ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-застосунку для менеджменту командних студентських проєктів»

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

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

_____ Єгор ПОПІНАКО
(підпис)

Виконав: здобувач вищої освіти групи ПД-41
Єгор ПОПІНАКО

Керівник: Ірина ЗАМРІЙ
д.т.н., професор
Рецензент:

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Попінако Єгору Олексійовичу

1. Тема кваліфікаційної роботи: «Розробка web-застосунку для менеджменту командних студентських проєктів»
керівник кваліфікаційної роботи д.т.н., професор Ірина ЗАМРІЙ,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки web-застосунків, опис роботи web-застосунків, зокрема клієнт-серверної архітектури, документація фреймворку Spring Boot.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
    1. Огляд застосунку для організації командної роботи, визначення вимог до програмного забезпечення.
    2. Огляд та аналіз засобів реалізації застосунку для менеджменту командних студентських проєктів.
    3. Проєктування архітектури застосунку для менеджменту командних студентських проєктів.

4. Програмна реалізація застосунку для менеджменту командних студентських проєктів.
5. Тестування застосунку для менеджменту командних студентських проєктів.

5. Перелік графічного матеріалу: *презентація*
    1. Аналіз аналогів.
    2. Вимоги до програмного забезпечення.
    3. Програмні засоби реалізації.
    4. Діаграма варіантів використання
    5. Діаграма послідовності
    6. Діаграма класів
    7. Екранні форми.
    8. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вивчення теми, постановка задачі, формулювання цілей	25.02.-27.02.2025	
2	Аналіз та дослідження існуючих аналогів	28.02-02.03.2025	
3	Огляд застосунку для організації командної роботи, визначення вимог до програмного забезпечення.	03.03-06.03.2025	
4	Огляд та аналіз засобів реалізації застосунку для менеджменту командних студентських проєктів.	07.03-09.03.2025	
5	Проєктування архітектури застосунку для менеджменту командних студентських проєктів.	10.03-11.03.2025	
6	Програмна реалізація застосунку для менеджменту командних студентських проєктів.	12.03-27.04.2025	
7	Тестування застосунку для менеджменту командних студентських проєктів.	28.04-30.04.2025	
8	Розробка демонстраційних матеріалів	01.05-12.05.2024	
9	Попередній захист роботи	13.05-31.05.2024	

Здобувач вищої освіти _____ Єгор ПОПІНАКО
(підпис)

Керівник
кваліфікаційної роботи _____ Ірина ЗАМРІЙ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 64 стор., 1 табл., 29 рис., 20 джерел.

Мета роботи – покращення процесів організації, планування та контролю командної роботи студентів шляхом створення функціонального web-застосунку.

Об'єкт дослідження – процес організації та менеджменту командних проєктів.

Предмет дослідження – web-застосунок для управління командною роботою студентів у рамках навчальних проєктів.

Короткий зміст роботи: У кваліфікаційній роботі проаналізовано предметну область управління студентськими проєктами та методи реалізації сучасних web-застосунків. Проведено огляд існуючих систем для координації командної роботи студентів, зокрема Trello, Asana, Google Classroom. На основі виявлених вимог розроблено web-застосунок для керування студентськими проєктами з підтримкою таких функціональних можливостей: реєстрація та авторизація користувачів за допомогою Google-акаунтів, створення проєктів і завдань, призначення учасників, відстеження дедлайнів, інтеграція з Google Classroom API, надсилання сповіщень У роботі реалізовано бекенд на основі фреймворку Spring Boot з використанням Java, JPA (Hibernate), MySQL, Redis та JWT для захисту API, а також фронтенд з використанням HTML/CSS, JavaScript і Thymeleaf.

Результати роботи можуть бути використані для організації навчальної діяльності студентських команд, викладачами, кураторами груп або як основа для розширених систем управління освітніми процесами.

КЛЮЧОВІ СЛОВА: BACK-END, FRONT-END, SPRING BOOT, JAVA, GOOGLE CLASSROOM API, JWT, WEB-ЗАСТОСУНОК, УПРАВЛІННЯ ПРОЄКТАМИ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

API – Application Programming Interface

JWT – JSON Web Token

OAuth2 – Open Authorization 2.0

DTO – Data Transfer Object

JPA – Java Persistence API

ORM – Object-Relational Mapping

SQL – Structured Query Language

IDE – Integrated Development Environment

DI – Dependency Injection

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

JS – JavaScript

Redis – Remote Dictionary Server (сервер ключ-значення для кешування)

BLOB – Binary Large Object (двійкові дані)

ВСТУП

Актуальність: У сучасних умовах цифровізації освіти та широкого впровадження дистанційного та змішаного навчання особливої важливості набуває ефективна організація командної роботи студентів. В умовах, коли більшість проєктної діяльності переміщується в онлайн-простір, постає потреба в інструментах, які дозволяють не лише розподіляти завдання, а й забезпечити контроль їх виконання, комунікацію між учасниками та інтеграцію з іншими цифровими сервісами.

Розробка спеціалізованого web-застосунку для менеджменту студентських командних проєктів дозволяє автоматизувати процеси постановки та розподілу завдань, відстеження прогресу, взаємодії між членами команди, що в результаті сприяє підвищенню ефективності освітнього процесу. Такий застосунок допомагає не лише студентам краще координувати свою роботу, а й викладачам – ефективніше здійснювати моніторинг командної динаміки та рівня залученості кожного учасника.

Крім того, популяризація дистанційного та змішаного навчання створює додатковий попит на подібні цифрові рішення, що робить тему розробки web-застосунку для підтримки командної роботи студентів актуальною як у контексті освітніх потреб, так і в ширшому контексті цифрової трансформації освіти.

Об'єктом дослідження є процес організації та менеджменту командних проєктів.

Предметом дослідження є web-застосунок для управління командною роботою студентів у рамках навчальних проєктів.

Метою роботи є покращення процесів організації, планування та контролю командної роботи студентів шляхом створення функціонального web-застосунку.

Методи дослідження: Першим етапом дослідження стало вивчення аналогічних рішень у сфері керування проєктами, зокрема таких інструментів як

Trello, Asana, Microsoft Teams та Jira. На основі їх аналізу було визначено перелік функціональних і нефункціональних вимог до власного застосунку.

На наступному етапі розробки було сформовано технологічний стек, який забезпечив реалізацію всіх вимог до функціональності, масштабованості та інтеграційної здатності системи. Для серверної частини було обрано мову програмування Java у поєднанні з фреймворком Spring Boot, який забезпечує модульну побудову, чітке розділення відповідальностей між компонентами та вбудовану підтримку безпеки, авторизації та інтеграції з базами даних. Для зберігання структурованої інформації використовувалась реляційна база даних MySQL, а для збереження сесійних токенів і швидкого доступу до тимчасових даних — Redis, розгорнутий у Docker-контейнері. Авторизацію користувачів реалізовано через поєднання OAuth2-протоколу з використанням облікових записів Google та JWT-токенів, що забезпечують гнучке й безпечне управління доступом на рівні ресурсів.

Наукова новизна виконаної роботи полягає у створенні інтегрованого інструменту управління студентськими проєктами, який поєднує в собі можливості традиційної системи керування завданнями та освітньої платформи. Ключовими елементами новизни є автоматизована синхронізація з Google Classroom а також реалізація модуля персоналізованих сповіщень про дедлайни.

Практична значущість результатів дипломної роботи полягає у можливості використання системи як готового рішення для підтримки дисциплін проєктного типу у вищих навчальних закладах. Запропонований підхід сприяє підвищенню організованості командної роботи, відповідальності учасників, прозорості виконання завдань та покращенню взаємодії між студентами і викладачами. Крім того, застосунок може бути використаний як база для подальшого вдосконалення та масштабування — наприклад, шляхом розширення API, додавання аналітичних модулів або реалізації мобільної версії.

1 АНАЛІЗ WEB-ЗАСТОСУНКУ ДЛЯ МЕНЕДЖМЕНТУ СТУДЕНТСЬКИХ КОМАНДНИХ ПРОЄКТІВ

1.1 Web-застосунок для управління студентськими проєктами

Web-застосунок для управління студентськими командними проєктами — це спеціалізована інформаційна система, створена з метою організації, планування та моніторингу навчальних командних активностей. Такий застосунок надає студентам, викладачам і координаторам зручний цифровий простір для взаємодії, де кожен учасник може бачити завдання, терміни, виконання, коментарі та результати командної роботи. Основна мета web-застосунку — підвищити ефективність спільної роботи над проєктами, сприяти дотриманню дедлайнів, оптимізувати розподіл обов’язків і інтегрувати навчальний процес з платформою Google Classroom.

Основні функціональні компоненти web-застосунку включають:

- авторизація через Google-акаунт – інтеграція з Google OAuth2 дозволяє користувачам швидко входити до системи, використовуючи наявні облікові записи, що значно спрощує процес реєстрації та підвищує безпеку;
- управління проєктами – користувачі можуть створювати нові проєкти, редагувати їх характеристики, призначати учасників, видаляти завершені або неактуальні проєкти;
- менеджмент завдань – застосунок підтримує створення, оновлення, призначення, відмітку виконання, зміну пріоритетів і видалення завдань. Завдання можуть мати статуси (наприклад: "To Do", "In Progress", "Done"), що дозволяє зручно відслідковувати стан роботи;
- канбан-дошка – візуальне представлення завдань за статусами, що дозволяє командам працювати в стилі Agile та зручно планувати свої дії;

- інтеграція з Google Classroom – застосунок автоматично імпортує курси й завдання користувача, а також дозволяє переглядати прикріплені матеріали та відкривати їх безпосередньо в інтерфейсі застосунку;

- сторінка користувача (профіль) – кожен користувач має доступ до персонального профілю, де може редагувати свої дані;

- система сповіщень про дедлайни – користувачі отримують повідомлення про наближення термінів виконання завдань, що підвищує відповідальність і продуктивність командної роботи;

- ролі та права доступу – реалізована гнучка модель авторизації, яка обмежує доступ до функціоналу залежно від ролі користувача (наприклад: адміністратор, користувач). Це забезпечує безпеку системи та розмежування відповідальності.

Основні цілі web-застосунку для управління студентськими командними проєктами:

- забезпечення ефективної командної взаємодії студентів у межах навчального процесу;

- інтеграція з освітніми платформами для зручної роботи з навчальними матеріалами;

- покращення контролю за виконанням завдань та дотриманням термінів;

- покращення самостійної організації та відповідальності студентів.

Переваги:

- централізованість управління – уся інформація щодо проєкту, учасників,

- завдань і дедлайнів зібрана в одному місці; зручність використання – інтуїтивно зрозумілий інтерфейс з використанням шаблонізатора Thymeleaf;

- реальне відображення прогресу – завдяки канбан-дошці та статусам завдань користувачі бачать повну картину виконання;

- інтеграція з Google Classroom – підвищує ефективність навчального процесу через доступ до курсів, завдань і матеріалів;

- безпека – аутентифікація через Google та розмежування прав доступу гарантують безпечну роботу з персональними даними.

Недоліки:

- відсутність REST API – ускладнює інтеграцію з мобільними застосунками або іншими зовнішніми системами;
- залежність від Google-сервісів – для роботи деяких функцій (аутентифікація, курси) необхідна стабільна робота Google API;
- обмежена масштабованість – архітектура з орієнтацією на MVC і JSP/Thymeleaf має меншу гнучкість порівняно з фронтенд-фреймворками (React, Angular).

1.2 Сфера застосування web-застосунку

Розроблений web-застосунок призначений для використання у вищих навчальних закладах, коледжах та інших освітніх установах, де практикується командна форма роботи над навчальними або науковими проєктами. Також застосунок може бути адаптований для шкільних проєктів або внутрішньогрупової організації на курсах і хакатонах.

1.3 Цільова аудиторія

Цільовою аудиторією web-застосунку є:

- студенти технічних та гуманітарних спеціальностей, які працюють над командними проєктами в межах навчальних курсів;
- викладачі та керівники проєктів, які координують групову діяльність студентів, контролюють виконання завдань і оцінюють результати;
- освітні IT-адміністратори, які відповідають за інтеграцію та технічну підтримку цифрових платформ у навчальному закладі.

Застосунок може використовуватись як у межах одного факультету, так і на рівні всього навчального закладу. Його гнучка структура дозволяє адаптувати під різні освітні сценарії та масштаби.

1.4 Дослідження існуючих аналогів

На сьогоднішній день одними з найпопулярніших вебсайтів для управління проєктами є Trello, Asana, Jira, Microsoft Planner та ClickUp. Розглянемо їх детальніше.

1.4.1. Trello

Trello[1] — це інтуїтивно зрозумілий інструмент візуального управління проєктами, який використовує канбан-підхід. Його інтерфейс базується на дошках, що поділяються на колонки (наприклад, «To Do», «In Progress», «Done»), а задачі представлені у вигляді карток, які можна перетягувати між етапами виконання. Trello дозволяє користувачам створювати персоналізовані списки завдань, додавати теги, дедлайни, вкладення, коментарі та чек-листи. За допомогою функції Butler користувачі можуть автоматизувати частину робочих процесів без написання коду. Серед корисних можливостей також слід відзначити інтеграції з Google Drive, Dropbox, Slack, GitHub та іншими популярними сервісами. Trello найкраще підходить для невеликих команд або особистого планування, оскільки у безкоштовній версії його функціональність обмежена, особливо у сфері аналітики та ресурсного планування. На відміну від більш складних систем, таких як Jira чи ClickUp, Trello не підтримує створення діаграм Ганта або глибоке управління ролями в команді, однак його простота і візуальна подача роблять його зручним для швидкого старту.

Основні можливості:

- створення дошок для кожного проєкту;
- розбиття завдань на картки з описами, чек-листами та дедлайнами;
- можливість прикріплювати файли та додавати коментарі;
- відстеження статусу завдання за допомогою перетягування між колонками;
- інтеграція з Google Drive, Slack, GitHub та іншими сервісами;

Переваги:

- простий у використанні інтерфейс;
- гнучке управління завданнями;
- безкоштовний базовий план з достатнім функціоналом.

Недоліки:

- обмежені можливості для великих проєктів;
- відсутність розширених функцій управління ресурсами.

Екранні форми застосунку Trello наведено на рисунку 1.1

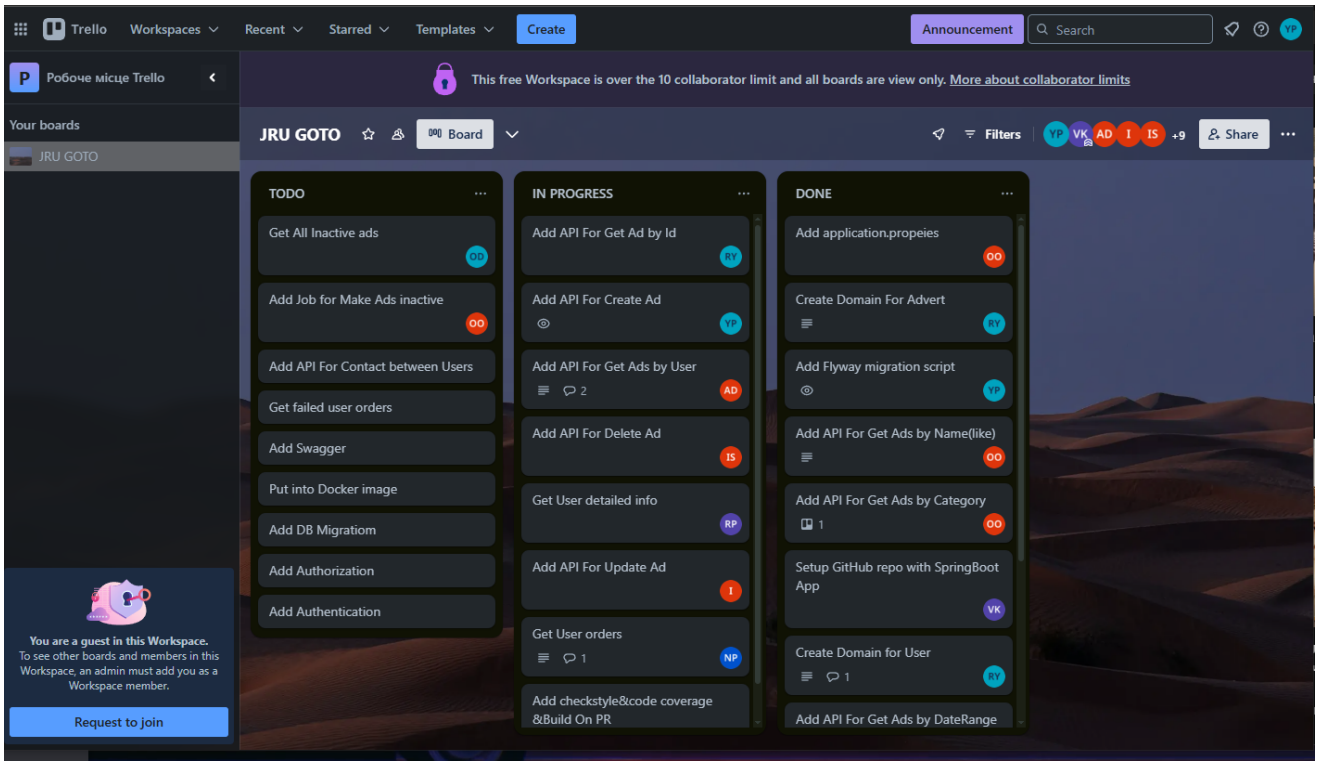


Рис. 1.1 Екранні форми застосунку Trello

1.4.2. Asana

Asana[2] — це гнучкий інструмент для команд, який дозволяє планувати, організовувати та відстежувати роботу в одному місці. Завдяки багатофункціональності, Asana є популярним серед стартапів та середніх компаній. Вона дозволяє створювати завдання та підзадачі, призначати їх на відповідальних осіб, встановлювати терміни, пріоритети, додавати описи, файли, теги та коментарі. Завдяки різним режимам перегляду (список, дошка, календар, таймлайн) кожен

учасник команди може працювати у зручному для себе форматі. Функція Workload дозволяє менеджерам бачити, як завантажені учасники проєкту, а Timeline — візуалізувати план виконання завдань у вигляді діаграми Ганта. Asana підтримує інтеграції з великою кількістю сервісів, таких як Slack, Google Calendar, Zoom, Zapier, Dropbox та ін. У порівнянні з Trello, Asana пропонує більше можливостей для аналізу продуктивності, проте її інтерфейс дещо складніший для новачків. Водночас, вона значно поступається Jira у сфері підтримки Agile-методологій та технічної документації.

Основні можливості:

- планування завдань та підзадач;
- встановлення пріоритетів та дедлайнів;
- відстеження прогресу виконання;
- можливість додавання коментарів та файлів до завдань;
- інтеграція з Google Drive, Slack, Zapier тощо.

Переваги:

- зручне управління проєктами та завданнями.
- можливість встановлення дедлайнів та відстеження прогресу.
- підтримка інтеграції з іншими сервісами.

Недоліки:

- відсутність вбудованого інструменту для фінансового моніторингу.
- складніший інтерфейс для нових користувачів.

Екранні форми застосунку Asana наведено на рисунку 1.2

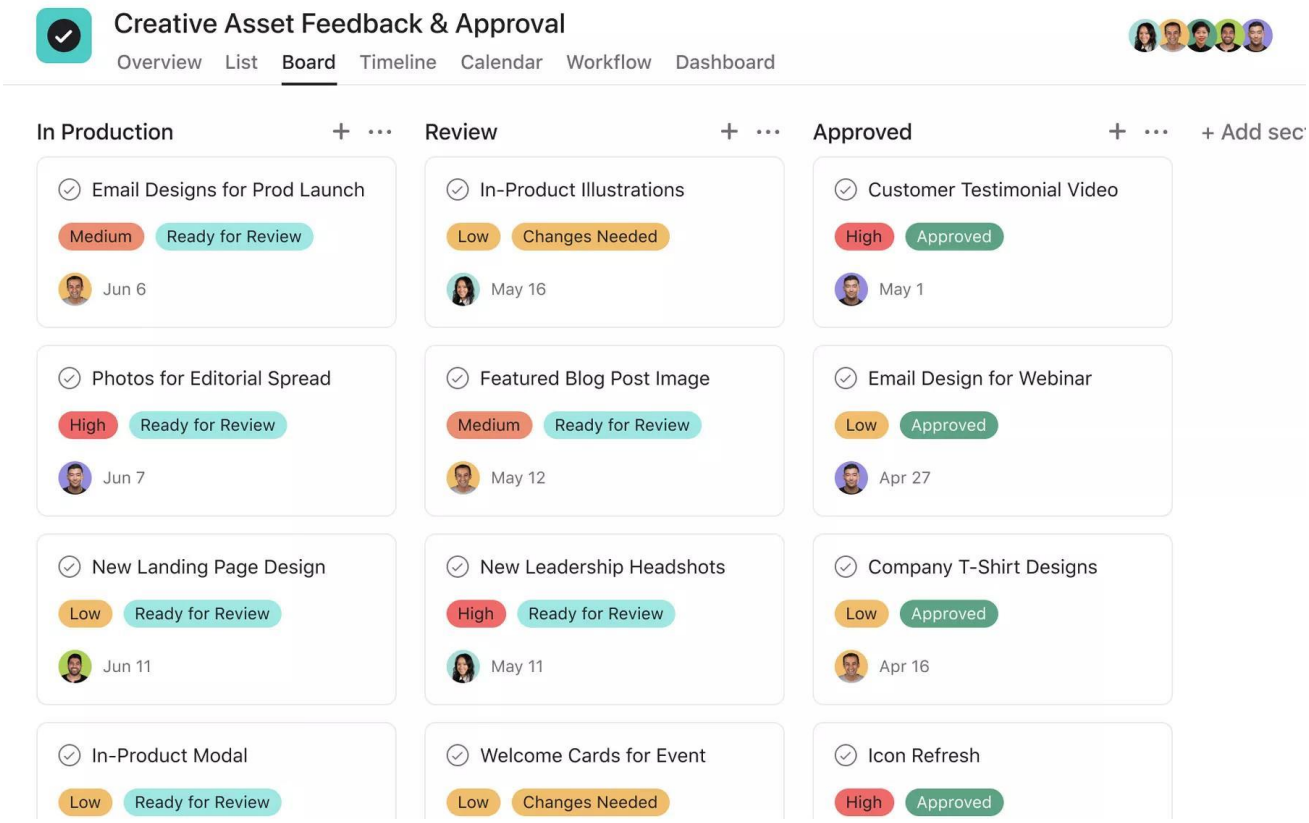


Рис. 1.2 Екранні форми застосунку Asana

1.4.3. Jira

Jira[3] — це потужний інструмент управління проєктами, який широко використовується в IT-сфері, особливо у командах розробників програмного забезпечення. Розроблений компанією Atlassian, Jira надає розвинений функціонал для ведення задач, спринтів, бектлогів, а також для баг-трекінгу. Він підтримує методології Agile, зокрема Scrum і Kanban, дозволяє гнучко налаштовувати робочі процеси та використовувати workflow-схеми, які відображають послідовність дій при виконанні задачі. Крім базових функцій, Jira дозволяє створювати кастомні поля, вести історію змін, формувати докладні звіти про продуктивність (burn down chart, velocity chart, control chart) та інтегруватися з багатьма DevOps-інструментами: Bitbucket, Confluence, GitHub, Jenkins тощо. Завдяки гнучкій системі дозволів, великі команди можуть ефективно управляти доступом до проєктів та інформації. Водночас, у порівнянні з Trello або Planner, Jira має складніший інтерфейс і потребує більше часу для початкового налаштування. Але

саме це дозволяє масштабувати її під будь-які проєктні потреби — від стартапів до великих корпорацій.

