String toString() {
        return "ToDo { " +
                "id = " + id +
                ", title = '" + title + '\'' +
                ", createdAt = " + createdAt +
                " }";
    }
}
package com.softserve.itacademy.todolist.model;

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import org.hibernate.Hibernate;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import javax.persistence.*;
```

```
import javax.validation.constraints.Pattern;
import java.util.Collection;
import java.util.Collections;
import java.util.List;
import java.util.Objects;

@Getter @Setter @NoArgsConstructor
@Entity
@Table(name = "users")
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Pattern(regexp = "[A-Z][a-z]+",
            message = "Must start with a capital letter followed by one or more lowercase letters")
    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Pattern(regexp = "[A-Z][a-z]+",
            message = "Must start with a capital letter followed by one or more lowercase letters")
    @Column(name = "last_name", nullable = false)
    private String lastName;

    @Pattern(regexp = "[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}", message = "Must be a valid e-mail address")
    @Column(name = "email", nullable = false, unique = true)
    private String email;
```

```
        @Pattern(regexp = "[A-Za-z\\d]{6,}",
                message = "Must be minimum 6 symbols long, using digits and latin letters")
        @Pattern(regexp = ".*\\d.*",
                message = "Must contain at least one digit")
        @Pattern(regexp = ".*[A-Z].*",
                message = "Must contain at least one uppercase letter")
        @Pattern(regexp = ".*[a-z].*",
                message = "Must contain at least one lowercase letter")
        @Column(name = "password", nullable = false)
        private String password;

        @ManyToOne
        @JoinColumn(name = "role_id")
        private Role role;

        @OneToMany(mappedBy = "owner", cascade = CascadeType.REMOVE)
        private List<ToDo> myTodos;

        @ManyToMany
        @JoinTable(name = "todo_collaborator",
                joinColumns = @JoinColumn(name = "collaborator_id"),
                inverseJoinColumns = @JoinColumn(name = "todo_id"))
        private List<ToDo> otherTodos;


        @Override
        public Collection<? extends GrantedAuthority> getAuthorities() {
           return List.of(role);
```

```
}

@Override
public String getUsername() {
    return email;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
```

```
            if (o == null || Hibernate.getClass(this) != Hibernate.getClass(o)) return
false;
            User user = (User) o;
            return getId() != null && getId().equals(user.getId());
        }

        @Override
        public int hashCode() {
            return getClass().hashCode();
        }

        @Override
        public String toString() {
            return "User { " +
                    "id = " + id +
                    ", firstName = '" + firstName + '\'' +
                    ", lastName = '" + lastName + '\'' +
                    ", email = '" + email + '\'' +
                    ", password = '" + password + '\'' +
                    ", role = " + role +
                    " }";
        }
    }
    package com.softserve.itacademy.todolist.service.impl;

    import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
    import com.softserve.itacademy.todolist.model.Role;
    import com.softserve.itacademy.todolist.repository.RoleRepository;
    import com.softserve.itacademy.todolist.service.RoleService;
```

```
import org.springframework.stereotype.Service;

import javax.persistence.EntityNotFoundException;
import java.util.List;

@Service
public class RoleServiceImpl implements RoleService {

   private final RoleRepository roleRepository;

   public RoleServiceImpl(RoleRepository roleRepository) {
      this.roleRepository = roleRepository;
   }

   @Override
   public Role create(Role role) {
      if (role != null) {
         return roleRepository.save(role);
      }
      throw new NullEntityReferenceException("Role cannot be 'null'");
   }

   @Override
   public Role readById(long id) {
      return roleRepository.findById(id).orElseThrow(
            () -> new EntityNotFoundException("Role with id " + id + " not
found"));
   }

   @Override
```

```
    public Role update(Role role) {
        if (role != null) {
            readById(role.getId());
            return roleRepository.save(role);
        }
        throw new NullEntityReferenceException("Role cannot be 'null'");
    }

    @Override
    public void delete(long id) {
        Role role = readById(id);
        roleRepository.delete(role);
    }