Основні можливості:

- управління завданнями та баг-трекінг;
- планування спринтів та відстеження прогресу;
- підтримка Scrum та Kanban дошок;
- можливість генерувати звіти щодо продуктивності;
- інтеграція з Bitbucket, Confluence та іншими DevOps-інструментами.

Переваги:

- потужний інструмент для управління великими проєктами;
- можливість відстеження помилок та тестування.

Недоліки:

- складний у налаштуванні для новачків;
- високий рівень навантаження на ресурси при масштабних проєктах.

Екранні форми застосунку Jira наведено на рисунку 1.3

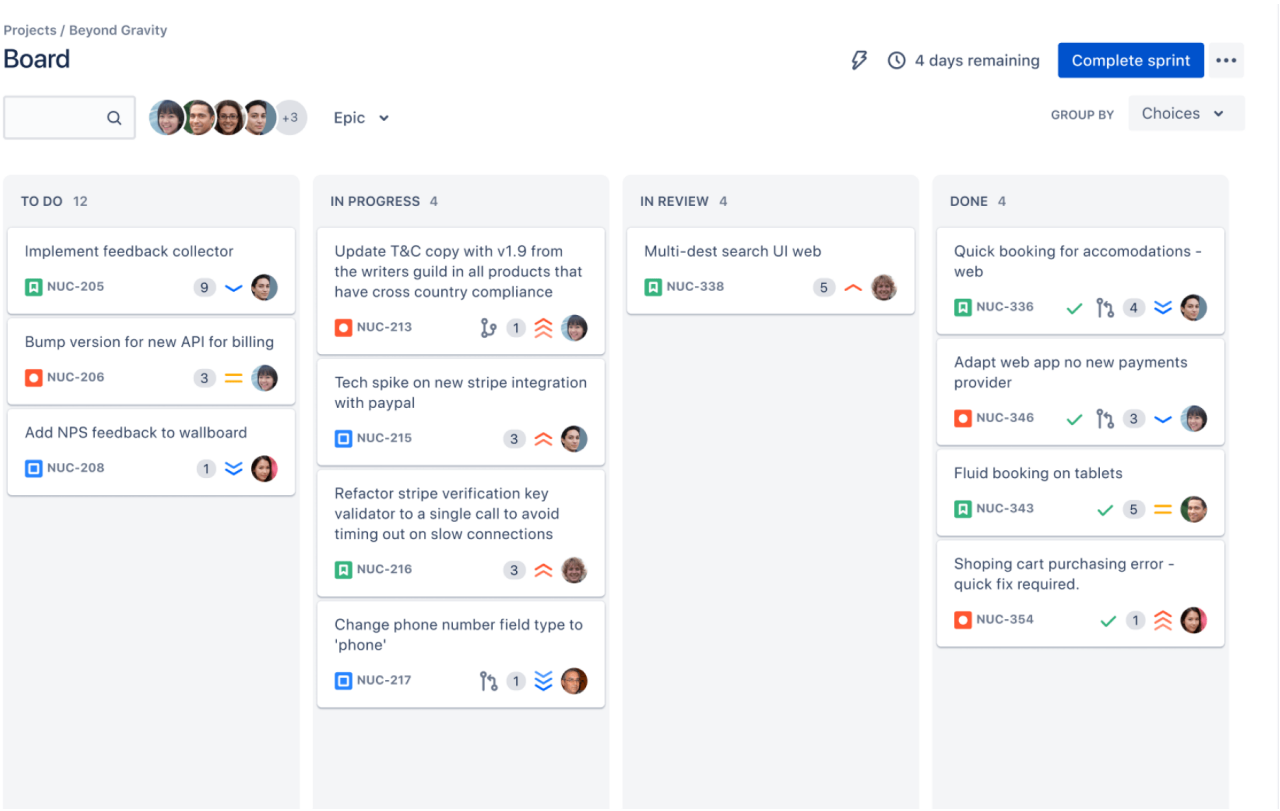


Рис. 1.3 Екранні форми застосунку Jira

1.4.4. Microsoft Planner

Microsoft Planner[4] — це простий та зручний інструмент управління завданнями, інтегрований в екосистему Microsoft 365. Його основна перевага — тісна інтеграція з Microsoft Teams, Outlook, SharePoint та іншими сервісами Microsoft, що робить його зручним вибором для організацій, які вже використовують ці продукти. Інтерфейс Planner базується на концепції дошок із завданнями у вигляді карток, які можна розподіляти по категоріях (buckets), призначати виконавцям, встановлювати дедлайни та додавати файли. Кожне завдання можна позначити мітками, додати нотатки та перевірочні списки. Хоча Planner не має глибоких функцій для моніторингу продуктивності чи побудови діаграм Ганта, його простота є водночас і перевагою — вхідний поріг дуже низький. На відміну від Jira або ClickUp, Planner не підтримує складних сценаріїв автоматизації, кастомних звітів або детального ресурсного планування, однак чудово підходить для невеликих команд або внутрішніх ініціатив у межах компаній, де головне — швидко створити план і організувати виконання.

Основні можливості:

- створення планів і завдань у вигляді карток;
- призначення відповідальних осіб;
- відстеження виконання завдань;
- інтеграція з Microsoft Teams, Outlook та SharePoint.

Переваги:

- глибока інтеграція з іншими продуктами Microsoft;
- проста організація завдань у форматі дошок.

Недоліки:

- недостатня кількість функцій для складних проєктів;
- відсутність можливостей для детального моніторингу ресурсів.

Екранні форми застосунку Microsoft Planner наведено на рисунку 1.4

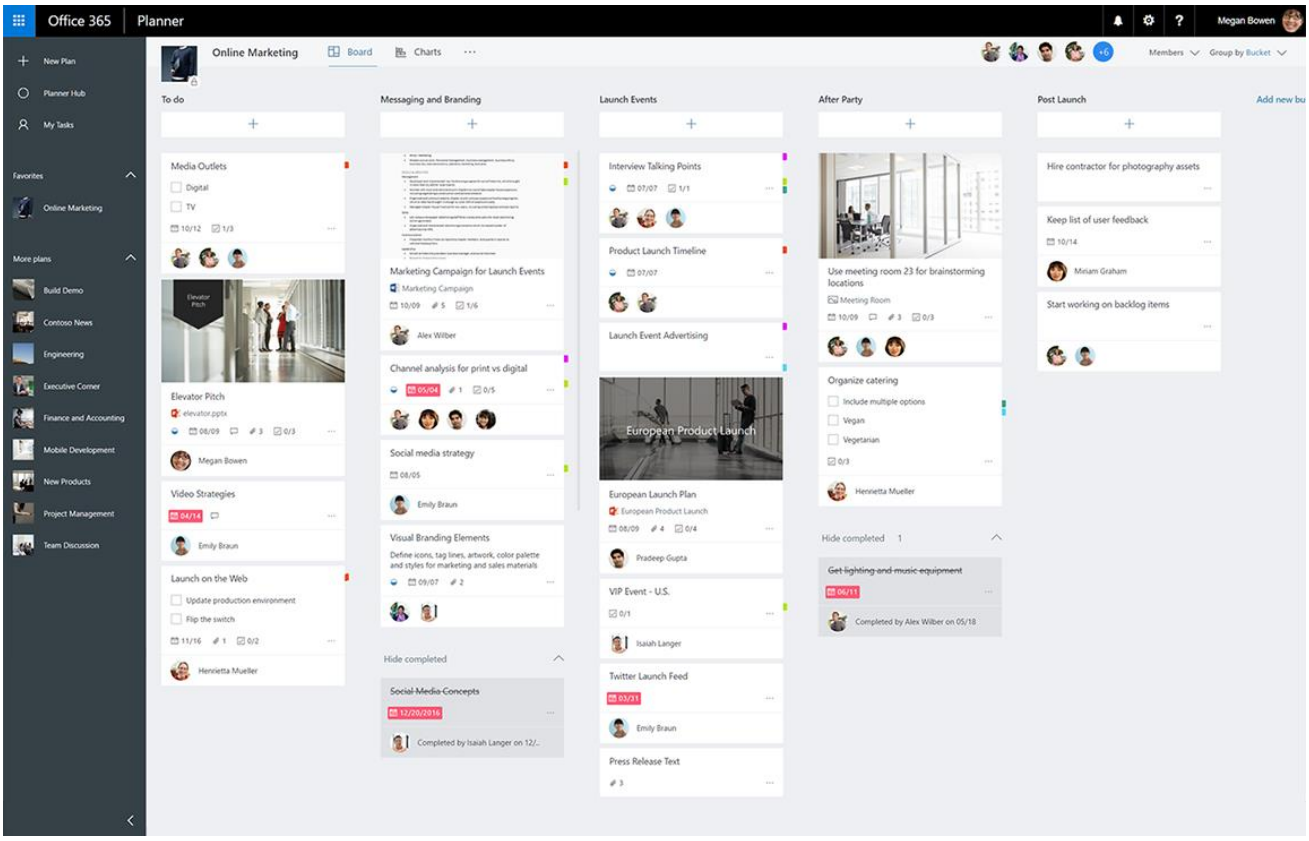


Рис. 1.4 Екранні форми застосунку Microsoft Planner

1.4.5. ClickUp

ClickUp[5] — це універсальний інструмент управління проєктами, який поєднує в собі можливості кількох платформ. Його головна особливість — максимальна гнучкість і налаштовуваність, що дозволяє адаптувати систему під практично будь-яку команду або стиль роботи. ClickUp підтримує різні типи відображення: список, дошку, календар, таймлайн, діаграму Ганта, мапу думок (mind map), workload, що дозволяє користувачам вибирати найзручніший спосіб роботи з інформацією. Завдання можуть мати підзадачі, залежності, шаблони, повторюваність, часові ліміти тощо. Платформа також має вбудований чат, документи, цілі (goals), форми та навіть власну автоматизацію, яка дозволяє запускати дії за заданими умовами. ClickUp добре інтегрується з Google Drive, Slack, Zoom, GitHub, Dropbox, Outlook та іншими сервісами. На відміну від Asana чи Trello, цей інструмент має значно більший функціонал "з коробки", однак для нових користувачів він може здатися перевантаженим. Через велику кількість можливостей стартова конфігурація потребує часу. Проте в перспективі ClickUp

дозволяє повністю централізувати управління проєктами, не користуючись сторонніми рішеннями.

Основні можливості:

- створення завдань, підзадач та контроль прогресу;
- підтримка різних видів відображення (дошка, список, календар);
- інтеграція з Google Drive, Slack, GitHub та іншими сервісами;
- можливість створення тайм-лінів та діаграм Ганта.

Переваги:

- гнучке налаштування інтерфейсу;
- багатофункціональність для управління складними проєктами.

Недоліки:

- може бути складним для нових користувачів.

Екранні форми застосунку ClickUp наведено на рисунку 3.5

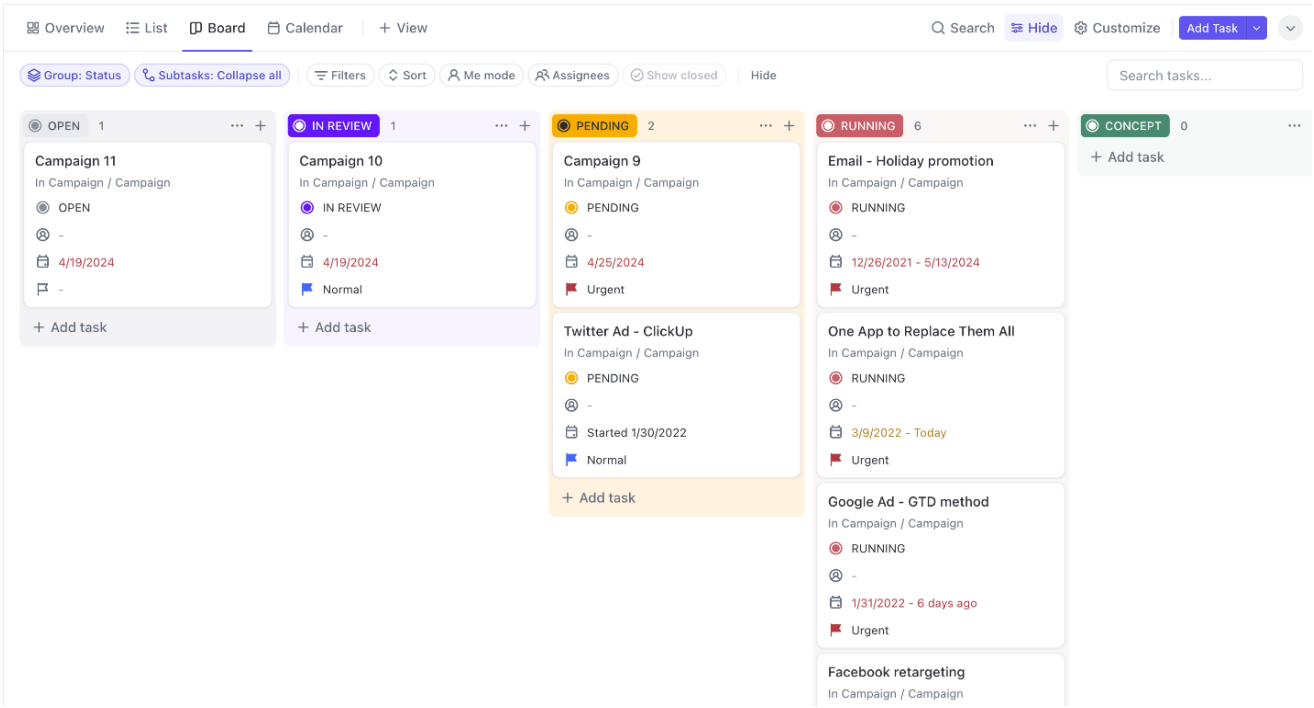


Рис. 1.5 Екранні форми застосунку ClickUp

Розглянуті програмні засоби мають свої переваги та недоліки, однак кожен з них підходить для різних категорій користувачів та типів проєктів. Вибір

інструменту залежить від складності задач, необхідних інтеграцій та специфіки роботи команди.

1.5 Порівняльний аналіз існуючих застосунків управління командними проєктами

В таблиці 1.1 зведено результати аналізу існуючих застосунків для вивчення японської мови та їх порівняння.

Таблиця 1.1

Зведена таблиця аналізу існуючих застосунків управління командними проєктами

Показник	Trello	Asana	Jira	Microsoft Planner	ClickUp
Інтеграція з Google Classroom	-	-	-	-	-
Ролі та управління доступом	+	+	+	+	+
Запрошення користувачів через email	+	+	+	+	+
Внутрішні повідомлення про кінцевий термін завдань	-	+	-	-	+
Профіль користувача	+	+	+	+	+
Управління завданнями	+	+	+	+	+

1.6 Визначення вимог до застосунку

Проаналізовані програмні продукти мають спільні недоліки, які обмежують їх ефективне використання в освітньому середовищі. Зокрема, жоден з них не підтримує інтеграцію з Google Classroom, що є ключовою вимогою для організації командної роботи студентів. Повідомлення про наближення дедлайнів завдань відсутні всередині багатьох систем або реалізовані лише у вигляді email-сповіщень, що знижує зручність моніторингу кінцевих термінів. Окремі рішення, як-от Jira, хоча й надають розширені можливості автоматизації, вимагають використання спеціалізованої мови запитів JQL для налаштування сповіщень. Більшість систем орієнтовані на корпоративне або загальне використання, тому не враховують специфіку навчального процесу та потреб освітніх команд.

Враховуючи мету дослідження та аналіз аналогів, були сформовані вимоги до застосунку.

Функціональні вимоги:

- реєстрація та авторизація користувачів;
- можливість створення, редагування та видалення проєктів;
- можливість управління завданнями в проєкті;
- можливість відстеження статусу завдань "не розпочато", "у процесі", "виконано";
- розмежування прав доступу(користувач, адміністратор);
- реалізація внутрішніх повідомлень про кінцевий термін завдань;
- можливість користувачам бачити свої курси, а також завдання та навчальні матеріали всередині системи з можливістю відкриття цього завдання у Classroom.

Нефункціональні вимоги:

- використання протоколу OAuth 2.0 для входу через Google-акаунт;
- можливість розширення функціоналу;
- відгук системи не більше ніж 2-3 секунди.

Таким чином, сформульовані вимоги відображають реальні потреби цільової аудиторії — студентів і викладачів, залучених до проєктної командної роботи.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Середовище розробки

Інтегроване середовище розробки (IDE)— це програмне забезпечення, яке забезпечує розробників усіма необхідними інструментами для написання, редагування, компіляції, налагодження та запуску коду в межах одного інтерфейсу. Такі середовища значно підвищують ефективність програміста, завдяки автоматизації рутинних процесів і зручній інтеграції з іншими інструментами — системами контролю версій, системами збірки, тестування тощо[6].

2.1.1 IntelliJ IDEA

IntelliJ IDEA — це потужне інтегроване середовище розробки від компанії JetBrains, яке спеціалізується на розробці застосунків мовою Java, а також підтримує інші мови, зокрема Kotlin, Groovy, Scala, JavaScript, TypeScript та SQL. IntelliJ IDEA широко використовується професіоналами завдяки своїй інтелектуальній системі автодоповнення, навігації по коду, рефакторингу та глибокій інтеграції з популярними фреймворками й інструментами розробки[7].

IDE має зручний графічний інтерфейс, вбудовану підтримку системи збірки Maven та Gradle, інтеграцію з Git, Docker, базами даних, а також повноцінний інструментарій для тестування й профілювання застосунків. Однією з сильних сторін IntelliJ IDEA є її підтримка сучасних фреймворків, таких як Spring Boot, Hibernate, JPA, що особливо актуально в розробці web-застосунків корпоративного рівня.

Інтеграція з інструментами розробки, налагодження коду безпосередньо з IDE, зручне управління залежностями та автогенерація конфігурацій дозволяють розробнику зосередитись на логіці застосунку, а не на технічних деталях

середовища. IntelliJ IDEA — це вибір багатьох професіоналів завдяки своїй стабільності, функціональності та підтримці великої екосистеми Java.

2.2 Мови програмування

Мова програмування — це формалізований засіб спілкування з комп'ютером, який дозволяє розробникам задавати логіку обробки даних, управління потоками виконання та взаємодії з зовнішніми ресурсами. Сучасні мови програмування підтримують різні парадигми (об'єктно-орієнтовану, процедурну, функціональну тощо) і широко застосовуються в різних галузях — від веб-розробки до машинного навчання.

2.2.1 Java

Java — це об'єктно-орієнтована, високорівнева, строго типізована мова програмування, розроблена компанією Sun Microsystems у 1995 році (нині розвивається Oracle). Java працює за принципом «напиши один раз — запускай будь-де» (Write Once — Run Anywhere), що досягається завдяки використанню віртуальної машини Java (JVM), яка забезпечує кросплатформенність[8].

Java є однією з найпопулярніших мов для створення корпоративних застосунків, завдяки своїй стабільності, масштабованості та розвиненій екосистемі. Мова активно використовується у фінансовому секторі, телекомунікаціях, веброзробці, мобільних застосунках (через Android), а також у створенні серверної логіки.

Окрім базових можливостей, Java підтримує величезну кількість бібліотек і фреймворків, серед яких особливо популярними є Spring Framework, Hibernate, JUnit, Apache Maven та інші. Використання Java в проєкті дозволяє розробляти масштабовані, безпечні та легко підтримувані web-застосунки, що відповідає вимогам сучасної інженерії програмного забезпечення.

2.3 Вебфреймворки

Вебфреймворк — це програмний інструментарій, що полегшує створення, підтримку та розгортання web-застосунків, зокрема веб-сервісів, сайтів і API[9, с. 20]. Він забезпечує набір готових компонентів і архітектурних рішень, які допомагають розробникам зосередитися на логіці застосунку, а не на базовій інфраструктурі.

2.3.1 Spring Framework

Spring Framework — це потужний, гнучкий та широко використовуваний фреймворк для розробки Java- застосунків[10]. Він надає всебічну інфраструктуру для створення як простих web-застосунків, так і масштабованих корпоративних систем. Spring був створений у 2003 році з метою спростити розробку Java-застосунків, надаючи модульну, легко конфігурувану альтернативу складним стандартам Java EE.

Одна з ключових особливостей Spring — підтримка інверсії керування (IoC) та аспектно-орієнтованого програмування (AOP), що дозволяє розробникам писати гнучкий, розширюваний код із чітким розділенням відповідальностей. Крім того, фреймворк забезпечує глибоку інтеграцію з різними технологіями доступу до даних (JDBC, JPA, Hibernate), безпекою (Spring Security), обробкою транзакцій та розгортанням у хмарі.

Spring активно підтримується спільнотою та розвивається як одна з провідних платформ для розробки Java- застосунків у сучасному IT-середовищі. Його відкритий код, велика кількість документації та гнучка архітектура роблять його ідеальним вибором для створення надійного серверного програмного забезпечення.

2.3.2 Spring Boot

Spring Boot — це розширення фреймворку Spring для мови Java, створене з метою спростити розробку повнофункціональних web-застосунків із мінімальними

зусиллями з боку розробника. Представлений компанією Pivotal у 2014 році, Spring Boot дотримується принципу «конвенція понад конфігурацію», що дозволяє автоматично налаштовувати середовище на основі стандартних шаблонів[11, с. 5].

Фреймворк інтегрує ключові компоненти екосистеми Spring і значною мірою усуває необхідність у великій кількості конфігураційних файлів. Це дає змогу розробникам зосередитися на реалізації бізнес-логіки замість технічних налаштувань.

Однією з важливих особливостей Spring Boot є наявність вбудованого веб-сервера (наприклад, Tomcat або Jetty), що дозволяє запускати застосунок як самостійний виконуваний файл, без необхідності окремого розгортання на зовнішньому сервері. Це суттєво спрощує процеси запуску, тестування та розгортання.

Spring Boot має широку екосистему модулів, які покривають такі аспекти, як безпека, доступ до даних, побудова REST API, аналітика, моніторинг, асинхронна обробка подій тощо. Він також підтримує автоматичне керування залежностями за допомогою системи стартових залежностей (starters), що ще більше полегшує розробку.

2.4 Система управління базами даних

Система управління базами даних (СУБД) — це програмне забезпечення, яке забезпечує створення, обслуговування та маніпулювання базами даних. Вона виступає посередником між прикладними програмами та даними, дозволяючи ефективно зберігати, шукати, оновлювати і видаляти інформацію. СУБД підтримують організацію даних у структурованій формі, забезпечують цілісність, безпеку, багатокористувацький доступ і захист від збоїв. Розробники використовують СУБД для роботи з інформацією в багатьох сферах, включаючи web-застосунки, фінанси, науку та освіту.

2.4.1 MySQL

У розробці застосунку як основна система управління базами даних (СУБД) використовується MySQL — одна з найпопулярніших реляційних СУБД з відкритим кодом. MySQL забезпечує надійне зберігання структурованих даних, високу продуктивність і стабільність[12, с. 34].

MySQL повністю підтримує реляційну модель даних і стандарт мови SQL. Система гарантує дотримання властивостей ACID, що є критично важливим для забезпечення цілісності даних у транзакційних системах. MySQL дозволяє створювати складні запити, реалізовувати зовнішні ключі, індекси, тригери, процедури та функції, що дає змогу гнучко опрацьовувати дані.

У межах проєкту база даних використовується для зберігання інформації про користувачів, команди, проєкти, завдання, а також їхні статуси, коментарі та взаємодії. MySQL легко інтегрується з Java- застосунками через JPA (Java Persistence API) та ORM-фреймворк Hibernate, що значно спрощує обробку даних у застосунку.

2.4.2 Redis

Redis — це високопродуктивна система керування базами даних типу ключ-значення, що працює в оперативній пам’яті. Вона використовується переважно як кеш або брокер повідомлень, завдяки дуже швидкому доступу до даних[13].

Redis підтримує різноманітні структури даних, включаючи рядки, хеші, списки, множини та впорядковані множини. Вона ідеально підходить для реалізації черг, сесій користувачів, кешування часто використовуваних запитів або об'єктів, зменшуючи навантаження на основну базу даних.

У моєму проєкті Redis використовується для зберігання тимчасових даних, як-от токени авторизації, що дозволяє оптимізувати час відповіді сервера та зменшити навантаження на реляційну БД.

2.5 Технології взаємодії з базами

Технології взаємодії з базами даних — це програмні засоби, що забезпечують зручне і ефективне з’єднання між прикладним рівнем застосунку та СУБД. Вони автоматизують процес зчитування, запису, оновлення та видалення даних, спрощують роботу з транзакціями, зв'язками між таблицями і дозволяють працювати з об’єктами високого рівня замість сирих SQL-запитів. Ці технології значно прискорюють розробку, забезпечують контроль за цілісністю даних і полегшують масштабування застосунків.

2.5.1 JPA (Java Persistence API)

JPA — це офіційна специфікація Java EE (тепер Jakarta EE) для роботи з об’єктно-реляційним відображенням (ORM). Вона визначає набір анотацій, інтерфейсів та правил для взаємодії Java-об'єктів з реляційними базами даних.

Основна мета JPA — спростити зберігання, отримання, оновлення та видалення об'єктів у базі даних. JPA дозволяє працювати з базою, використовуючи стандартні інтерфейси та мову запитів JPQL (Java Persistence Query Language), яка схожа на SQL, але оперує сутностями замість таблиць.

JPA використовується для опису сутностей (Entity) застосунку, управління транзакціями та виконання запитів через репозиторії. Це забезпечує структуровану, незалежну від реалізації взаємодію з базою даних.

2.5.2 Hibernate

Hibernate — це одна з найпопулярніших реалізацій JPA, яка надає додаткові функціональні можливості, що виходять за межі базової специфікації. Hibernate автоматизує перетворення Java-об’єктів у таблиці бази даних і навпаки, керує кешуванням, lazy loading (відкладеним завантаженням), відстеженням стану об'єктів і багатьма іншими аспектами ORM[14].

Hibernate підтримує як стандарт JPA, так і власне API для роботи з даними. У ньому також реалізовано потужну систему конфігурацій, оптимізацію запитів та механізм міграції схеми БД.

Hibernate працює в парі з JPA, забезпечуючи фактичну реалізацію збереження даних. Завдяки цьому, розробка базується на стандартах JPA, але має доступ до додаткових функцій Hibernate, що покращує продуктивність та гнучкість взаємодії з базою даних.

2.6 Система контролю версій

Для керування версіями коду та організації командної роботи над проєктом було використано систему контролю версій **Git** у поєднанні з платформою **GitHub**. Git — це розподілена система контролю версій, яка дозволяє зберігати історію змін у коді, здійснювати паралельну розробку, повертатися до попередніх станів проєкту та уникати конфліктів під час командної роботи.

GitHub, у свою чергу, надає зручний веб-інтерфейс для управління репозиторіями, дозволяє створювати гілки, пул-реквести, обговорювати зміни та переглядати коміти. У рамках проєкту GitHub використовувався як основна платформа для зберігання коду, відстеження змін, а також для координації командної розробки та рев'ю коду.

Крім того, GitHub забезпечує інтеграцію з іншими інструментами CI/CD, що дозволяє автоматизувати процеси тестування та розгортання. Це робить платформу зручною для ефективної співпраці в команді та контролю якості коду на всіх етапах життєвого циклу розробки.

2.7 Технології авторизації та безпеки

Технології авторизації та безпеки відіграють ключову роль у захисті web-застосунків. Вони забезпечують ідентифікацію користувачів, контроль доступу до

ресурсів та захист переданих даних від несанкціонованого втручання. Надійна система безпеки включає механізми аутентифікації, авторизації, шифрування та перевірки токенів, що особливо важливо у розподілених системах.

2.7.1 JWT (JSON Web Token)

JSON Web Token (JWT) — це відкритий стандарт[15, с. 1], який визначає компактний і самодостатній спосіб безпечної передачі інформації між сторонами у вигляді JSON-об’єкта. JWT зазвичай складається з трьох частин, закодованих у форматі Base64: заголовка (header), корисного навантаження (payload) та підпису (signature).

Header містить тип токена (JWT) та алгоритм підпису (наприклад, RSA).

Payload включає основні дані (claims), такі як ідентифікатор користувача, час створення та час закінчення дії токена.

Signature забезпечує перевірку автентичності даних шляхом хешування попередніх частин разом із секретним ключем.

JWT активно використовується у механізмах авторизації. Після успішної автентифікації сервер генерує токен, який зберігається на стороні клієнта (наприклад, у cookie або localStorage) і надсилається разом із кожним наступним HTTP-запитом.

Завдяки цьому можна реалізувати безстейтову авторизацію — серверу не потрібно зберігати сесію в пам’яті, оскільки вся необхідна інформація міститься в токені.

Переваги JWT полягають у його компактності, простоті обробки, незалежності від мови програмування та можливості масштабованої обробки запитів без використання стану на сервері. Його також можна шифрувати, щоб приховати вміст, хоча зазвичай обмежуються лише підписуванням.

2.7.2 OAuth 2.0

OAuth 2.0 — це відкритий протокол авторизації[16, с. 1], який дозволяє застосункам отримувати обмежений доступ до захищених ресурсів користувача без

необхідності передавати його логін і пароль. Він був розроблений як гнучкий та безпечний механізм делегованої авторизації, що дозволяє стороннім сервісам діяти від імені користувача.

Основна суть роботи OAuth 2.0 полягає у видачі токена доступу (access token) клієнтському застосунку після дозволу від користувача. Цей токен використовується для автентифікованого доступу до API або інших захищених ресурсів. Крім того, може використовуватись токен оновлення (refresh token), який дозволяє клієнту отримати новий access token без повторного втручання користувача.

OAuth 2.0 підтримує декілька «грантів» або сценаріїв отримання токенів:

- Authorization Code Grant — найбільш поширений варіант для web-застосунків із серверною частиною.

- Implicit Grant — спрощений, але менш безпечний варіант для SPA (single-page applications).

- Client Credentials Grant — для взаємодії між сервісами без участі користувача.

- Resource Owner Password Credentials Grant — для довірених клієнтів, які отримують логін/пароль безпосередньо.

Завдяки гнучкості, розширюваності та високому рівню безпеки, OAuth 2.0 став де-факто стандартом авторизації в сучасних системах. Його використовують такі провайдери, як Google, Facebook, GitHub, Microsoft та інші. Протокол дозволяє інтегрувати сторонні сервіси, делегувати повноваження та реалізовувати SSO (Single Sign-On) рішення.

2.8 Технології контенерізації

Технології контейнеризації набули широкого поширення у розробці сучасного програмного забезпечення, особливо в умовах DevOps-культур та хмарних обчислень. Контейнеризація передбачає інкапсуляцію застосунку разом із

усім необхідним середовищем виконання — бібліотеками, залежностями, конфігураціями — в автономну й ізольовану одиницю, яка може бути запущена практично на будь-якому сервері з відповідним рушієм (наприклад, Docker Engine).

Перевагою контейнерного підходу є те, що контейнери легші та швидші за традиційні віртуальні машини, оскільки вони не потребують окремого ядра операційної системи. Це дозволяє суттєво зменшити накладні витрати, покращити продуктивність і прискорити масштабування. Завдяки цьому контейнери активно використовуються для реалізації мікросервісної архітектури, CI/CD-процесів, автоматизованого тестування, а також для гнучкого розгортання у різних середовищах — від локального комп'ютера до хмарних інфраструктур (AWS, Google Cloud, Azure).

2.8.1 Docker

Docker — це програмна платформа, яка дозволяє створювати, розгортати та запускати застосунки в ізольованих середовищах, відомих як контейнери. Контейнер — це легка, автономна одиниця програмного забезпечення, яка містить усе необхідне для роботи застосунку: вихідний код, бібліотеки, залежності, інструменти та конфігурації[17].

На відміну від віртуальних машин, контейнери мають менший об'єм і запускаються майже миттєво, що значно спрощує тестування, масштабування та розгортання застосунків.

Завдяки підтримці декларативного опису інфраструктури через Dockerfile та docker-compose, ця платформа стала стандартом для розгортання сучасних хмарних і мікросервісних рішень.

2.9 Інтеграція з зовнішніми сервісами

Інтеграція з зовнішніми сервісами є важливою складовою сучасних web-застосунків, оскільки дозволяє розширити функціональність системи без необхідності розробки всіх модулів із нуля. Замість дублювання вже наявних

рішень, застосунок підключається до перевірених API інших платформ, що дозволяє суттєво зменшити час розробки, підвищити надійність та забезпечити відповідність сучасним стандартам.

До типових прикладів таких сервісів належать:

- автентифікаційні системи (наприклад, OAuth2 від Google чи Facebook);
- освітні платформи, які надають доступ до навчальних курсів, контенту та оцінок;
- системи електронних платежів;
- сховища даних і хмарні API (наприклад, Google Drive, Dropbox);
- аналітичні та маркетингові інструменти.

Інтеграція передбачає роботу із зовнішнім API, яка зазвичай реалізується через HTTP-запити до RESTful-інтерфейсів, обробку JSON-відповідей, авторизацію через протоколи (зокрема OAuth 2.0) і налаштування відповідних дозволів. Основною перевагою є підвищення зручності для кінцевого користувача, який може працювати в одному інтерфейсі, не перемикаючись між платформами

2.9.1 Google Classroom API

Google Classroom API — це RESTful API, що надає стороннім застосункам можливість взаємодіяти з функціональністю освітнього сервісу Google Classroom.

Через цей інтерфейс можна отримати доступ до даних про курси, учасників, завдання, оцінки, повідомлення та інші компоненти навчального процесу. API підтримує CRUD-операції (створення, читання, оновлення, видалення), дозволяє створювати інтеграції для керування навчальними курсами, автоматизації адміністративних процесів, побудови аналітики або розробки застосунків для викладачів і студентів.

Аутентифікація здійснюється за допомогою протоколу OAuth 2.0, що гарантує безпечний контроль доступу до персоналізованих даних користувачів. Google Classroom API є частиною Google Workspace for Education і активно використовується в освітніх технологіях по всьому світу.

2.10 Інструменти проєктування інтерфейсу користувача

Інструменти проєктування інтерфейсу користувача відіграють ключову роль у розробці web-застосунків, оскільки саме через них користувач взаємодіє з функціоналом системи. Вони дозволяють створювати адаптивні, інтуїтивно зрозумілі та привабливі інтерфейси, які відповідають сучасним стандартам UX/UI-дизайну. У сфері Java-розробки для генерації HTML-інтерфейсів часто використовуються серверні шаблонізатори, які дозволяють поєднувати HTML-код із динамічними даними, що надходять із backend-частини застосунку.

2.10.1 Thymeleaf

Thymeleaf — це сучасний шаблонізатор для Java, який використовується для генерації динамічного HTML-контенту у серверних вебзастосунках. Його головною перевагою є можливість безпосередньо передавати змінні з Java-коду до HTML-сторінки та керувати їхнім відображенням за допомогою спеціальних атрибутів (th:text, th:if, th:each тощо)[18]. Це забезпечує зручну прив’язку даних до інтерфейсу користувача та дозволяє створювати логіку відображення без використання сторонніх мов шаблонів. Окрім того, HTML-код із розміткою Thymeleaf залишається валідним і доступним для попереднього перегляду навіть без запуску застосунку, що спрощує розробку і налагодження шаблонів.

Thymeleaf є потужним інструментом, який тісно інтегрується зі Spring Framework, особливо у Spring Boot-проєктах. Він підтримує двостороннє прив’язування даних, умовне відображення елементів, ітерації, валідацію форм, обробку фрагментів та багато іншого. Завдяки цьому розробники можуть легко створювати шаблони сторінок, що автоматично наповнюються динамічними даними з контролерів.

Використання Thymeleaf у цьому проєкті дозволяє реалізувати повноцінну модель MVC, де «View» представлено саме шаблонами, що генеруються на стороні сервера. Це підвищує надійність і контроль за відображенням даних.

3 ПРОЄКТУВАННЯ

3.1 Проєктування архітектури застосунку

Застосунок реалізований за класичною клієнт-серверною архітектурою, у якій логіка формування та обробки даних зосереджена на стороні сервера. Такий підхід забезпечує чітке розділення відповідальностей між клієнтом і сервером, покращуючи масштабованість, підтримуваність та безпеку системи.

Серверна частина реалізує архітектурний шаблон MVC (Model-View-Controller), що дозволяє ефективно організувати структуру застосунку. Контролери приймають вхідні HTTP-запити, викликають відповідні сервіси, які реалізують бізнес-логіку, та взаємодіють із рівнем доступу до даних. У відповідь сервер формує повноцінні HTML-сторінки й надсилає їх клієнту, що мінімізує потребу у складному клієнтському JavaScript і спрощує підтримку інтерфейсу.

Клієнтська частина працює у веб-браузері користувача та відображає згенеровані сервером сторінки. Завдяки використанню серверного шаблонізатора, динамічний вміст вбудовується безпосередньо під час генерації HTML, що дозволяє уникати зайвих асинхронних запитів та забезпечує стабільний і передбачуваний інтерфейс.

Обробка даних реалізується через шар доступу до бази, що абстрагує SQL-запити й забезпечує зручну роботу з реляційною моделлю. Для підвищення продуктивності застосунок використовує окреме швидкодійне сховище для тимчасових даних, що дозволяє кешувати проміжну інформацію, зменшити навантаження на основну базу даних та покращити час відгуку системи.

Сервер також може взаємодіяти з зовнішніми сервісами через API, що забезпечує можливість інтеграції з іншими платформами та обміну даними в реальному часі. На рисунку 3.1 зображено діаграму клієнт-серверної моделі застосунку.

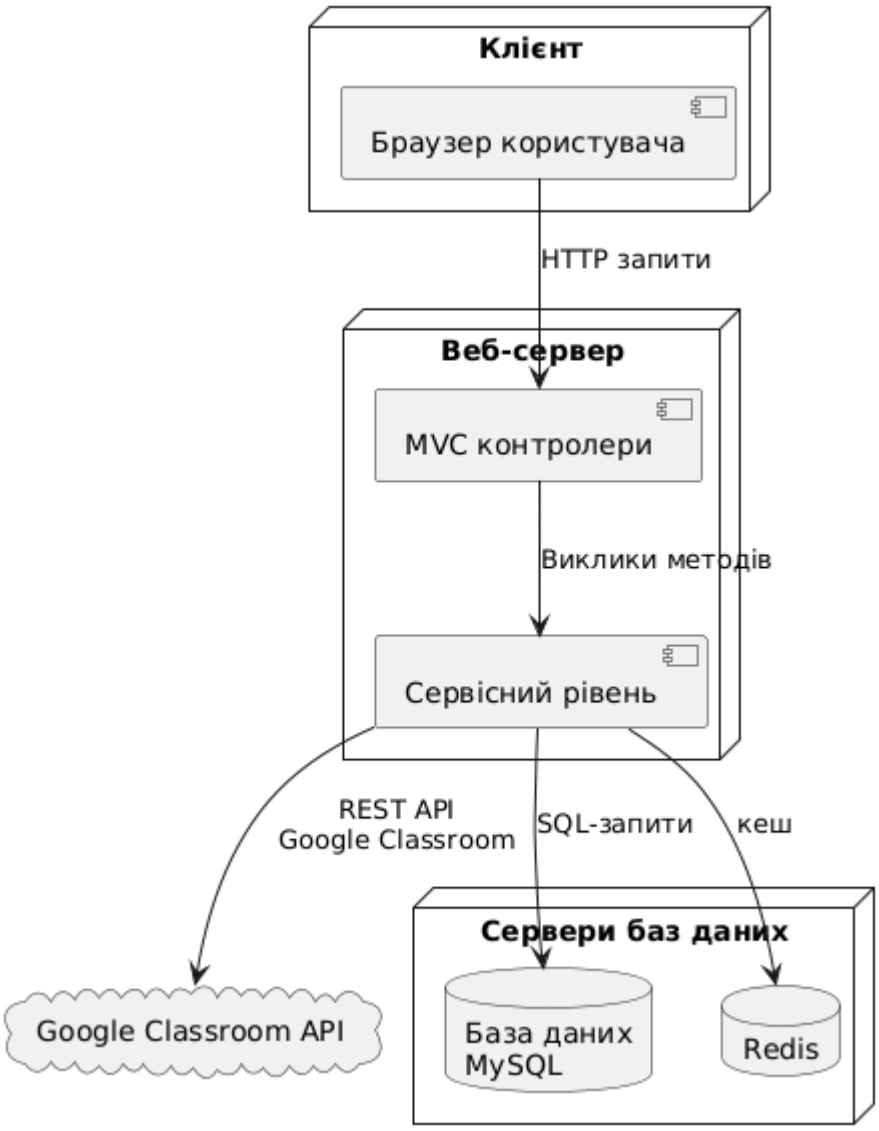


Рис. 3.1 Діаграма розгортання клієнт-серверної моделі

3.2 Архітектура серверної частини

Серверна частина застосунку побудована на основі фреймворку Spring Boot, що забезпечує високий рівень модульності, розширюваності та підтримки структурованої архітектури. Завдяки використанню шаблонізатора Thymeleaf у проєкті реалізована модель MVC, у межах якої контролери обробляють запити та формують HTML-сторінки, а моделі взаємодіють із базою даних. На верхньому рівні знаходяться загальні конфігурації, які охоплюють усе застосування, включаючи налаштування маршрутизації, безпеки, обробки HTTP-запитів та

інтеграцій з іншими сервісами. Нижче розміщені шари, що реалізують бізнес-логіку, доступ до даних, взаємодію з користувачем та забезпечення безпеки., обробку помилок, взаємодію з користувачем, доступ до даних і зовнішніх API.

Model (Моделі): Усі основні сутності системи описані у вигляді Java-класів, що реалізують об’єктно-реляційне відображення даних. Моделі відповідають таблицям бази даних MySQL і описують структуру інформації, зокрема поля, зв’язки між сутностями та типи даних. Використання JPA у поєднанні з Hibernate дозволяє повністю абстрагуватись від роботи з SQL-запитами та забезпечує надійний механізм взаємодії з базою.

Controller (Контролери): Контролери є посередниками між користувачем і внутрішніми процесами застосунку. Вони приймають HTTP-запити, передають їх на обробку в сервісний шар і формують відповідь у вигляді HTML-сторінки. У межах кожного контролера реалізовано функціонал, який відповідає за роботу з конкретними елементами застосунку, зокрема створення, оновлення, видалення та перегляд проєктів, користувачів, завдань. Таким чином, контролери виступають ключовим елементом зв’язку між інтерфейсом користувача та логікою системи.

Service (Сервіси): Сервісний шар реалізує основну логіку програми, обробляючи дані, які надходять від контролерів, та взаємодіючи з репозиторіями, DTO і зовнішніми API. У цьому шарі виконується створення та оновлення проєктів, керування завданнями, перевірка ролей користувачів. Сервіси дозволяють централізовано управляти логікою застосунку, ізолюючи її від технічних деталей реалізації доступу до даних, що сприяє зручності тестування та підтримки.

Repository (Репозиторії): Репозиторії відповідають за взаємодію з базою даних. Вони реалізовані за допомогою Spring Data JPA, що дозволяє виконувати стандартні CRUD-операції без необхідності написання SQL-запитів. Для кожної сутності створено окремий інтерфейс, який надає методи для пошуку, збереження, оновлення та видалення записів. У разі потреби також використовуються кастомні методи пошуку за параметрами. Завдяки репозиторіям забезпечується ефективне розмежування між логікою програми та шаром доступу до даних.

Security (Безпека): Пакет безпеки містить усі класи, необхідні для автентифікації та авторизації користувачів. Реалізовано підтримку реєстрації, входу в систему, перевірки валідності JWT-токенів, доступу до ресурсів залежно від ролей, а також конфігурації шифрування і дешифрування токенів. Система автентифікації базується на Spring Security з підтримкою авторизації через Google (OAuth2), що дозволяє забезпечити високий рівень захисту даних користувачів і контроль доступу до функціоналу.

DTO (Об’єкти передачі даних): DTO-класи використовуються для передачі даних між клієнтською частиною та сервером без розкриття внутрішньої структури моделей. Кожна сутність у системі має окремі DTO для створення, оновлення та перегляду повної інформації. Такий підхід дозволяє гнучко контролювати дані, що надсилаються чи повертаються, та підвищує безпеку і чистоту архітектури.

Redis: Для збереження токенів авторизації в системі використовується база даних Redis, яка працює як швидке тимчасове сховище. Реалізовано повноцінні CRUD-операції до Redis, що дозволяє записувати, оновлювати, перевіряти та видаляти токени користувачів. Redis забезпечує високу швидкодію при роботі з сесіями, що критично важливо для зручного користувацького досвіду.

Exception (Обробка винятків): У системі реалізовано окремий шар для обробки помилок, який містить кастомні винятки та глобальний обробник. Цей обробник перехоплює усі помилки, що виникають у процесі виконання запитів, та формує відповідні повідомлення для користувача та серверної частини. Наявність централізованої обробки винятків дозволяє зробити систему більш передбачуваною та полегшує налагодження.

Templates (HTML-шаблони): Усі сторінки, що відображаються користувачу, реалізовані за допомогою шаблонізатора Thymeleaf. У структурі проєкту передбачено шаблони для основної сторінки, сторінки профілю, сторінки проєкту, курсів, курсових робіт, сторінок запрошень, а також для повідомлень про помилки (401, 403, загальна помилка). Завдяки розділенню шаблонів за функціональністю забезпечується зручність у підтримці інтерфейсу та можливість гнучкого оновлення окремих елементів UI.

Config (Конфігурації): У пакеті конфігурацій містяться класи, відповідальні за загальні налаштування застосунку. Тут реалізовано обробку PUT та DELETE-запитів, конфігурацію маршрутизації та безпеки а також налаштування періодичних задач. Центральна конфігурація дозволяє гнучко змінювати поведінку програми без втручання в основну логіку.

Переваги такої архітектури:

- модульність — компоненти чітко розділені, що дозволяє розробляти і тестувати їх незалежно;

- гнучкість і масштабованість — завдяки Spring IoC-контейнеру легко замінювати реалізації або додавати нову функціональність;

- надійність — Spring надає багато вбудованих механізмів для обробки помилок, безпеки та транзакцій;

- швидкий розвиток — зменшення кількості шаблонного коду через Spring Boot, автоконфігурації, вбудовані засоби логування та тестування.