    @Override
    public List<Role> getAll() {
        return roleRepository.findAll();
    }
}
```

```
package com.softserve.itacademy.todolist.service.impl;

import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
import com.softserve.itacademy.todolist.model.State;
import com.softserve.itacademy.todolist.repository.StateRepository;
import com.softserve.itacademy.todolist.service.StateService;
import org.springframework.stereotype.Service;

import javax.persistence.EntityNotFoundException;
import java.util.List;
```

```
import java.util.Optional;

@Service
public class StateServiceImpl implements StateService {

    private final StateRepository stateRepository;

    public StateServiceImpl(StateRepository stateRepository) {
        this.stateRepository = stateRepository;
    }

    @Override
    public State create(State state) {
        if (state != null) {
            return stateRepository.save(state);
        }
        throw new NullEntityReferenceException("State cannot be 'null'");
    }

    @Override
    public State readById(long id) {
        return stateRepository.findById(id).orElseThrow(
                () -> new EntityNotFoundException("State with id " + id + " not
found"));
    }

    @Override
    public State update(State state) {
        if (state != null) {
            readById(state.getId());
```

```
            return stateRepository.save(state);
        }
        throw new NullEntityReferenceException("State cannot be 'null'");
    }

    @Override
    public void delete(long id) {
        State state = readById(id);
        stateRepository.delete(state);
    }

    @Override
    public List<State> getAll() {
        return stateRepository.findByOrderByIdAsc();
    }

    @Override
    public State getByName(String name) {
        Optional<State> optional = Optional.ofNullable(stateRepository.findByName(name));
        if (optional.isPresent()) {
            return optional.get();
        }
        throw new EntityNotFoundException("State with name '" + name + "' not found");
    }
}
package com.softserve.itacademy.todolist.service.impl;
```

```
import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
import com.softserve.itacademy.todolist.model.Task;
import com.softserve.itacademy.todolist.repository.TaskRepository;
import com.softserve.itacademy.todolist.service.TaskService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;

import javax.persistence.EntityNotFoundException;
import java.util.List;

@Service
public class TaskServiceImpl implements TaskService {

   private static final Logger logger =
LoggerFactory.getLogger(TaskServiceImpl.class);


   private final TaskRepository taskRepository;

   public TaskServiceImpl(TaskRepository taskRepository) {
      this.taskRepository = taskRepository;
   }

   @Override
   public Task create(Task task) {
      if (task != null) {
         return taskRepository.save(task);
      }
```

```
            throw new NullEntityReferenceException("Task cannot be 'null'");
        }

        @Override
        public Task readById(long id) {

            EntityNotFoundException exception = new
EntityNotFoundException("Task with id " + id + " not found");
            logger.error(exception.getMessage(), exception);

            return taskRepository.findById(id).orElseThrow(
                    () -> exception);
        }

        @Override
        public Task update(Task task) {
            if (task != null) {
                readById(task.getId());
                return taskRepository.save(task);
            }
            throw new NullEntityReferenceException("Task cannot be 'null'");
        }

        @Override
        public void delete(long id) {
            Task task = readById(id);
            taskRepository.delete(task);
        }

        @Override
```

```
    public List<Task> getAll() {
        return taskRepository.findAll();
    }

    @Override
    public List<Task> getByTodoId(long todoId) {
        return taskRepository.getByTodoId(todoId);
    }
}
package com.softserve.itacademy.todolist.service.impl;

import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
import com.softserve.itacademy.todolist.model.ToDo;
import com.softserve.itacademy.todolist.model.User;
import com.softserve.itacademy.todolist.repository.ToDoRepository;
import com.softserve.itacademy.todolist.repository.UserRepository;
import com.softserve.itacademy.todolist.service.ToDoService;
import com.softserve.itacademy.todolist.service.UserService;
import org.springframework.stereotype.Service;

import javax.persistence.EntityNotFoundException;
import java.util.List;

@Service
public class ToDoServiceImpl implements ToDoService {

    private final ToDoRepository todoRepository;
    private final UserRepository userRepository;
```