На рисунку 3.2 представлено діаграму пакетів архітектури серверної частини застосунку, яка ілюструє структуру основних компонентів, їхню взаємодію та розмежування відповідальності між шарами системи.

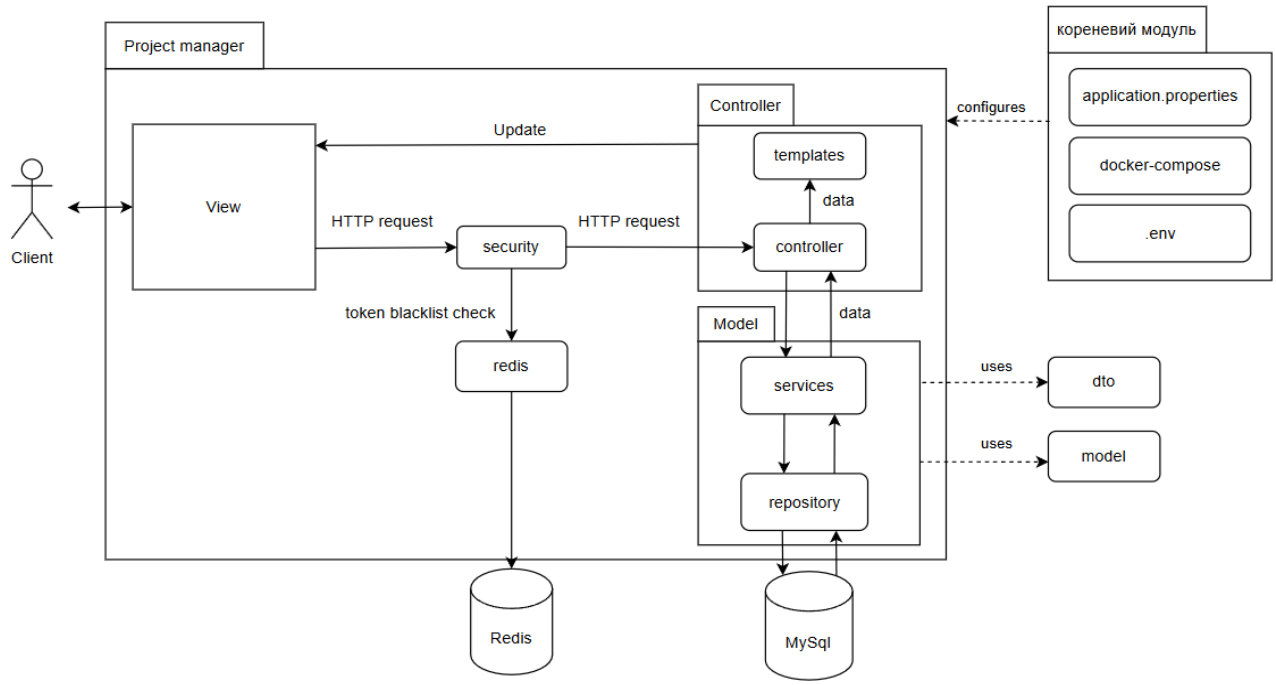


Рис. 3.2 Діаграма пакетів архітектури

3.3 Архітектура бази даних

Архітектура бази даних у даному проєкті побудована на основі реляційної моделі з використанням системи управління базами даних MySQL. Обрана модель є однією з найбільш поширених та ефективних для web-застосунків, оскільки дозволяє чітко структурувати дані, підтримує складні зв’язки між сутностями та забезпечує високу продуктивність при обробці великих обсягів інформації.

База даних складається з набору таблиць, що зберігають інформацію про основні сутності системи — користувачів, проєкти, завдання, ролі, запрошення. Таблиці створюються автоматично на основі Java-класів (моделей), які анотовані JPA-нотаціями та обробляються через ORM-бібліотеку Hibernate. Кожна модель описує окрему сутність і визначає відповідні поля, типи даних, первинні ключі та зв’язки між таблицями.

Кожна таблиця містить колонки, які відповідають полям об'єкта у коді. Дані зберігаються у вигляді рядків — кожен рядок представляє окремий екземпляр сутності. Усі CRUD-операції виконуються через репозиторії з використанням JPA, що дозволяє працювати з даними на рівні об’єктів, не використовуючи безпосередньо SQL-запити.

У системі реалізовані всі основні типи зв’язків між таблицями, характерні для реляційної моделі:

- один до багатьох (One-to-Many): один користувач може створювати багато проєктів або завдань;

- багато до багатьох (Many-to-Many): наприклад, користувачі можуть бути учасниками кількох проєктів, і кожен проєкт може включати декількох користувачів, зв'язок реалізується через проміжну таблицю.

Перевагами використаної архітектури бази даних є:

- цілісність даних – завдяки використанню зовнішніх ключів, обмежень і автоматичного управління зв’язками через ORM забезпечується послідовність та узгодженість інформації у таблицях;

- гнучкість – можливість створення складних запитів для отримання необхідної інформації, фільтрації, сортування або агрегації даних у зручному форматі;

- безпека – обмеження доступу до бази даних через механізми автентифікації, авторизації, а також ізоляцію через застосунок, дозволяє знизити ризики несанкціонованого доступу.

Завдяки використанню реляційної моделі в поєднанні з ORM-рівнем, ця архітектура є надійним, масштабованим та продуктивним рішенням для реалізації інформаційної системи керування командними студентськими проєктами.

Структурну організацію бази даних та взаємозв'язки між основними сутностями проєкту проілюстровано на діаграмі класів бази даних, що подана на рисунку 3.3.

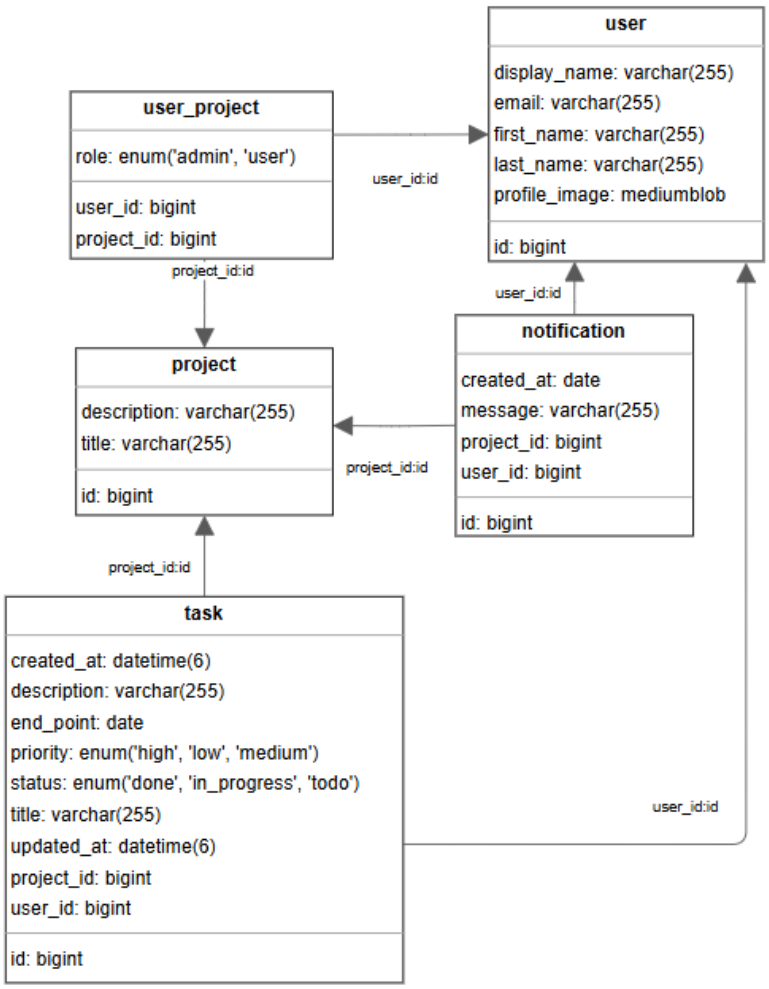


Рис. 3.3 Діаграма класів бази даних

Для відображення взаємодії між користувачем і системою на рисунку 3.4 було створено діаграму варіантів використання, яка демонструє ключові сценарії поведінки користувача у системі. В даному випадку актором є зареєстрований користувач, який взаємодіє із системою шляхом виконання певних дій через графічний інтерфейс. Система реагує на запити користувача, обробляє їх та виконує відповідні операції.

На діаграмі відображено можливості кожного користувача, зокрема перегляд курсів у Класрумі, редагування профілю, створення проєктів та запрошення інших користувачів до них, водночас також показано можливості андміністратора проєкту такі як редагування проєкту, видалення учасників з нього а також зняття користувачів із завдання.

Ця діаграма дозволяє представити функціональність системи на рівні бізнес-логіки з точки зору кінцевого користувача.

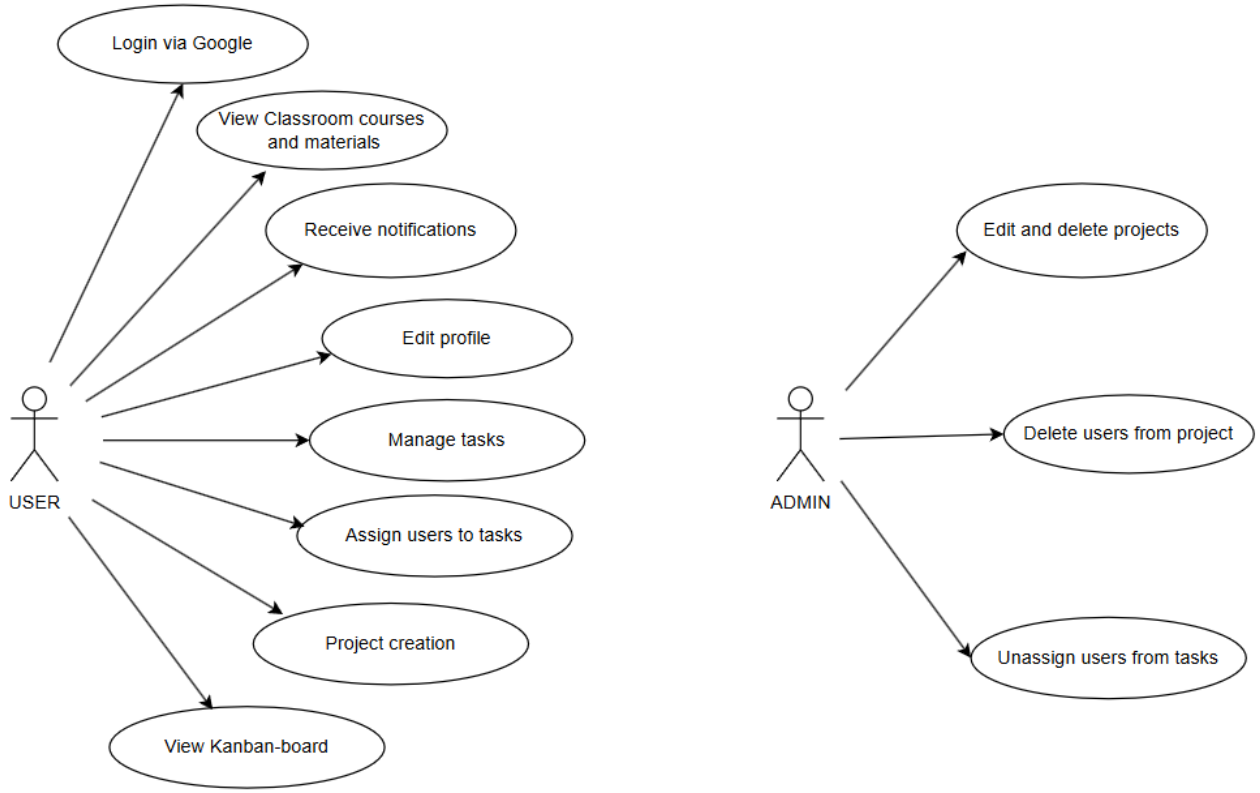


Рис. 3.4 Діаграма варіантів використання

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Структура програмного застосунку

Архітектура застосунку побудована на основі принципів модульності та логічного поділу відповідальностей. Усі функціональні компоненти згруповані у відповідні пакети, що полегшує навігацію в проєкті та забезпечує масштабованість системи. Код поділено на окремі частини: моделі (Model), контролери (Controller), сервіси (Service), репозиторії (Repository), DTO, конфігурації, обробники помилок, безпека, шаблони інтерфейсу та взаємодія з Redis і Google Classroom API.

Такий підхід дозволяє чітко розділити зони відповідальності між компонентами й спростити їх подальшу підтримку. Завдяки логічному поділу функціоналу на окремі модулі забезпечується зручність навігації в коді, можливість паралельної роботи над різними частинами системи, а також підвищується якість тестування і модифікації окремих блоків без впливу на інші частини проєкту. Це відповідає сучасним підходам до розробки масштабованих і підтримуваних web-застосунків.

На рисунку 4.1 нижче подано фрагменти структури проєкту у середовищі розробки, які демонструють загальну організацію системи. Такий вигляд структури демонструє логічну організацію внутрішніх компонентів системи, що дозволяє чітко розмежувати функціональні зони відповідальності — зокрема конфігурацію, бізнес-логіку, обробку помилок, доступ до даних, безпеку тощо.

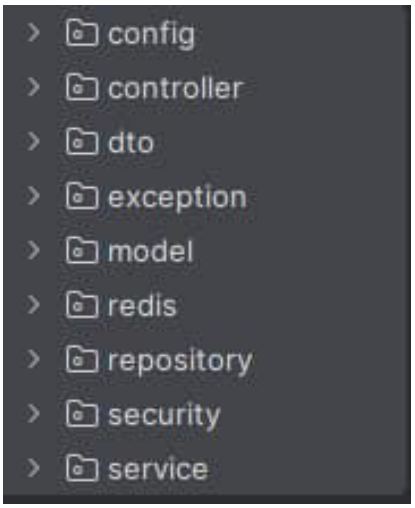


Рис. 4.1 Структура пакетів застосунку у середовищі IntelliJ IDEA

4.2 Реалізація основного функціоналу

4.2.1 Моделі сутностей в базі даних

Напочатку роботи були визначені основні моделі сутностей у системі, реалізовані за допомогою засобів об’єктно-реляційного відображення (ORM) на основі JPA та Hibernate. Усі сутності мають відповідні анотації для визначення зв’язків та ключів, що дозволяє точно відображати структуру бази даних у програмному коді та автоматично генерувати схему таблиць. Кожна модель відповідає окремій таблиці в реляційній базі даних, з визначеними зв’язками між ними: один до одного, один до багатьох або багато до багатьох.

Центральною є сутність User, яка представляє зареєстрованого користувача системи. Особливістю реалізації цієї моделі є те, що вона реалізує інтерфейс UserDetails, що забезпечує можливість інтеграції з Spring Security для подальшої автентифікації та авторизації. Завдяки цьому система підтримує перевірку облікових даних користувача, керування ролями та перевірку прав доступу безпосередньо на рівні об’єкта користувача.

Сутність Project описує командний проєкт, який може мати декілька учасників і включати набір завдань. Зв’язки між користувачами та проєктами реалізовано через окрему допоміжну сутність — UserProject, яка виступає як

таблиця зв’язків для відношення «багато до багатьох» між користувачами та проєктами. Оскільки цей зв’язок містить складовий ключ, у структурі UserProject використовується вбудований ідентифікатор UserProjectId, який об’єднує ідентифікатори користувача та проєкту. Клас UserProjectId позначено анотацією @Embeddable і він слугує як ключ для @EmbeddedId у самій сутності UserProject. Такий підхід дозволяє точно визначити складові первинного ключа і водночас зберігати додаткову інформацію про участь користувача в конкретному проєкті.

Сутність Task реалізує завдання, яке прив’язане до певного проєкту та може бути призначене конкретному користувачу. Між завданням і проєктом існує зв’язок типу «багато до одного», так само як і між завданням та виконавцем. Усі моделі містять необхідні анотації JPA для коректного формування таблиць і забезпечення цілісності даних.

Приклад реалізованої на рівну коду сутності з описаними зв’язками з іншими сутностями:

```
@Entity
@Data
@NoArgsConstructor
public class Project {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL,
orphanRemoval = true)
    private Set<UserProject> userProjects = new HashSet<>();
    @OneToMany(mappedBy = "project", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Task> tasks;
}
```

4.2.2 Управління проєктами та завданнями

На початковому етапі розробки система була реалізована як REST-застосунок, у якому вся взаємодія між клієнтом і сервером здійснювалась через HTTP-запити до REST-контролерів. Для управління проєктами та завданнями було створено окремі контролери, які підтримують повний набір CRUD-операцій: створення, читання, оновлення та видалення даних. Архітектура побудована відповідно до принципів REST, що дозволило забезпечити чітке розділення ресурсів та уніфікованість взаємодії з ними.

Кожен проєкт є окремим ресурсом і може містити кілька пов’язаних завдань. Створення нового проєкту або завдання реалізується через HTTP POST-запит, оновлення — через PUT, видалення — через DELETE, а перегляд інформації — через GET-запит. Всі запити обробляються контролерами, які передають їх у відповідні сервіси для подальшої бізнес-обробки. Структура запитів і відповідей стандартизована, а для обміну даними використовуються DTO-об’єкти а дані повертаються клаєнту у вигляді JSON.

Завдання можуть бути прив’язані до проєктів та призначені конкретним користувачам. Взаємозв’язок між проєктами та завданнями реалізовано на рівні моделей через анотації @OneToMany та @ManyToOne, що дозволяє зручно формувати відповідні запити до бази даних за допомогою JPA.

На рисунку 4.2 видно приклад запиту та відповіді системи.

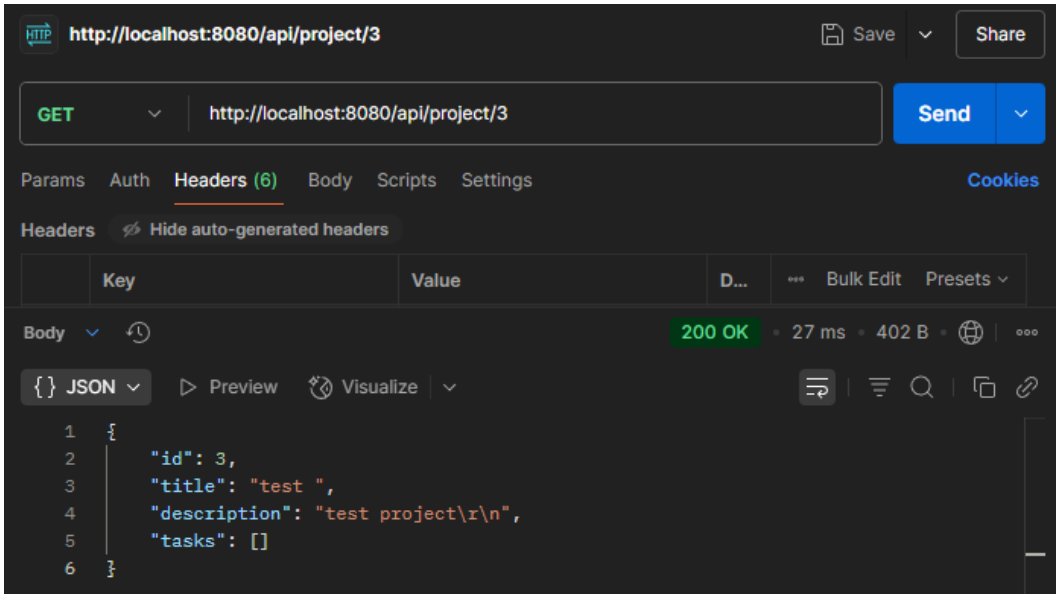


Рис. 4.2 Приклад GET запиту та відповіді рест-контролера

Вся взаємодія між частинами застосунку побудована на основі принципів розділення відповідальностей. Контролери відповідають за прийом HTTP-запитів та повернення відповіді клієнту. Вони не містять бізнес-логіки, а лише делегують виконання запитів відповідним методам сервісного шару. У сервісах реалізовано всю основну логіку, зокрема створення проєктів, додавання завдань, а також перевірку цілісності вхідних даних. Доступ до бази даних здійснюється через репозиторії, які взаємодіють із моделями за допомогою Spring Data JPA. Таким чином, реалізується чіткий і контрольований потік даних: Controller → Service → Repository.

На рисунку 4.3 видно, що для кожної сутності в базі даних було реалізовано свій контролер, сервіс та репозиторій, що дозволяє підтримувати чітке розділення логіки обробки запитів, бізнес-процесів та взаємодії з базою даних а також робить код чистішим.

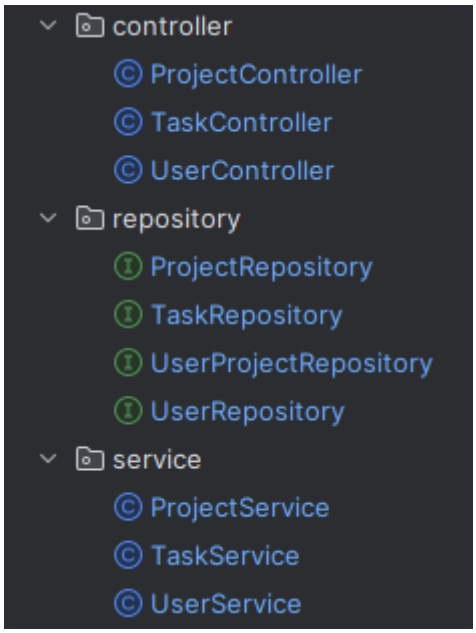


Рис. 4.3 Демонстрація класів основної логіки

4.2.3 DTO та обмін даними між клієнтом і сервером

Обмін даними між клієнтською та серверною частинами застосунку реалізовано з використанням об’єктів DTO (Data Transfer Object). DTO — це спеціальні класи, призначені для передачі даних між різними шарами застосунку,

зокрема між контролером і сервісом або між клієнтом і сервером. Вони використовуються для відділення внутрішніх моделей бази даних (сутностей) від зовнішніх форматів передачі інформації, що надходить або повертається в API.

Застосування DTO дозволяє уникнути прямої роботи з JPA-сутностями при обробці HTTP-запитів. Це забезпечує кращу безпеку, оскільки користувач не має доступу до внутрішньої структури об’єктів, а також дає змогу контролювати, які саме поля передаються, зберігаються або оновлюються. Крім того, DTO дозволяють виконувати валідацію вхідних даних за допомогою анотацій (наприклад, @NotNull, @Size, @Email), що значно підвищує стабільність і надійність системи.

Для кожної сутності в проєкті було створено окремі DTO-класи для різних типів операцій. Наприклад, для сутності Project використовуються:

- CreateProjectDto — для прийому даних під час створення нового проєкту;
- UpdateProjectDto — для редагування існуючого проєкту;

Аналогічні структури застосовуються і для інших сутностей — Task, User, UserProject тощо. Це дозволяє чітко відокремити логіку читання, запису й оновлення даних, а також робить API системи більш зрозумілим, передбачуваним і стабільним.

Усередині контролерів DTO-класи використовуються як вхідні аргументи методів для прийому даних з тіла HTTP-запиту (@RequestBody), а також як тип повернення (ResponseEntity<SomeDto>), що дозволяє явно керувати вмістом відповіді.

Усі перетворення між DTO та сутностями відбуваються у сервісному шарі — як вручну, так і за допомогою спеціалізованих утиліт для автоматичного мапінгу. У межах реалізації було використано два підходи: бібліотеку ModelMapper та інструмент MapStruct.

ModelMapper — це динамічна бібліотека мапінгу, яка виконує перетворення об’єктів під час виконання програми (runtime), автоматично зіставляючи поля з однаковими назвами. Вона є простою у використанні, але менш передбачуваною, коли потрібен контроль над окремими полями чи специфічною логікою.

Для більш точного та декларативного управління перетворенням використовувався MapStruct — компіляторний інструмент, який генерує код

мапінгу під час компіляції. MapStruct забезпечує високу продуктивність, передбачувану поведінку та дозволяє явно визначати логіку перетворення між об’єктами.

Приклад використання MapStruct у проєкті подано нижче. Інтерфейс ProjectMapper визначає метод оновлення даних проєкту на основі DTO з ігноруванням null-значень. Генерація реалізації відбувається автоматично під час компіляції:

```
@Mapper(componentModel = "spring")
public interface ProjectMapper {
    @BeanMapping(nullValuePropertyMappingStrategy =
    NullValuePropertyMappingStrategy.IGNORE)
    void mapProjectFromProjectDto(ProjectUpdateDto dto, @MappingTarget
Project project);
}
```

У цьому прикладі, якщо в DTO передано лише частину змінених даних, лише вони будуть оновлені в цільовому об’єкті Project, а інші залишаться без змін. Це особливо корисно для патч-запитів та часткового оновлення сутностей.

Використання таких утиліт дозволяє значно зменшити обсяг ручного коду, зменшити кількість помилок та забезпечити зручне масштабування, якщо в майбутньому з’являться нові поля чи типи DTO.

Таким чином, використання DTO в архітектурі застосунку відповідає принципам хорошого проєктування (зокрема, SRP — принципу єдиної відповідальності) і забезпечує зрозумілий, масштабований та безпечний механізм обміну даними в системі.

4.3 Реалізація автентифікації та авторизації

Безпечний доступ до системи реалізовано через інтеграцію механізмів автентифікації та авторизації за допомогою JWT (JSON Web Token) у поєднанні з

протоколом OAuth 2.0. Уся логіка побудована на базі бібліотек, що входять до складу Spring Security та OAuth2 Resource Server, які забезпечують гнучке керування токенами, контроль доступу та перевірку користувачів.

Resource Server — це компонент архітектури безпеки, який обробляє запити до захищених API та даних. Його ключова роль полягає у перевірці токена доступу (зазвичай JWT), який клієнт надсилає разом із запитом. Сервер перевіряє автентичність токена, його термін дії, підпис і наявність відповідних прав доступу (scopes або ролей). Якщо токен дійсний і користувач має необхідні дозволи, ресурсний сервер надає доступ до запитуваного ресурсу. Таким чином, він діє як захисний бар'єр між клієнтом і критично важливими частинами системи, забезпечуючи авторизований доступ відповідно до політики безпеки[19].

Для створення та перевірки токенів використовуються спеціалізовані компоненти — JWT Encoder і JWT Decoder, які автоматично надаються через Spring Security OAuth2. У конфігурації безпеки було визначено генерацію пари ключів RSA (2048 біт), які використовуються для асиметричного шифрування токена — відкритий ключ для перевірки підпису (декодування), приватний — для його створення (підпису).

Фрагмент конфігурації генерації ключів та ініціалізації JWT-компонентів виглядає наступним чином:

```
@Bean
public KeyPair keyPair() throws NoSuchAlgorithmException {
   var keyPairGenerator = KeyPairGenerator.getInstance("RSA");
   keyPairGenerator.initialize(2048);
   return keyPairGenerator.generateKeyPair();
}

@Bean
public JwtDecoder jwtDecoder(KeyPair keyPair) {
   return NimbusJwtDecoder.withPublicKey((RSAPublicKey)
   keyPair.getPublic()).build();
```

```
}

@Bean
public JwtEncoder jwtEncoder(KeyPair keyPair) {
   RSAKey jwk = new RSAKey.Builder((RSAPublicKey) keyPair.getPublic())
                        .privateKey(keyPair.getPrivate())
                        .build();
   var jwkSet = new ImmutableJWKSet<>(new JWKSet(jwk));
   return new NimbusJwtEncoder(jwkSet);
}
```

У результаті, при автентифікації користувача створюється токен, в якому вказуються основні claims — видавець, час створення, термін дії та ідентифікатор користувача. Ці дані кодуються за допомогою JwtEncoder, після чого токен зберігається в Redis, що дозволяє реалізувати його відкликання або перевірку дійсності під час подальших запитів:

Таким чином, токен генерується на основі приватного ключа та підписується, що унеможливлює його фальсифікацію. При отриманні запиту, JwtDecoder перевіряє підпис за допомогою публічного ключа, витягує claims і на їх основі формує об’єкт Authentication.

На момент обробки кожного запиту система перевіряє:

1. Чи дійсний формат токена та його підпис;
2. Чи не минув термін дії;
3. Чи зберігається токен у Redis.

Для реалізації авторизації через Google OAuth 2.0 у системі було використано стандартний механізм, що надається Spring Security. Перед початком інтеграції необхідно зареєструвати застосунок у Google Cloud Console, де створюється новий обліковий запис OAuth 2.0. Після реєстрації розробник отримує унікальні облікові дані: Client ID та Client Secret, які використовуються для ідентифікації застосунку під час автентифікації користувача. Ці значення зберігаються у конфігураційному файлі application.properties у такому вигляді:

spring.security.oauth2.client.registration.google.client-id=${CLIENT_ID}

spring.security.oauth2.client.registration.google.client-secret=${CLIENT_SECRET}

spring.security.oauth2.client.registration.google.scope=email,profile

OAuth 2.0 — це протокол авторизації, який дозволяє одному застосунку отримати обмежений доступ до захищених ресурсів іншого сервісу від імені користувача, без передачі його логіна та пароля. Основна ідея полягає в тому, що користувач підтверджує дозвіл на доступ, після чого зовнішній застосунок отримує токен доступу з чітко визначеним набором прав. У контексті розробленого застосунку використання OAuth 2.0 дозволило забезпечити захищену та зручну авторизацію користувачів через Google-акаунт. Замість створення окремої системи реєстрації, користувач отримує можливість увійти в систему, використовуючи свій обліковий запис Google, не передаючи паролі стороннім сервісам. При цьому застосунок має доступ лише до обмеженого обсягу інформації — електронної пошти та загальнодоступного профілю користувача (відповідно до налаштувань scope). Такий підхід забезпечує високий рівень безпеки персональних даних, спрощує користувацький досвід та зменшує ризики витоку облікових даних[20].

На рисунку 4.4 наведено механізм авторизації за допомогою OAuth 2.0.

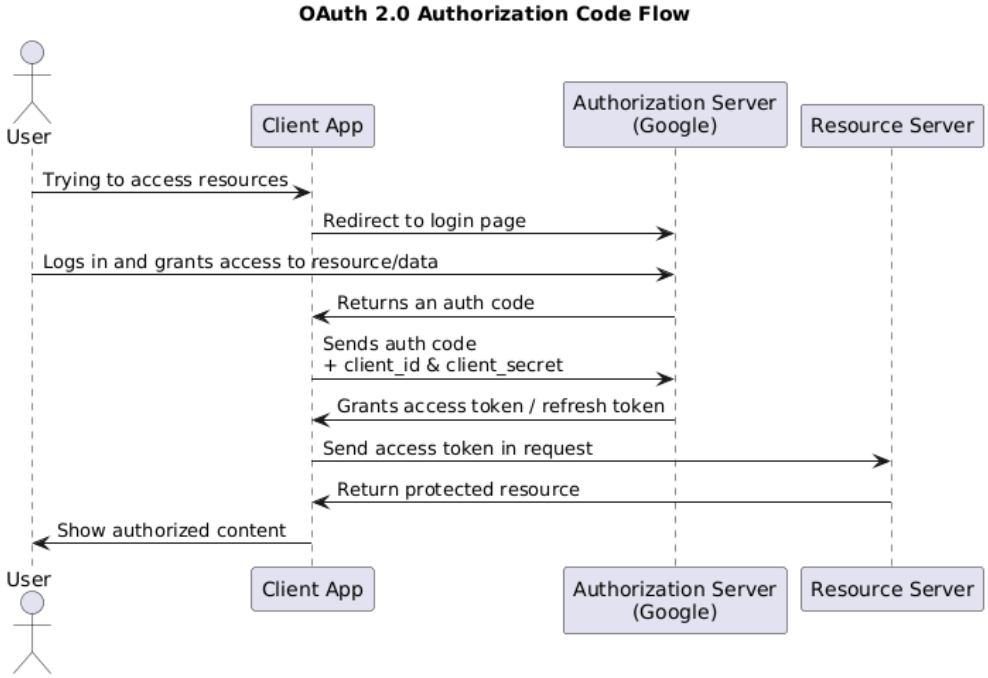


Рис. 4.4 Діаграма взаємодії компонентів під час OAuth 2.0 авторизації

4.4 Система безпеки web-застосунку

Система безпеки розробленого web-застосунку реалізована на базі фреймворку Spring Security і включає в себе механізми автентифікації, авторизації, управління токенами, обробки помилок доступу та взаємодії з зовнішніми сервісами. Архітектура побудована таким чином, щоб поєднати зручність для кінцевого користувача з високим рівнем захисту даних і гнучкістю налаштувань.

У реалізації системи безпеки використано два взаємодоповнюючі підходи:

- автентифікація через Google за протоколом OAuth 2.0, яка забезпечує надійний та швидкий вхід до системи без потреби реєстрації нового облікового запису;

- авторизація через JWT-токени, які генеруються після входу, зберігаються у Redis та перевіряються при кожному запиті до захищених ресурсів.

Центральним елементом конфігурації безпеки є клас SecurityConfig, який визначає правила авторизації, список дозволених маршрутів, фільтри безпеки та механізми шифрування та перевірки токенів. Компонент TokenService реалізує логіку генерації, валідації та відкликання JWT-токенів, а також їх збереження у Redis-сховищі.

Клас OAuth2AuthenticationSuccessHandler відповідає за обробку дій після успішної авторизації через Google, зокрема за створення токена та його передачу клієнту. Компонент CustomJwtAuthenticationConverter трансформує JWT у внутрішню структуру Authentication, зрозумілу для Spring Security. Авторизаційні запити до Google налаштовуються через CustomAuthorizationRequestResolver, що дає змогу гнучко змінювати параметри запиту.

Користувацький логін і вихід із системи реалізовано через AuthController, який надає методи логіну та логауту. Токени можуть передаватися або через HTTP-заголовки, або через cookie, що витягуються за допомогою CookieBearerTokenResolver. Додаткові компоненти — CustomAccessDeniedHandler та CustomEntryPoint — забезпечують належну обробку помилок доступу: наприклад, спроб без авторизації або без належних прав.

Управління даними користувача здійснюється через класи UserService та CustomOAuth2UserService, які забезпечують отримання, створення або оновлення облікових записів під час OAuth2-автентифікації.

Для кращого розуміння структури підсистеми безпеки на рисунку 4.5 представлено діаграму класів, яка ілюструє взаємозв'язки між ключовими компонентами: конфігураційними класами, сервісами, контролерами, обробниками винятків, токен-сервісом та інструментами роботи з автентифікацією:

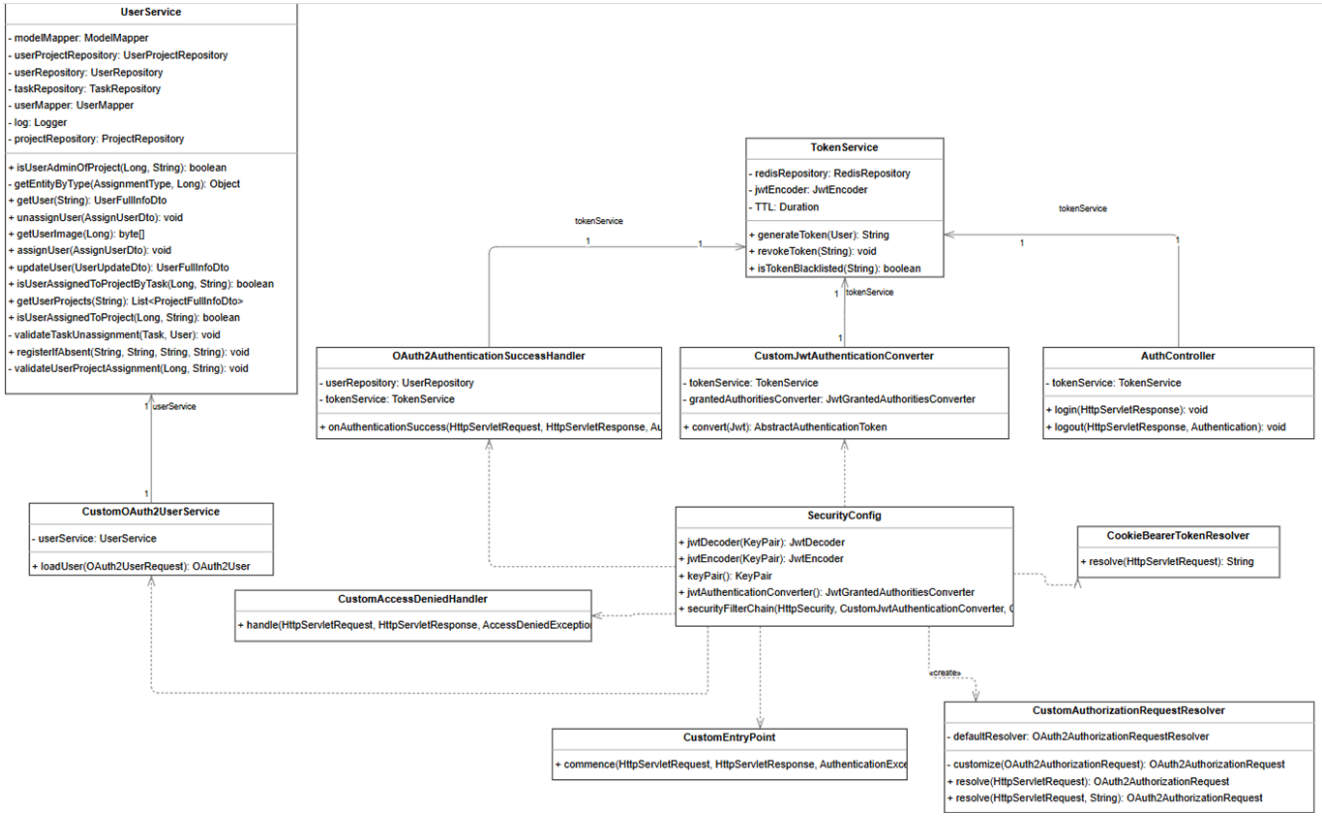


Рис. 4.5 Діаграма класів безпеки

4.5 Реалізація інтерфейсу користувача

Після завершення реалізації основної логіки REST-застосунку, включаючи управління проєктами, завданнями, безпековий модуль та інтеграцію з Google Classroom API, постала потреба у створенні зручного і зрозумілого інтерфейсу користувача, який би забезпечував ефективну взаємодію з системою та покривав

повний функціонал з боку фронтенду. Технічною передумовою стало те, що більшість запитів уже оброблялися на стороні сервера через REST-контролери, а отже інтерфейс мав бути тісно пов’язаний із серверною логікою.

Для реалізації представлення (View) було обрано шаблонізатор Thymeleaf, який є рекомендованим рішенням у екосистемі Spring Boot. Основними перевагами такого підходу є:

- глибока інтеграція з Spring MVC, що дозволяє безпосередньо передавати об'єкти з контролерів у шаблони;

- простота у створенні динамічних HTML-сторінок, які поєднують статичний контент із логікою на стороні сервера;

- можливість роботи без клієнтського JavaScript-фреймворку, що зменшує складність підтримки;

- швидка розробка MVP або навчальних застосунків, де важлива зрозумілість та стабільність.

Завдяки використанню Thymeleaf було створено набір сторінок, кожна з яких відповідає за окремий аспект функціональності: головна сторінка, авторизація, список проєктів, перегляд завдань, профіль користувача, перегляд курсів Google Classroom тощо. Всі шаблони адаптовані до взаємодії з контролерами, які передають у модель необхідні дані (курси, завдання, список учасників, дедлайни тощо). Шаблони сторінок зображені на рисунку 4.6.

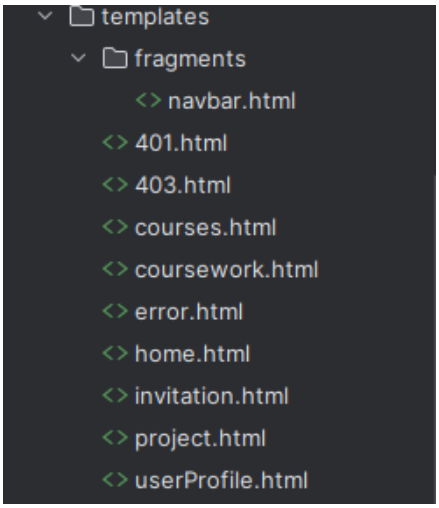


Рис. 4.6 Файли HTML сторінок

У початковій реалізації система була побудована за принципами REST-архітектури, де всі запити оброблялися через REST-контролери з використанням анотацій @RestController, @RequestBody, і відповіді повертались у форматі JSON. Однак після впровадження інтерфейсу на основі шаблонів Thymeleaf виникла необхідність адаптувати логіку до моделі Spring MVC, де замість JSON-відповідей користувач отримує HTML-сторінки через браузер.

У зв'язку з цим частина REST-контролерів була перероблена на звичайні MVC-контролери (@Controller), які повертали представлення у вигляді назви HTML-шаблону, а не JSON-об'єкта. Відповідно, змінилися і сигнатури методів.

На рисунку 4.7 наведено приклад початкового варіанта REST-методу оновлення проєкту. Метод позначено анотацією @PutMapping, він приймає дані у форматі JSON через анотацію @RequestBody та повертає DTO-об'єкт із розширеною інформацією про проєкт:

```
@Operation(summary = "Update existing project")  new *
@ApiResponse(responseCode = "200", description = "Project updated")
@ApiResponse(responseCode = "400", description = "Invalid project data")
@ApiResponse(responseCode = "404", description = "Project not found")
@PutMapping
@PreAuthorize("@userService.isUserAdminOfProject(#projectUpdateDto.id, authentication.name)")
public ProjectFullInfoDto updateProject(@RequestBody ProjectUpdateDto projectUpdateDto) {
    return projectService.updateProject(projectUpdateDto);
}
```

Рис. 4.7 Метод оновлення проєкту у реалізації підходу REST

У результаті рефакторингу, як показано на рисунку 4.8, метод було адаптовано для обробки HTML-форм. Замість @RequestBody використовується анотація @ModelAttribute, яка дозволяє отримати дані з параметрів PUT-запиту HTML-форми. Також змінено тип повернення: замість DTO метод повертає строку з інструкцією для перенаправлення на відповідну сторінку, що відповідає підходу Spring MVC:

```java
@Operation(summary = "Update existing project")  YegorPopinako
@ApiResponse(responseCode = "200", description = "Project updated")
@ApiResponse(responseCode = "400", description = "Invalid project data")
@ApiResponse(responseCode = "404", description = "Project not found")
@PutMapping
@PreAuthorize("@userService.isUserAdminOfProject(#projectUpdateDto.id, authentication.name)")
public String updateProject(@ModelAttribute ProjectUpdateDto projectUpdateDto) {
    projectService.updateProject(projectUpdateDto);
    return "redirect:/api/project/" + projectUpdateDto.getId();
}
```

Рис. 4.8 Метод оновлення проєкту у реалізації підходу MVC

Для зручності редагування проєкту без необхідності переходу на окрему сторінку, на вебінтерфейсі реалізовано модальне вікно, яке відкривається безпосередньо на сторінці проєкту. Вікно містить HTML-форму, для заповнення даних для обраного проєкту. При введені нових даних та натисканні кнопки збереження відповідний контролер обробляє PUT-запит, після чого відбувається оновлення даних у базі та автоматичне повернення на ту ж сторінку.

Такий підхід покращує користувацький досвід, дозволяючи швидко вносити зміни без перезавантаження інтерфейсу.

Приклад сторінки з відкритим модальним вікном для редагування проєкту на сторінці цього ж проєку наведено на рисунку 4.9.

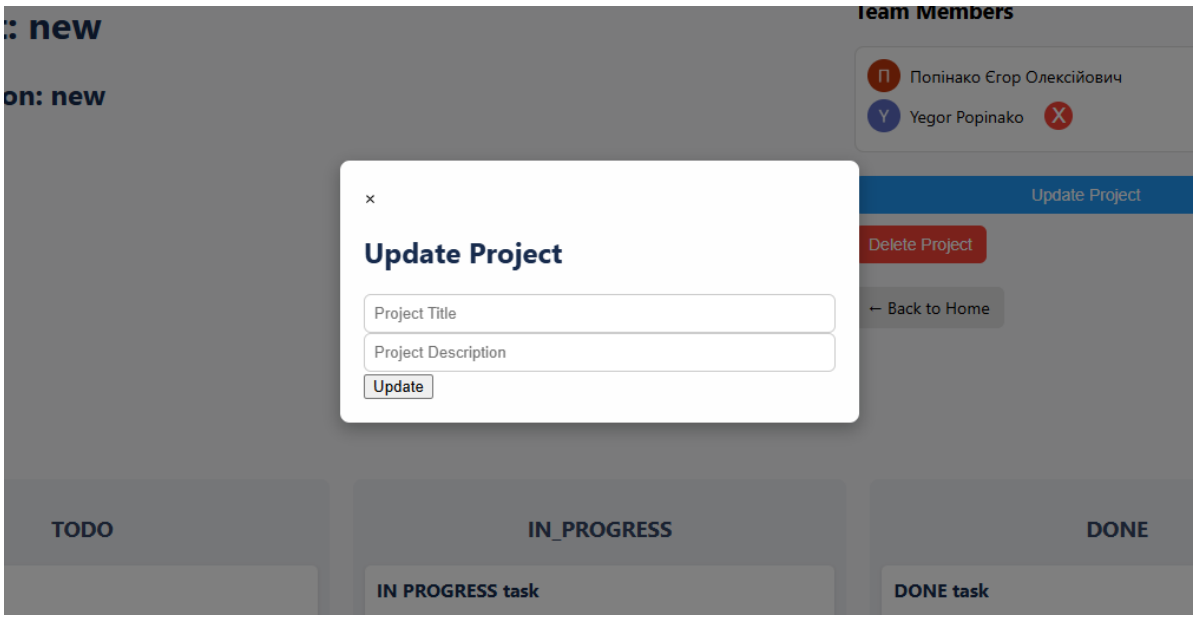


Рис. 4.9 Модальне вікно для оновлення даних проєкту

Ще одним важливим аспектом при переході від REST до MVC була організація передачі даних від контролера до HTML-шаблону. У REST-контролерах відповідь поверталася у вигляді JSON-об’єкта, а в Spring MVC структура даних передається до шаблонізатора через об’єкт Model.