```
    public ToDoServiceImpl(ToDoRepository todoRepository,UserRepository
userRepository) {
        this.todoRepository = todoRepository;
        this.userRepository= userRepository;
    }

    @Override
    public ToDo create(ToDo todo) {
        if (todo != null) {
            return todoRepository.save(todo);
        }
        throw new NullEntityReferenceException("ToDo cannot be 'null'");
    }

    @Override
    public ToDo readById(long id) {
        return todoRepository.findById(id).orElseThrow(
                () -> new EntityNotFoundException("ToDo with id " + id + " not
found"));
    }

    @Override
    public ToDo update(ToDo toDo, Long ownerId) {
        if (ownerId == null) {
            throw new IllegalArgumentException("OwnerId must not be null");
        }

        ToDo existingToDo = todoRepository.findById(toDo.getId())
                .orElseThrow(() -> new EntityNotFoundException("ToDo not found
with id " + toDo.getId()));
```

```
            existingToDo.setTitle(toDo.getTitle());

            User owner = userRepository.findById(ownerId)
                    .orElseThrow(() -> new EntityNotFoundException("User not found with id " + ownerId));
            existingToDo.setOwner(owner);

            return todoRepository.save(existingToDo);
        }


        @Override
        public void delete(long id) {
            ToDo todo = readById(id);
            todoRepository.delete(todo);
        }

        @Override
        public List<ToDo> getAll() {
            return todoRepository.findAll();
        }

        @Override
        public List<ToDo> getByUserId(long userId) {
            return todoRepository.getByUserId(userId);
        }
    }
    package com.softserve.itacademy.todolist.service.impl;
```

```
import
com.softserve.itacademy.todolist.exception.NullEntityReferenceException;
import com.softserve.itacademy.todolist.model.User;
import com.softserve.itacademy.todolist.repository.UserRepository;
import com.softserve.itacademy.todolist.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import javax.persistence.EntityNotFoundException;
import java.util.List;

@Service
public class UserServiceImpl implements UserService {
   @Autowired
   private UserRepository userRepository;

   @Override
   public User create(User role) {
      if (role != null) {
         return userRepository.save(role);
      }
      throw new NullEntityReferenceException("User cannot be 'null'");
   }

   @Override
   public User readById(long id) {
      return userRepository.findById(id).orElseThrow(
```

```
                () -> new EntityNotFoundException("User with id " + id + " not
found"));
        }

        @Override
        public User readByEmail(String email) {
            User user = userRepository.findByEmail(email);
            if (user == null) {
                throw new EntityNotFoundException("User with email " + email + " not
found");
            }
            return user;
        }

        @Override
        public User update(User role) {
            if (role != null) {
                readById(role.getId());
                return userRepository.save(role);
            }
            throw new NullEntityReferenceException("User cannot be 'null'");
        }

        @Override
        public void delete(long id) {
            User user = readById(id);
            userRepository.delete(user);
        }

        @Override
```

```
    public List<User> getAll() {
        return userRepository.findAll();
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByEmail(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not Found!");
        }
        return user;
    }
}
package com.softserve.itacademy.todolist;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ToDoListApplication {

    public static void main(String[] args) {
        SpringApplication.run(ToDoListApplication.class, args);
    }

}
```

Додаток Г (обов'язковий)

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: «Розробка API для системи управління завданнями із використанням Spring Framework»

Тип роботи: магістерська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра АІІТ
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 0.91 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

◉ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

• У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

• У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

Бісікало О.В., зав. каф. АІІТ (підпис)
(прізвище, ініціали, посада)

Овчинников К.В., доц. каф. АІІТ (підпис)
(прізвище, ініціали, посада)

Особа, відповідальна за перевірку (підпис) Маслій Р.В.
(прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник (підпис) Кабачій В. В. доц. кафедри АІІТ
(підпис) (прізвище, ініціали, посада)

Студент (підпис) Галіброда А. С.
(підпис) (прізвище, ініціали)