На рисунку 4.10 наведено приклад методу, що відповідає за відображення профілю користувача. Метод приймає об’єкт Authentication, який містить ім’я поточного авторизованого користувача, а також об’єкт Model, до якого додаються дані, що мають бути доступними у шаблоні.

```
@GetMapping  YegorPopinako *
public String getUser(Authentication authentication, Model model) {
    UserFullInfoDto userDto = userService.getUser(authentication.getName());
    model.addAttribute( attributeName: "userDto", userDto);
    return "userProfile";
}
```

Рис. 4.10 Метод отримання та передачі даних про користувача

У цьому прикладі з бази даних завантажується інформація про користувача (userDto). Об’єкт передаються у модель і стає доступними на сторінці userProfile.html через відповідні вирази Thymeleaf (наприклад, ${userDto.email}). На цій сторінці користувач бачить своє ім’я, прізвище, відображуване ім’я (Display Name), електронну пошту, а також зображення профілю. Інформація завантажується з бази даних і передається у представлення через об’єкт Model, з якого доступна у шаблоні через вирази Thymeleaf (наприклад, ${userDto.email}).

Редагування здійснюється безпосередньо на сторінці: відповідні дані можна змінювати у текстових полях, після чого зміни надсилаються на сервер. Це забезпечує зручний спосіб оновлення інформації без необхідності переходу на окремі екрани або перезавантаження сторінки.

Візуальне відображення цієї функціональності наведено на рисунку 4.11, де зображено інтерфейс користувача зі структурованим представленням особистих даних.

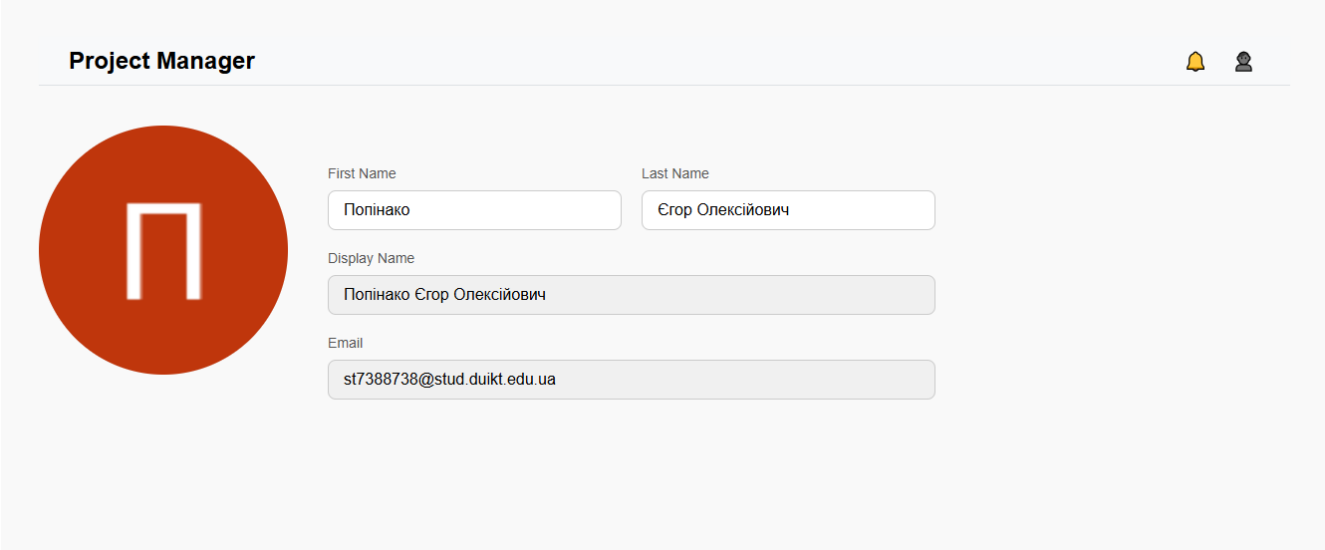


Рис. 4.11 Сторінка профілю користувача

Найбільшу кількість даних таким чином передається на головну сторінку застосунку відображає персоналізовану інформацію для авторизованого користувача. На ній розміщено блок з активними проєктами, у яких бере участь користувач, кнопки для створення проєктами, та для виходу із системи, і кнопка для перегляду курсів Google Classroom.

Верхня частина сторінки (навігаційна панель) містить посилання для переходу до профілю користувача, головну сторінку та модуль зі сповіщеннями. Завдяки адаптивному дизайну та логічній структурі головної сторінки, користувач може швидко орієнтуватися в поточному статусі проєктної діяльності та оперативно переходити до потрібних дій.

На рисунку 4.12 зображено головну сторінку із всією інформацією яка стосується поточного користувача. Ця інформація формується на стороні сервера, передається до шаблону за допомогою моделі Model і далі відображається у відповідних HTML-елементах з використанням синтаксису Thymeleaf.

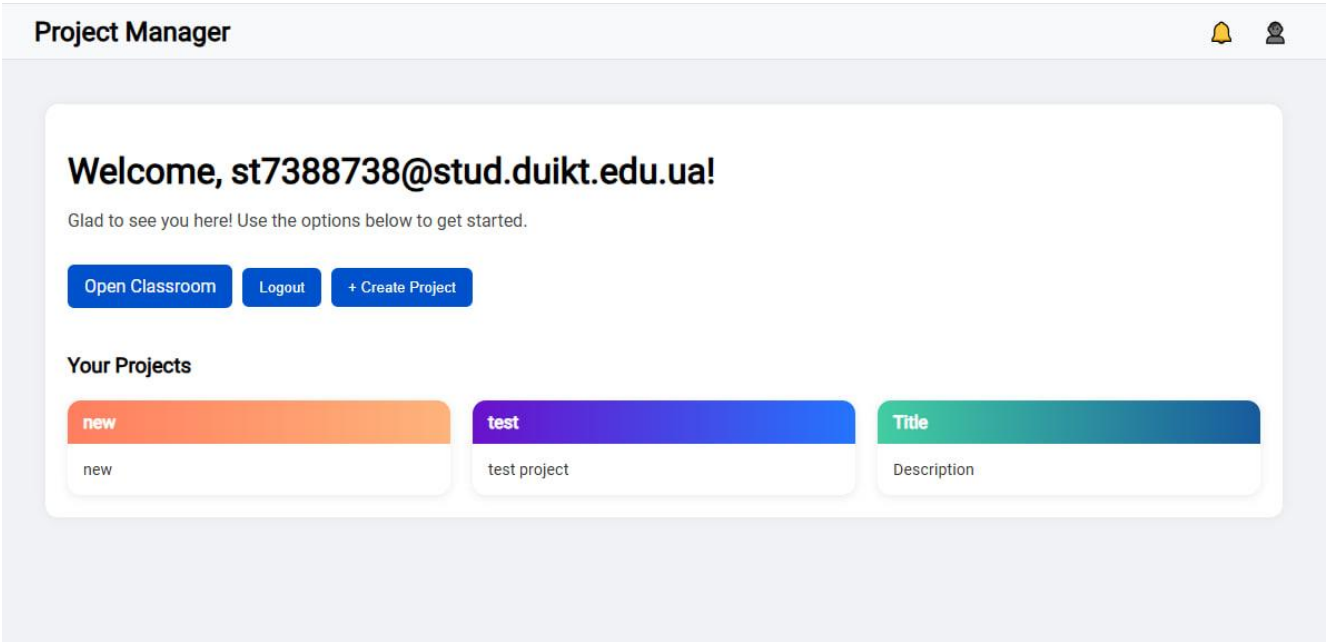


Рис. 4.12 Головна сторінка застосунку

Таким чином, відбулося зміщення парадигми з REST-орієнтованого контролера на контролер, що обслуговує вебінтерфейс, з метою прямої взаємодії з HTML-шаблонами, покращення користувацького досвіду та забезпечення узгодженості між бізнес-логікою і відображенням.

Для більш детального моделювання одного з ключових сценаріїв — створення нового проєкту користувачем — створено діаграму послідовностей, яка демонструє процес виконання цієї дії на рівні взаємодії об'єктів.

У цьому випадку користувач надсилає POST запит на створення проєкту через інтерфейс, який передається контролеру. Контролер обробляє запит і викликає відповідний метод у сервісному шарі. Сервіс, у свою чергу, створює модель об’єкта проєкту, передає його до репозиторію, де зберігаються дані в базі. У результаті користувач отримує підтвердження про успішне створення проєкту або повідомлення про помилку, якщо введені дані були некоректними.

Діаграма послідовностей відображає логіку обробки запиту від початку до кінця, включаючи всі основні етапи: взаємодію з формою, валідацію, обробку даних і збереження результату.

На рисунку 4.13 зображено послідовність дій під час створення проєкту:

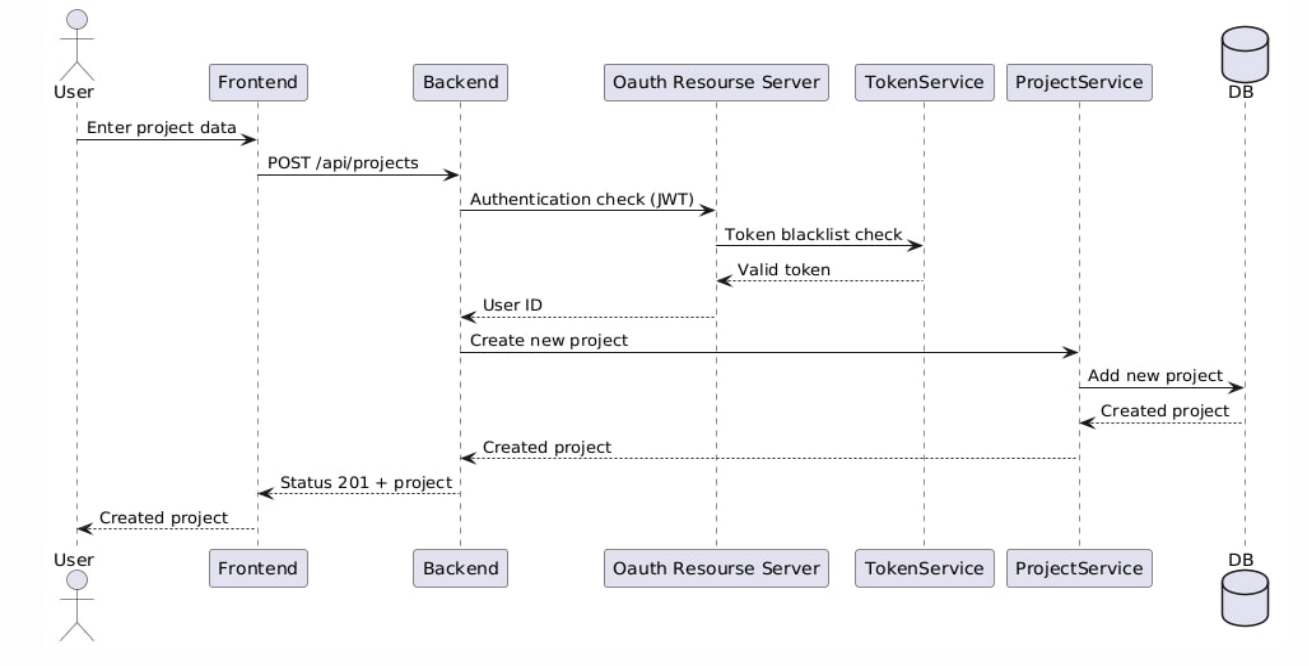


Рис. 4.13 Діаграма послідовності при створенні нового проєкту

4.6 Інтеграція з зовнішніми сервісами

Інтеграція з зовнішніми сервісами є важливою складовою функціональності сучасних web-застосунків, оскільки дозволяє отримувати доступ до актуальних даних із перевірених джерел, підвищуючи ефективність взаємодії з користувачем. У рамках розробленої системи було реалізовано інтеграцію з Google Classroom API, яка дає змогу підключеним користувачам переглядати свої навчальні курси, завдання та пов’язані матеріали безпосередньо з інтерфейсу застосунку.

Для налаштування інтеграції з Google Classroom першим етапом є реєстрація застосунку в Google Cloud Console. Під час реєстрації розробник обирає необхідні API (у даному випадку — Google Classroom API), створює облікові дані та отримує унікальні значення Client ID і Client Secret, які надалі використовуються для ініціалізації автентифікації користувачів. Крім того, визначається список дозволених адрес перенаправлення (redirect URI), а також перелік дозволів доступу (областей доступу), які застосунок буде запитувати у користувача, таких як перегляд курсів, завдань і навчальних матеріалів.

У контролері, що відповідає за авторизацію, попередньо визначається список дозволів (scopes), які вказують на ті ресурси користувача, до яких застосунок потребує доступу. Потім формується запит до авторизаційного сервера Google — спеціальне посилання, на яке буде перенаправлено користувача:

```
@GetMapping("/classroom-auth")
public void redirectToGoogle(HttpServletResponse response) throws IOException
{
   String authUrl = UriComponentsBuilder
      .fromUriString("https://accounts.google.com/o/oauth2/v2/auth")
      .queryParam("client_id", clientId)
      .queryParam("redirect_uri", REDIRECT_URI)
      .queryParam("response_type", "code")
      .queryParam("scope", String.join(" ", SCOPES))
      .queryParam("access_type", "offline")
      .queryParam("prompt", "consent")
      .build()
      .toUriString();
   response.sendRedirect(authUrl);
}
```

Цей запит ініціює вхід користувача до облікового запису Google та запитує дозвіл надати застосунку Project Manager доступ до ресурсів, перелічених у дозволах, що зображено на рисунку 4.14. Після підтвердження згоди користувача Google перенаправляє його назад до застосунку, передаючи авторизаційний код.

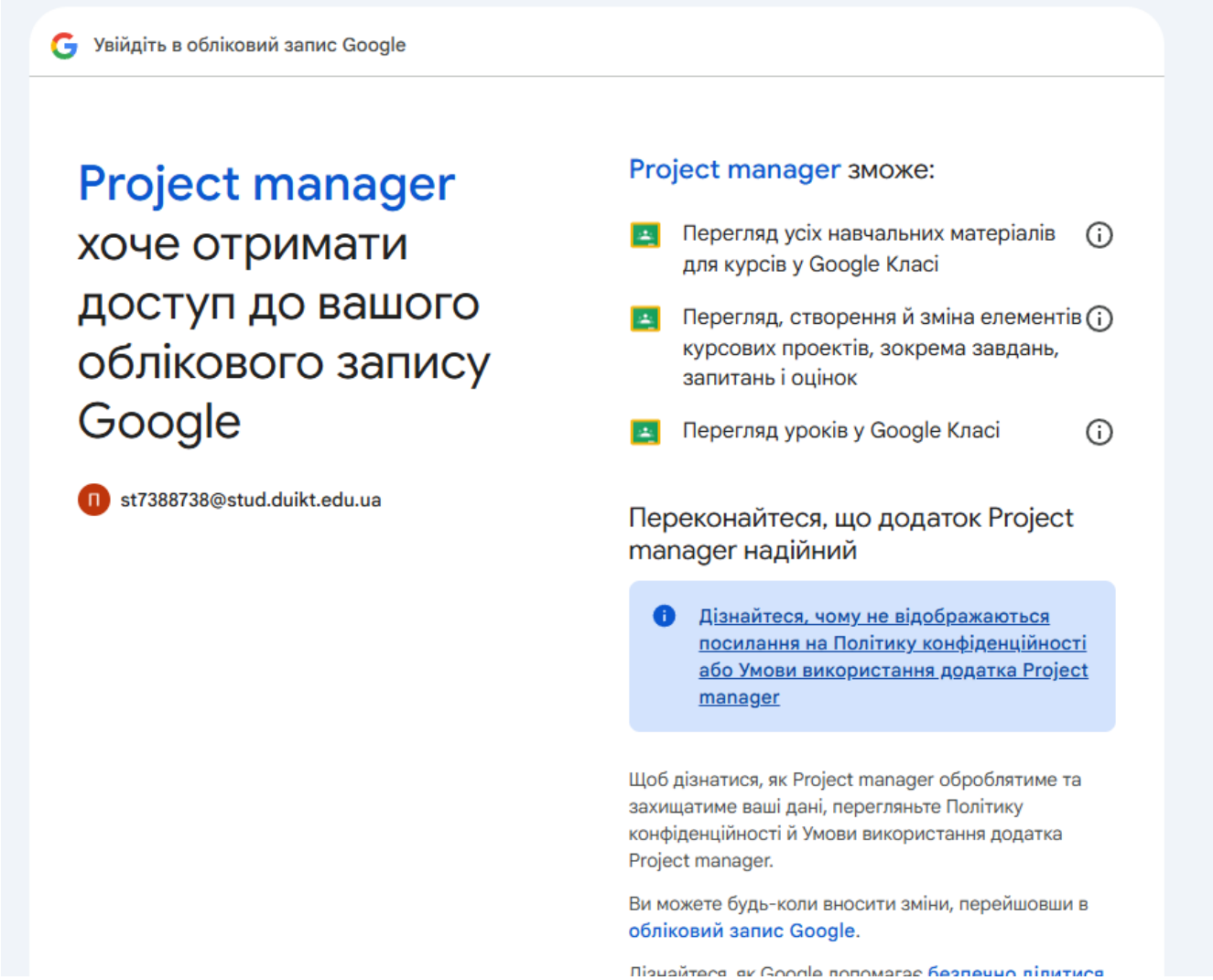


Рис. 4.14 Сторінка запиту застосунку над керуванням ресурсами

У наступному контролері авторизаційний код обмінюється на облікові дані (credentials), що включають access token та, за потреби, refresh token. Ці облікові дані зберігаються у сесії користувача для подальшого використання:

```
@GetMapping("/oauth2/callback/classroom")
public String handleCallback(@RequestParam("code") String code, HttpSession
session) throws Exception {
    NetHttpTransport transport = GoogleNetHttpTransport.newTrustedTransport();
    GoogleTokenResponse tokenResponse = new
GoogleAuthorizationCodeTokenRequest(
        transport,
        JSON_FACTORY,
```

```
            "https://oauth2.googleapis.com/token",
            clientId,
            clientSecret,
            code,
            REDIRECT_URI
        ).execute();
        GoogleCredential credential = new GoogleCredential.Builder()
            .setTransport(transport)
            .setJsonFactory(JSON_FACTORY)
            .setClientSecrets(clientId, clientSecret)
            .build()
            .setFromTokenResponse(tokenResponse);
        session.setAttribute("googleCredential", credential);
        return "redirect:/classroom/courses";
    }
```

Після успішної авторизації користувач перенаправляється до ендпоінту який формує та надає визначену раніше область доступів, а саме інформацію про курси. У відповідному методі контролера з сесії дістаються збережені облікові дані, на основі яких формується об’єкт для взаємодії з Google Classroom API. Відбувається ініціалізація клієнта Google і надсилається запит на отримання списку курсів:

```
GoogleCredential credential = (GoogleCredential)
session.getAttribute("googleCredential");
NetHttpTransport transport = GoogleNetHttpTransport.newTrustedTransport();
Classroom classroom = new Classroom.Builder(transport, JSON_FACTORY,
credential)
    .setApplicationName("Project Manager")
    .build();
List<Course> courses = classroom.courses().list().execute().getCourses();
```

Після успішного отримання облікових даних з Google API, користувач перенаправляється до сторінку, на якій відображається список курсів, пов’язаних з

його обліковим записом Google Classroom. Кожен курс містить назву, секцію (наприклад, групу або курс навчання) та є інтерактивним посиланням, яке веде до більш детальної інформації про навчальний процес. Як показано на рисунку 4.15, дані виводяться у вигляді чисто структурованих карток, що дозволяє легко сприймати перелік курсів.

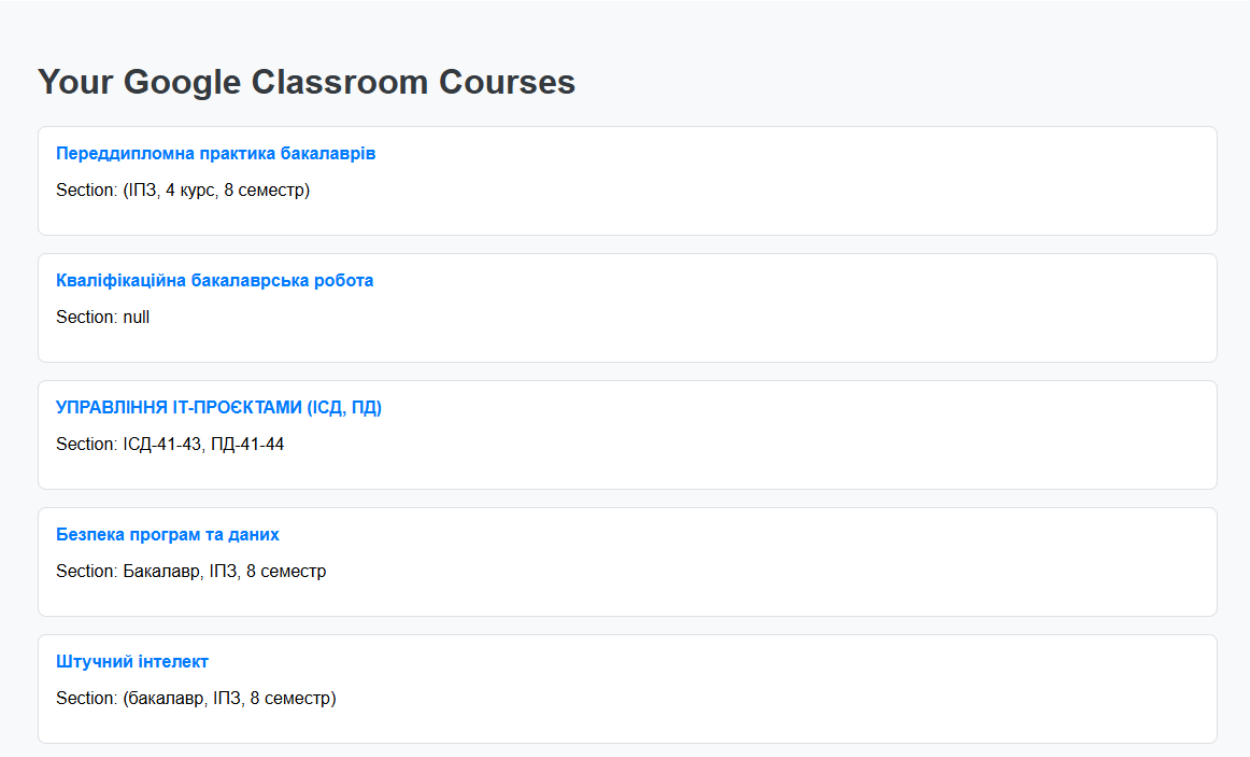


Рис. 4.15 Сторінка з курсами автентифікованого користувача

При переході до конкретного курсу відкривається нова сторінка, де користувач може переглянути завдання (Coursework) та навчальні матеріали (Course Materials), пов’язані з цим курсом. На сторінці, наведеної на рисунку 4.16, відображаються назви лабораторних робіт із зазначенням терміну здачі, а також матеріали занять. Кожен елемент має кнопку, що веде безпосередньо до відповідного ресурсу в Google Classroom.

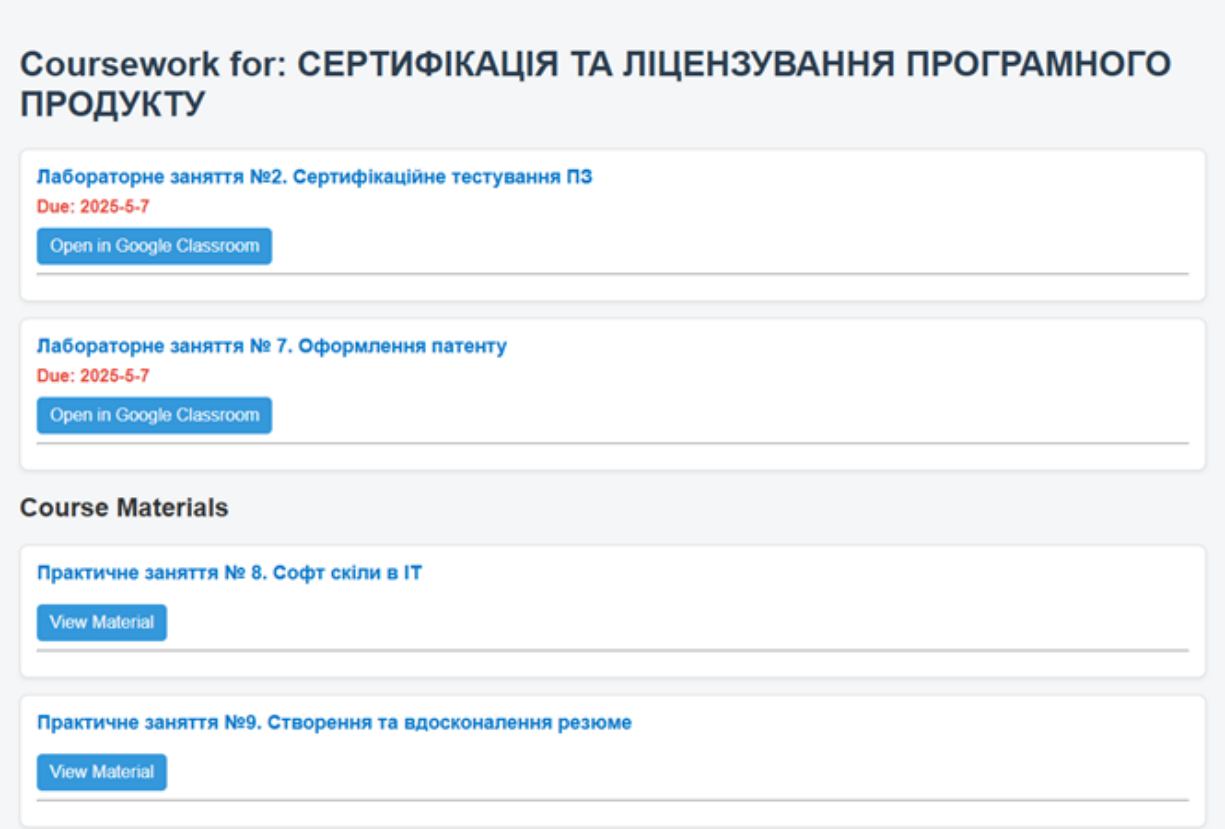


Рис. 4.16 Сторінка з навчальними матеріалами курсу

Таким чином, реалізована інтеграція з Google Classroom API забезпечує безпечний обмін навчальною інформацією між Google-середовищем і розробленим застосунком, покращуючи користувацький досвід без порушення принципів приватності.

4.7 Система сповіщень

У рамках розробки було реалізовано систему сповіщень, яка інформує користувача про важливі події в межах застосунку, зокрема — про наближення дедлайнів завдань, що закріплені за ним. Щоб забезпечити регулярну перевірку умов для створення сповіщень, у застосунку використано механізм планувальника задач (CRON jobs) — функціональність, яка дозволяє автоматично виконувати певні дії через задані інтервали часу.

Для реалізації цієї логіки було створено окремий конфігураційний @EnableScheduling:

```
@Configuration
@EnableScheduling
public class NotificationConfig {
}
```

Анотація @EnableScheduling активує підтримку механізму планування у Spring — тобто дозволяє позначати методи як періодичні за допомогою анотацій типу @Scheduled(cron = "..."). Це дало змогу автоматично запускати метод, який перевіряє дедлайни завдань і створює відповідні сповіщення для користувачів, якщо завдання вже прострочене або наближається його термін виконання.

Ключовим елементом реалізації є модель сповіщення:

```
@Entity
@Data
public class Notification {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String message;
   @ManyToOne
   private User user;
   private Long projectId;
}
```

У цій сутності передбачено посилання на користувача (модель User), якому належить сповіщення, та ідентифікатор проєкту, до якого воно стосується. Наявність зв’язку з користувачем дає змогу персоналізувати сповіщення — кожен користувач бачить лише повідомлення, що стосуються його завдань. Ідентифікатор проєкту, у свою чергу, дозволяє зробити сповіщення інтерактивним — тобто забезпечити можливість переходу від повідомлення безпосередньо до сторінки відповідного проєкту.

Для відображення сповіщень у вебінтерфейсі було реалізовано навігаційну панель, яка присутня на більшості сторінок застосунку. Вона містить кнопку переходу на головну сторінку, кнопку доступу до профілю користувача, а також значок сповіщень, при натисканні на який відкривається випадаючий блок зі списком повідомлень.

Основна частина HTML-коду, що відповідає за відображення повідомлень, наведена нижче:

```
<div id="notifDropdown">
  <ul th:if="${notifications != null and !notifications.isEmpty()}">
    <li th:each="notif : ${notifications}">
      <a th:href="@{'/api/project/' + ${notif.projectId}}"
        th:text="${notif.message}"></a>
    </li>
  </ul>
  <span th:if="${notifications == null or notifications.isEmpty()}">
    No notifications
  </span>
</div>
```

Цей фрагмент відповідає за динамічне формування списку сповіщень, що надходять до шаблону через Model. Якщо список не порожній — кожне повідомлення відображається як посилання на відповідний проєкт як зображено на рисунку 4.17. Якщо ж сповіщень немає, користувачу показується відповідне повідомлення про їх відсутність.

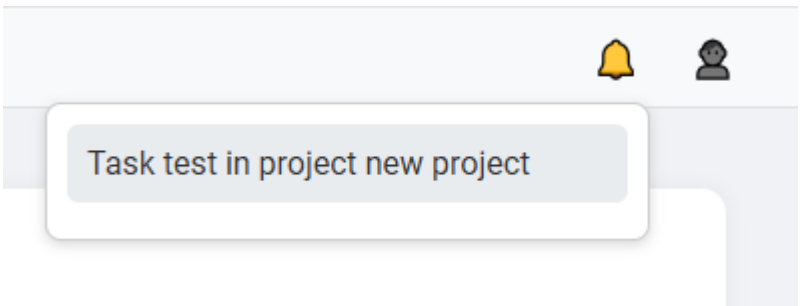


Рис. 4.17 Випадаючий список з повідомленням про дедлайн завдання

Самі методи, що відповідають за створення та очищення сповіщень, розміщено в класі NotificationService. Зокрема, реалізовано два ключові методи:

```
@Scheduled(cron = "0 00 00 * * *")
@Transactional
public void generateDailyNotifications() {…}
```

Цей метод виконується щодня опівночі (00:00) та перевіряє список активних завдань для кожного користувача. Якщо завдання мають дедлайн на наступний день та мають в собі назначеного користувача, для відповідного користувача створюється нове повідомлення з коротким текстовим описом та посиланням на проєкт, якого воно стосується.

Для очищення застарілих сповіщень використовується інший метод:

```
@Scheduled(cron = "0 59 23 * * *")
@Transactional
public void clearNotifications() {
   notificationRepository.deleteAll();
}
```

Цей метод викликається щоденно о 23:59 і очищує таблицю сповіщень, щоб уникнути накопичення неактуальної інформації. Завдяки цій системі, повідомлення мають короткотривалу дію та оновлюються щодня.

Таким чином, реалізована система сповіщень не лише підвищує інформованість користувача, а й забезпечує прямий та зручний доступ до важливої інформації в межах інтерфейсу застосунку.

4.8 Тестування

Для перевірки коректної роботи системи було застосовано інтеграційне тестування, яке дозволяє перевірити взаємодію між різними модулями застосунку — наприклад, між контролерами, сервісами, репозиторіями та зовнішніми ресурсами. Інтеграційні тести охоплюють більшу частину архітектури, ніж

модульні тести, і дають змогу виявити помилки, пов’язані з конфігурацією середовища або некоректною передачею даних між компонентами.

Особливу увагу в тестовому середовищі було приділено роботі з Redis, який використовується для зберігання JWT-токенів. Оскільки під час інтеграційного тестування важливо уникати залежності від зовнішніх сервісів, було використано embedded Redis — вбудовану тестову інстанцію Redis-сервера, яка піднімається локально в процесі запуску тестів і автоматично завершується після їх завершення. Це дає змогу проводити тести ізольовано, без необхідності запускати Redis-сервер окремо.

Для налаштування embedded Redis було створено три допоміжні класи в пакеті test, кожен із яких виконує свою функцію:

- TestRedisConfiguration — відповідає за запуск та зупинку вбудованого Redis-сервера. Він ініціалізує Redis на порту, вказаному в тестових конфігураційних файлах, за допомогою анотацій @PostConstruct та @PreDestroy, що гарантує його запуск перед тестами та завершення після них;

- RedisConfiguration — відповідає за створення з’єднання з Redis у тестовому середовищі. У цьому класі визначено бин LettuceConnectionFactory, що забезпечує підключення до Redis, а також RedisTemplate, який використовується для читання та запису об'єктів у Redis у рамках тестових сценаріїв;

- RedisProperties — допоміжний конфігураційний клас, у якому зберігаються параметри з’єднання з Redis: порт і адреса хоста. Ці параметри зчитуються з файлу application-test.properties, що знаходиться у директорії test/resources, і забезпечують централізоване керування тестовими налаштуваннями.

У тестовому середовищі для зберігання даних використовувалась вбудована база даних H2 — це швидка SQL-база даних, яка особливо зручна для написання інтеграційних тестів, оскільки може запускатися в пам’яті та не потребує зовнішнього підключення, і автоматично очищується після завершення тестів.

H2 підтримує кілька режимів сумісності з іншими СУБД. H2 було налаштовано у режим MySQL, що дозволяє H2-інтерпретатору поводитися максимально подібно

до основної СУБД застосунку. Це дозволяє уникнути помилок сумісності SQL-операцій та легко протестувати застосунок.

Налаштування підключення до H2-бази даних та її режим роботи здійснюється у файлі application-test.properties:

spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=1;DATABASE_TO_UPPER=false;MODE=MySQL

Для ініціалізації структури бази даних було використано файл schema.sql, який розташовано у папці test/resources. У цьому файлі зберігаються SQL-інструкції (DDL), які описують структуру таблиць, що потрібні для тестування. Усі сутності описані у цьому файлі створюються під час запуску інтеграційних тестів.

Приклад коду створення сутності для бази даних H2:

```
drop table if exists `project` cascade;
CREATE TABLE `project` (
    `id` bigint NOT NULL AUTO_INCREMENT,
    `description` varchar(255) DEFAULT NULL,
    `title` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Для забезпечення повторюваності тестів і уникнення конфліктів даних між запусками різних тестів, було також додано файл clear_h2.sql, який очищує дані всіх таблиць перед кожним запуском наступного тесту:

```
SET REFERENTIAL_INTEGRITY FALSE;
TRUNCATE TABLE `user_project`;
TRUNCATE TABLE `project`;
TRUNCATE TABLE `task`;
SET REFERENTIAL_INTEGRITY TRUE;
```

Таким чином, H2 у режимі MySQL з використанням schema.sql та clear_h2.sql дозволяє швидко створювати ізольоване середовище для тестів, не змінюючи реальну базу даних.

Для перевірки контролерів, які відповідають за роботу з проєктами, було реалізовано інтеграційні тести з повним запуском контексту Spring. На рисунку 4.18 зображено оголошення класу ProjectControllerTest, у якому конфігурація тестового середовища забезпечується через набір спеціалізованих анотацій:

- @SpringBootTest — піднімає повноцінний Spring-контекст для тестів;

- @AutoConfigureMockMvc — дозволяє використовувати MockMvc для тестування HTTP-запитів до контролерів без запуску сервера;

- @ActiveProfiles("test") — активує профіль test, що використовує налаштування з файлу application-test.properties;

- @Sql — виконує скрипт clear_h2.sql перед кожним тестом для очищення БД;

- @Import(TestRedisConfiguration.class) — імпортує конфігурацію embedded Redis;

- @TestPropertySource — підключає файл .env для зчитування змінних оточення.

```
@Slf4j  YegorPopinako *
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
@ActiveProfiles("test")
@Sql(scripts = "/clear_h2.sql", executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD)
@Import(TestRedisConfiguration.class)
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
@TestPropertySource(locations = "file:./.env")
class ProjectControllerTest {
```

Рис. 4.18 Приклад тестового класу

Для тестування потрібна взаємодія з різними шарами застосунку, тому залежності до необхідних компонентів (репозиторіїв, сервісів тощо) інжектуються за допомогою анотації @Autowired. У наступному фрагменті коду продемонстровано приклад автозв’язування репозиторію UserRepository, що дозволяє безпосередньо використовувати його логіку у тестах:

@Autowired

private UserRepository userRepository;

Перед виконанням кожного тесту відбувається попереднє налаштування початкового стану — створення користувача, проєкту, зв’язку між ними та генерація токена доступу. Як зображено на рисунку 4.19, для цього використовується метод setUp, позначений анотацією @BeforeEach, що забезпечує генерацію нових даних перед кожним новим тестовим методом. Цей метод забезпечує створення тестових даних, їх збереження у базі даних та формування cookie з JWT-токеном, що імітує авторизованого користувача під час виконання запитів у тестах.

```
@BeforeEach  YegorPopinako *
void setUp() {
    User user = new User();
    user.setEmail("testuser@example.com");
    savedUser = userRepository.save(user);

    Project project = new Project();
    project.setTitle("Test Project");
    project.setDescription("Test Description");
    savedProject = projectRepository.save(project);

    UserProject assignment = new UserProject();
    assignment.setId(new UserProjectId(savedUser.getId(), savedProject.getId()));
    assignment.setUser(savedUser);
    assignment.setProject(savedProject);
    assignment.setRole(Role.ADMIN);
    userProjectRepository.save(assignment);

    String token = tokenService.generateToken(savedUser);

    cookie = new Cookie( name: "token", token);
    cookie.setHttpOnly(true);
    cookie.setSecure(false);
    cookie.setPath("/");
    cookie.setMaxAge(300);
}
```

Рис. 4.19 Метод створення тестових даних

На рисунку 4.20 наведено приклад самого інтеграційного тесту. У ньому перевіряється сценарій успішного отримання сторінки проєкту користувачем, який

є учасником цього проєкту. Тест відправляє GET-запит на адресу /api/project/{id} з відповідним токеном у cookie, і очікує:

- статус відповіді 200 OK;
- рендеринг HTML-шаблону з іменем "project";
- наявність у моделі атрибуту projectDto;
- відповідність назви проєкту в DTO очікуваному значенню ("Test Project").

Такий підхід дозволяє перевірити повний ланцюг обробки запиту — від автентифікації через токен і перевірки ролей до обробки запиту контролером і передачі даних у шаблон. Тестування через MockMvc гарантує, що всі рівні застосунку працюють узгоджено, а інтерфейс користувача отримує необхідну інформацію в очікуваному вигляді.

```
@Test  YegorPopinako *
@SneakyThrows
void testGetProject_success_whenUserIsAssigned() {
    mockMvc.perform(get( uriTemplate: "/api/project/{id}", savedProject.getId())
                    .cookie(cookie))
            .andExpect(status().isOk())
            .andExpect(view().name( expectedViewName: "project"))
            .andExpect(model().attributeExists( ...names: "projectDto"))
            .andExpect(model().attribute( name: "projectDto", hasProperty( propertyName: "title", is( value: "Test Project"))));
}
```

Рис. 4.20 Приклад тестового методу

Інтеграційні тести охоплюють реальні сценарії взаємодії користувача з системою, включаючи автентифікацію, перевірку прав доступу, обробку запитів і передачу даних у шаблони представлення. Завдяки використанню MockMvc, Spring Test, планувальника задач, SQL-ініціалізації та токенів у cookie, тести відображають поведінку застосунку в умовах, максимально наближених до продуктивного середовища.

Таким чином, виконане тестування підтвердило стабільність, надійність і коректність реалізованої бізнес-логіки, забезпечуючи основу для подальшого розширення функціональності, підтримки та масштабування застосунку.

ВИСНОВКИ

У результаті виконання кваліфікаційної бакалаврської роботи було створено web-застосунок для менеджменту командних студентських проєктів, що забезпечує ефективну взаємодію між учасниками навчального процесу, контроль дедлайнів, доступ до навчальних курсів та інструменти для управління завданнями в межах команд. Застосунок дозволяє студентам зручно координувати спільну діяльність, а також інтегрує функціональність із Google Classroom для зручного доступу та відображення актуальних навчальних даних.

У ході виконання роботи було досягнуто наступних результатів:

1. Проведено аналіз наявних систем для організації командної роботи. Виявлено відсутність інтеграції з Google Classroom у більшості аналогів та обмеження щодо сповіщень і кастомізації під навчальний процес. Сформовано функціональні та нефункціональні вимоги до майбутнього застосунку з урахуванням потреб студентських команд.

2. Проведено аналіз засобів реалізації застосунку: Java, Spring Boot, Spring Security, Thymeleaf, MySQL, Redis, Docker, Google Classroom API. Визначено їх переваги для створення масштабованого, безпечного та адаптивного веб-застосунку.

3. Спроєктовано архітектуру системи, реалізовано логічний поділ на шари (конфігурації, контролери, сервіси, DTO, моделі, безпека), використано шаблон MVC, реалізовано систему автентифікації та авторизації з використанням OAuth 2.0 та JWT.

4. Реалізовано функціональність застосунку, включаючи авторизацію, створення, редагування і видалення проєктів та завдань, формування сповіщень за допомогою CRON-задач, а також HTML-інтерфейс через шаблони Thymeleaf. Додатково реалізовано інтеграцію з Google Classroom для відображення курсів, завдань і матеріалів у межах веб-застосунку.

5. Проведено тестування функціональних можливостей. Реалізовано інтеграційні тести із застосуванням H2-бази та вкладеним Redis. Створено окремі конфігурації для тестового середовища, забезпечено автоматичне очищення бази даних перед запуском кожного тесту. Виконано перевірку взаємодії основних компонентів і сценаріїв поведінки системи.

Таким чином, усі поставлені завдання кваліфікаційної роботи були виконані. Застосунок функціонує відповідно до визначених вимог і є придатним для практичного використання в умовах навчального закладу. Його структура забезпечує можливість подальшого масштабування, додавання нових функцій і налаштування під потреби конкретної організації.

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

1. Попінако Є.О., Замрій І.В. Архітектура web-застосунку для менеджменту командних студентських проєктів. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, С. 59-60.

2. Попінако Є.О., Замрій І.В. Безпека у web-застосунку для менеджменту командних студентських проєктів. // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.2025, КСУ імені Бориса Грінченка, Київ, Україна, С. 316-317.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Capture, organize, and tackle your to-dos from anywhere | Trello. *Capture, organize, and tackle your to-dos from anywhere | Trello*. URL: https://trello.com/
2. Manage your team's work, projects, & tasks online • Asana. *Asana*. URL: https://asana.com/
3. Jira | issue & project tracking software | atlassian. *Collaboration software for software, IT and business teams*. URL: https://www.atlassian.com/software/jira
4. Microsoft planner. *Microsoft Planner*. URL: https://planner.cloud.microsoft/
5. ClickUp. *ClickUp | The everything app for work*. URL: https://clickup.com/
6. What is an IDE? - integrated development environment explained - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/what-is/ide/
7. Simplilearn. What is intellij IDEA - A comprehensive guide. *Simplilearn.com*. URL: https://www.simplilearn.com/what-is-intellij-idea-article
8. Introduction to Java - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/introduction-to-java/
9. Spilca L. Spring Start Here: Learn What You Need and Learn It Well. Manning Publications Co. LLC, 2021.
10. Awati R. What is spring framework? | definition from techtarget. *Search App Architecture*.
    URL: https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework
11. Spring boot : up and running: building cloud native java and kotlin applications. O'Reilly Media, Incorporated, 2021. 325 p.
12. MySQL Crash Course. Pearson Education, Limited, 2023.
13. Eddelbuettel D. A brief introduction to redis. Іллінойс : arXiv, 2022. 3 с. (Препринт. University of Illinoi ; arXiv:2203.06559). URL: https://doi.org/10.48550/arXiv.2203.06559.
14. A Comprehensive Study On Hibernate As A Data Persistence Solution For Financial Applications. *International Journal of Applied Engineering & Technology*. 2024.

Vol. 6, no. 4. P. 124. URL: https://www.researchgate.net/profile/Anil-Kumar-Bayya/publication/387963380_A_Comprehensive_Study_On_Hibernate_As_A_Data_Persistence_Solution_For_Financial_Applications/links/67853bf155274940f1239b80/A-Comprehensive-Study-On-Hibernate-As-A-Data-Persistence-Solution-For-Financial-Applications.pdf.

15. RFC 8725. JSON web token best current practices. Replaces RFC 7519 ; effective from 2020-05-01. Official edition. 2020. URL: https://datatracker.ietf.org/doc/html/rfc7519.
16. RFC 6749. The oauth 2.0 authorization framework. Effective from 2012-10-01. Official edition. URL: https://datatracker.ietf.org/doc/html/rfc6749.
17. What is docker?. *Docker Documentation*. URL: https://docs.docker.com/get-started/docker-overview/
18. Thymeleaf. *Thymeleaf*. URL: https://www.thymeleaf.org/.
19. Jefster. OAuth2 in spring security: understanding the client, authorization server, and resource server. Medium. URL: https://medium.com/@dev.jefster/oauth2-in-spring-security-understanding-the-client-authorization-server-and-resource-server-e90c14630b20.
20. Oauth що це: визначення технології та основні принципи роботи. FoxmindEd. URL: https://foxminded.ua/oauth-shcho-tse/.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-застосунку для менеджменту командних студентських проєктів

Виконав студент 4 курсу
Групи ПД-41
Попінако Єгор Олексійович
Керівник роботи
д.т.н, проф., доцент кафедри ІПЗ Замрій Ірина Вікторівна
Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** – покращення процесів організації, планування та контролю командної роботи студентів шляхом створення функціонального web-застосунку.
- **Об'єкт дослідження** – процес організації та менеджменту командних проєктів.
- **Предмет дослідження** – web-застосунок для управління командною роботою студентів у рамках навчальних проєктів.

2

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести огляд застосунків для організації командної роботи та визначити вимоги до програмного забезпечення.
2. Провести огляд та аналіз засобів реалізації застосунку для менеджменту командних студентських проєктів.
3. Спроєктувати архітектуру до визначеного ПЗ.
4. Програмно реалізувати застосунок для менеджменту командних студентських проєктів.
5. Провести тестування застосунку для менеджменту командних студентських проєктів.

3

АНАЛІЗ АНАЛОГІВ

	Trello	Asana	Jira	Microsoft Planner	ClickUp	Project manager
Інтеграція з Google Classroom	-	-	-	-	-	+
Ролі та управління доступом	+	+	+	+	+	+
Запрошення користувачів через email	+	+	+	+	+	+
Внутрішні повідомлення про кінцевий термін завдань	-	+	-	-	+	+
Профіль користувача	+	+	+	+	+	+
Управління завданнями	+	+	+	+	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

1.Реєстрація та авторизація користувачів.

2.Можливість створення, редагування та видалення проєктів.

3.Можливість управління завданнями в проєкті.

4.Можливість відстеження статусу завдань "не розпочато", "у процесі", "виконано".

5.Розмежування прав доступу(користувач, адміністратор).

6.Реалізація внутрішніх повідомлень про кінцевий термін завдань.

7.Можливість користувачам бачити свої курси, а також завдання та навчальні матеріали всередині системи з можливістю відкриття цього завдання у Classroom.

Нефункціональні вимоги:

1. Використання протоколу OAuth 2.0 для входу через Google-акаунт
2. Можливість розширення функціоналу.
3. Відгук системи не більше ніж 2-3 секунди.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

IntelliJ Idea

Spring Framework

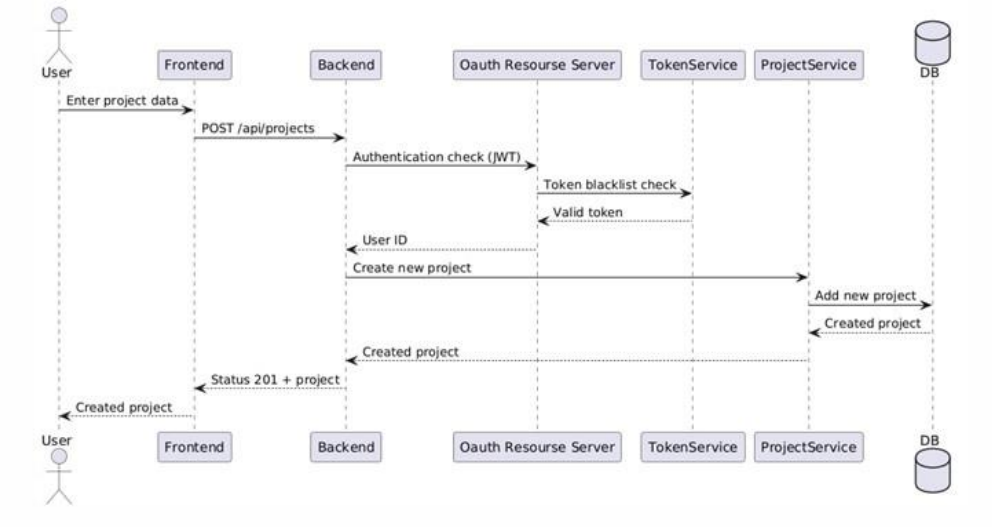
Docker

Thymeleaf

Google Classroom API

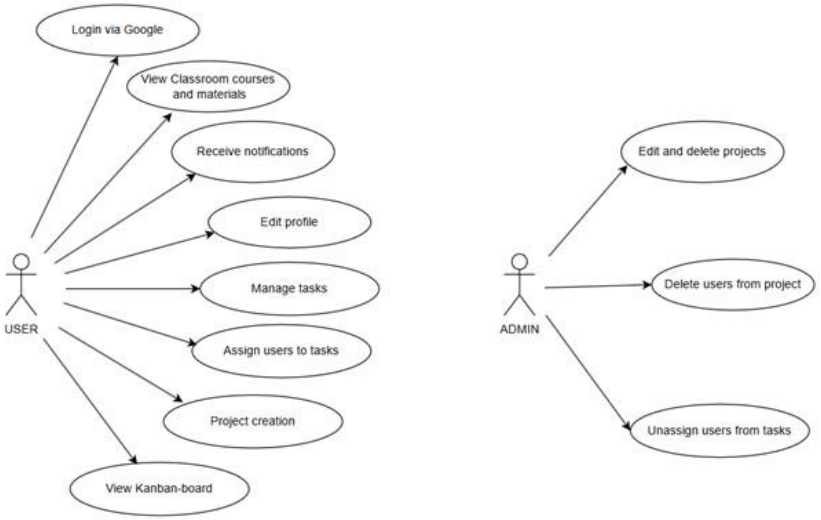
6

Діаграма послідовності створення користувачем нового проєкту



7

Діаграма варіантів використання

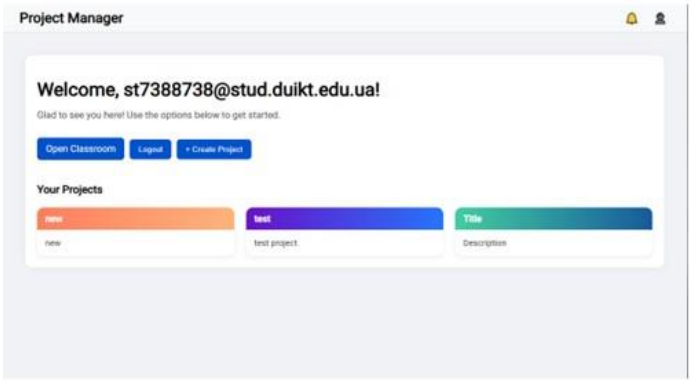


8

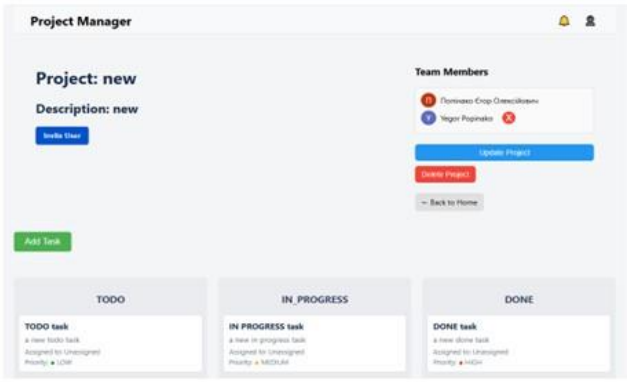
Діаграма класів безпеки

9

ЕКРАННІ ФОРМИ

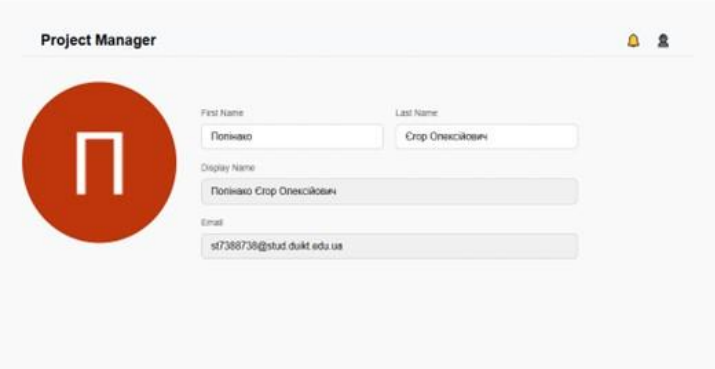


Головна сторінка застосунку

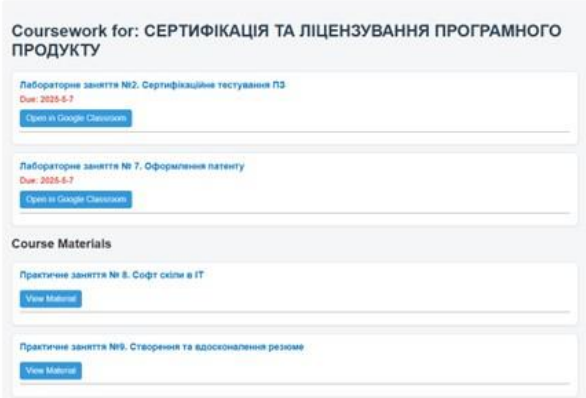


Головна сторінка проекту

ЕКРАННІ ФОРМИ



Сторінка з профілем користувача



Сторінка з завданнями курсу

11

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Попінако Є.О., Замрій І.В. АРХІТЕКТУРА WEB-ЗАСТОСУНКУ ДЛЯ МЕНЕДЖМЕНТУ КОМАНДНИХ СТУДЕНТСЬКИХ ПРОЄКТІВ. // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, ДУІКТ, Київ, Україна, С. 58-59.

2. Попінако Є.О., Замрій І.В. БЕЗПЕКА У WEB-ЗАСТОСУНКУ ДЛЯ МЕНЕДЖМЕНТУ КОМАНДНИХ СТУДЕНТСЬКИХ ПРОЄКТІВ. // Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025». 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 316-317.

12

ВИСНОВКИ

1. Проаналізовано існуючі системи командної роботи та сформовано вимоги до нового застосунку з урахуванням специфіки навчального процесу, що дозволило закласти основу для цільової функціональності.

2. Проведено аналіз технологій та обґрунтовано вибір Java, Spring Boot, Thymeleaf, Redis, Docker та інших інструментів, що дозволило забезпечити масштабованість, безпеку й адаптивність розробки.

3. Спроєктовано архітектуру з поділом на логічні шари та реалізовано авторизацію через OAuth 2.0 та JWT, що дозволило структурувати код і забезпечити контроль доступу.

4. Реалізовано ключову функціональність системи, включаючи управління проєктами й завданнями, сповіщення, інтеграцію з Google Classroom, що дозволило створити повноцінний веб-застосунок для підтримки навчальної діяльності.

5. Проведено інтеграційне тестування з H2-базою даних, налаштовано тестове середовище, що дозволило перевірити коректність взаємодії модулів та підвищити надійність системи.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
@Controller
@RequestMapping("/api/project")
@RequiredArgsConstructor
public class ProjectController {

private final ProjectService projectService;
private final UserService userService;
private final UserProjectRepository
userProjectRepository;
private final ModelMapper modelMapper;
private final NotificationService
notificationService;

@Operation(summary = "Get project by ID")
@ApiResponse(responseCode = "200", description
= "Project found")
@ApiResponse(responseCode = "404", description
= "Project not found")
@GetMapping("/{id}")
@PreAuthorize("@userService.isUserAssignedToP
roject(#id, authentication.name)")
public String getProject(@PathVariable Long id,
Model model, Authentication authentication) {
      ProjectFullInfoDto fullInfoDto =
projectService.getProject(id);
      boolean isOwner =
userService.isUserAdminOfProject(id,
authentication.getName());
      List<UserFullInfoDto> users =
userProjectRepository.getUsersByProjectId(id)
            .stream()
            .map(user -> modelMapper.map(user,
UserFullInfoDto.class))
            .toList();
      ProjectUpdateDto updateDto = new
ProjectUpdateDto();
      updateDto.setId(id);
      List<Notification> notifications =
notificationService.getUserNotifications(authenticat
ion.getName());
      model.addAttribute("notifications",
notifications);
      model.addAttribute("projectDto", fullInfoDto);
      model.addAttribute("isOwner", isOwner);
      model.addAttribute("projectUpdateDto",
updateDto);
      model.addAttribute("users", users);
      model.addAttribute("assignUserDto", new
AssignUserDto());
      model.addAttribute("authentication",
authentication);
      return "project";
   }

@PostMapping()
public String createProject(@ModelAttribute
ProjectDto projectDto, Authentication
authentication, RedirectAttributes
redirectAttributes) {
projectService.createProject(projectDto,
authentication.getName());
redirectAttributes.addFlashAttribute("message",
"Project created successfully!");
      return "redirect:/home";
   }


@Operation(summary = "Update existing project")
@ApiResponse(responseCode = "200", description
= "Project updated")
@ApiResponse(responseCode = "400", description
= "Invalid project data")
@ApiResponse(responseCode = "404", description
= "Project not found")
@PutMapping
@PreAuthorize("@userService.isUserAdminOfProj
ect(#projectUpdateDto.id, authentication.name)")
public String updateProject(@ModelAttribute
ProjectUpdateDto projectUpdateDto) {
projectService.updateProject(projectUpdateDto);
return "redirect:/api/project/" +
projectUpdateDto.getId();
   }


@Operation(summary = "Delete project by ID")
@ApiResponse(responseCode = "200", description
= "Project deleted")
@ApiResponse(responseCode = "404", description
= "Project not found")
@DeleteMapping("/{id}")
@PreAuthorize("@userService.isUserAdminOfProj
ect(#id, authentication.name)")
public String deleteProject(@PathVariable Long id)
{
      projectService.deleteProject(id);
      return "redirect:/home";
 }
```

```
@Operation(summary = "Unassign user from
project")
@ApiResponse(responseCode = "200", description
= "User unassigned from project")
@ApiResponse(responseCode = "404", description
= "User or project not found")
@PostMapping("/unassign")
@PreAuthorize("@userService.isUserAdminOfProj
ect(#assignUserDto.id, authentication.name)")
    public String
unassignUserFromProject(@ModelAttribute
AssignUserDto assignUserDto) {
        userService.unassignUser(assignUserDto);
        return "redirect:/api/project/" +
assignUserDto.getId();
    }
}

@Controller
@RequestMapping("/api/task")
@RequiredArgsConstructor
public class TaskController {

private final TaskService taskService;
private final UserService userService;
private final TaskRepository taskRepository;

@Operation(summary = "Get task by ID")
@ApiResponse(responseCode = "200", description
= "Task found")
@ApiResponse(responseCode = "404", description
= "Task not found")
@GetMapping("/{id}")
@PreAuthorize("@userService.isUserAssignedToP
rojectByTask(#id, authentication.name)")
public TaskFullInfoDto getTask(@PathVariable
Long id) {
        return taskService.getTask(id);
    }

@Operation(summary = "Create new task")
@ApiResponse(responseCode = "201", description
= "Task created")
@ApiResponse(responseCode = "400", description
= "Invalid task data")
@PostMapping
@PreAuthorize("@userService.isUserAssignedToP
roject(#taskDto.projectId, authentication.name)")
public String createTask(@Valid @ModelAttribute
TaskDto taskDto) {
        taskService.createTask(taskDto);
        return "redirect:/api/project/" +
taskDto.getProjectId();
    }

@Operation(summary = "Update existing task")
@ApiResponse(responseCode = "200", description
= "Task updated")
@ApiResponse(responseCode = "400", description
= "Invalid task data")
@ApiResponse(responseCode = "404", description
= "Task not found")
@PutMapping()
@PreAuthorize("@userService.isUserAssignedToP
rojectByTask(#taskUpdateDto.id,
authentication.name)")
    public String updateTask(@ModelAttribute
@Valid TaskUpdateDto taskUpdateDto) {
        taskService.updateTask(taskUpdateDto);
        return "redirect:/api/project/" +
taskUpdateDto.getProjectId();
    }

@Operation(summary = "Delete task by ID")
@ApiResponse(responseCode = "200", description
= "Task deleted")
@ApiResponse(responseCode = "404", description
= "Task not found")
@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.OK)

@PreAuthorize("@userService.isUserAssignedToP
rojectByTask(#id, authentication.name)")
    public void deleteTask(@PathVariable Long id) {
        taskService.deleteTask(id);
    }

@Operation(summary = "Assign user to task")
@ApiResponse(responseCode = "200", description
= "User assigned to task")
@ApiResponse(responseCode = "404", description
= "Task not found")
@PostMapping("/assign")
@PreAuthorize("@userService.isUserAssignedToP
rojectByTask(#assignUserDto.id,
authentication.name)")
public String assignUserToTask(@ModelAttribute
AssignUserDto assignUserDto) {
        Task task =
taskRepository.findById(assignUserDto.getId()).or
ElseThrow(() -> new
EntityNotFoundException("Task not found"));
        userService.assignUser(assignUserDto);
        return "redirect:/api/project/" +
task.getProject().getId();
    }

@Operation(summary = "Unassign user from task")
@ApiResponse(responseCode = "200", description
= "User unassigned from task")
@ApiResponse(responseCode = "404", description
= "Task not found")
@PostMapping("/unassign")
@PreAuthorize("@userService.isUserAssignedToP
```

```
rojectByTask(#assignUserDto.id,
authentication.name)")
public String
unassignUserFromTask(@ModelAttribute
AssignUserDto assignUserDto) {
        Task task =
taskRepository.findById(assignUserDto.getId()).or
ElseThrow(() -> new
EntityNotFoundException("Task not found"));
        userService.unassignUser(assignUserDto);
        return "redirect:/api/project/" +
task.getProject().getId();
    }


@PostMapping("/status")
@PreAuthorize("@userService.isUserAssignedToP
rojectByTask(#updateTaskStatusDto.taskId,
authentication.name)")
public ResponseEntity<Void>
updateTaskStatus(@RequestBody
UpdateTaskStatusDto updateTaskStatusDto) {
        Task task =
taskRepository.findById(updateTaskStatusDto.getT
askId()).orElseThrow(() -> new
EntityNotFoundException("Task not found"));

task.setStatus(TaskStatus.valueOf(updateTaskStatus
Dto.getNewStatus()));
        taskRepository.save(task);
        return ResponseEntity.ok().build();
    }
}
```

```
@Controller
@RequiredArgsConstructor
@RequestMapping("/api/user")
public class UserController {

private final UserService userService;
private final NotificationService
notificationService;

@GetMapping
public String getUser(Authentication
authentication, Model model) {
        List<Notification> notifications =
notificationService.getUserNotifications(authenticat
ion.getName());
```

```
@Service
@RequiredArgsConstructor
public class ProjectService {
private final ProjectRepository projectRepository;
private final ModelMapper modelMapper;
private final ProjectMapper projectMapper;
private final UserRepository userRepository;
private final UserProjectRepository
```

```
        UserFullInfoDto userDto =
userService.getUser(authentication.getName());
        model.addAttribute("notifications",
notifications);
        model.addAttribute("userDto", userDto);
        return "userProfile";
    }

@PutMapping
public String updateUser(@ModelAttribute
UserUpdateDto userDto, Model model,
Authentication authentication) {
        userService.updateUser(userDto);
        UserFullInfoDto updatedUser =
userService.getUser(authentication.getName());
        List<Notification> notifications =
notificationService.getUserNotifications(authenticat
ion.getName());
        model.addAttribute("notifications",
notifications);
        model.addAttribute("userDto", updatedUser);
        return "userProfile";
    }

@GetMapping("/{id}/image")
public ResponseEntity<byte[]>
getUserImage(@PathVariable Long id) {
        try {
            byte[] imageData =
userService.getUserImage(id);

            if (imageData == null || imageData.length
== 0) {
                return ResponseEntity.notFound().build();
            }

            HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.IMAGE_JPEG
);
            return new ResponseEntity<>(imageData,
headers, HttpStatus.OK);

        } catch (EntityNotFoundException e) {
            return ResponseEntity.notFound().build();
        }
    }
}
```

```
userProjectRepository;
private final TaskRepository taskRepository;

public ProjectFullInfoDto getProject(Long id) {
Project project = projectRepository.findById(id).
orElseThrow(() -> new
EntityNotFoundException("Project with id " + id +
" not found"));
```

```
        return modelMapper.map(project,
ProjectFullInfoDto.class);
    }

@Transactional
public ProjectFullInfoDto createProject(ProjectDto
projectDto, String email) {
        Project project =
modelMapper.map(projectDto, Project.class);

        User creator =
userRepository.findByEmail(email)
                .orElseThrow(() -> new
UsernameNotFoundException("User not found"));

        project = projectRepository.save(project);

        UserProject userProject = new UserProject();
        userProject.setUser(creator);
        userProject.setProject(project);
        userProject.setRole(Role.ADMIN);
        userProject.setId(new
UserProjectId(creator.getId(), project.getId()));

        userProjectRepository.save(userProject);

        return modelMapper.map(project,
ProjectFullInfoDto.class);
@Service
@RequiredArgsConstructor
public class TaskService {

    private final TaskRepository taskRepository;
    private final ProjectRepository
projectRepository;
    private final ModelMapper modelMapper;
    private final TaskMapper taskMapper;

    public TaskFullInfoDto getTask(Long id) {
        Task task = taskRepository.findById(id)
                .orElseThrow(() -> new
EntityNotFoundException("Task with id " + id + "
not found"));
        return modelMapper.map(task,
TaskFullInfoDto.class);
    }

    public List<TaskFullInfoDto>
findByEndPoint(LocalDate tomorrow) {
        return
taskRepository.findByEndPoint(tomorrow)
                .stream()
                .filter(task -> task.getUser() != null)
                .map(task -> modelMapper.map(task,
TaskFullInfoDto.class))
                .toList();
    }
    }

    @Transactional
    public ProjectFullInfoDto
updateProject(ProjectUpdateDto projectDto) {
        var project =
projectRepository.findById(projectDto.getId())
                .orElseThrow(() -> new
EntityNotFoundException("User with id: %s not
found".formatted(projectDto.getId())));


projectMapper.mapProjectFromProjectDto(project
Dto, project);

        return
modelMapper.map(projectRepository.save(project),
ProjectFullInfoDto.class);
    }

    @Transactional
    public void deleteProject(Long id) {
        taskRepository.deleteByProjectId(id);
        userProjectRepository.deleteByProjectId(id);
        projectRepository.deleteById(id);
    }
}


    @Transactional
    public TaskFullInfoDto createTask(TaskDto
taskDto) {
        Task task = new Task();
        task.setTitle(taskDto.getTitle());
        task.setDescription(taskDto.getDescription());
        task.setStatus(taskDto.getStatus());
        task.setPriority(taskDto.getPriority());
        task.setCreatedAt(LocalDateTime.now());
        task.setEndPoint(taskDto.getEndPoint());

task.setProject(projectRepository.findById(taskDto.
getProjectId()).orElseThrow(() ->
                new EntityNotFoundException("Project
not found")));
        return
modelMapper.map(taskRepository.save(task),
TaskFullInfoDto.class);
    }


@Transactional
public TaskFullInfoDto updateTask(TaskUpdateDto
taskDto) {
        var task =
taskRepository.findById(taskDto.getId())
                .orElseThrow(() -> new
EntityNotFoundException("Task with id: %s not
```

```
found".formatted(taskDto.getId())));

        taskMapper.mapTaskFromTaskDto(taskDto,
task);
        task.setUpdatedAt(LocalDateTime.now());


public void deleteTask(Long id) {
        getTask(id);
        taskRepository.deleteById(id);
    }
}
@Slf4j
@Service
@RequiredArgsConstructor
public class UserService {

private final ProjectRepository projectRepository;
private final UserRepository userRepository;
private final TaskRepository taskRepository;
private final ModelMapper modelMapper;
private final UserMapper userMapper;
private final UserProjectRepository
userProjectRepository;

public UserFullInfoDto getUser(String username) {
User user = userRepository.findByEmail(username)
            .orElseThrow(() -> new
EntityNotFoundException("User with email " +
username + " not found"));
        return modelMapper.map(user,
UserFullInfoDto.class);
    }

@Transactional
public UserFullInfoDto updateUser(UserUpdateDto
userDto) {var user =
userRepository.findById(userDto.getId())
            .orElseThrow(() -> new
EntityNotFoundException("User with id: %s not
found".formatted(userDto.getId())));
userMapper.mapUserFromUserDto(userDto, user);
user.setDisplayName(user.getFirstName() + " " +
user.getLastName());
return
modelMapper.map(userRepository.save(user),
UserFullInfoDto.class);
    }

@Transactional
public void assignUser(AssignUserDto dto) {
User user =
userRepository.findByEmail(dto.getUserEmail())
            .orElseThrow(() -> new
EntityNotFoundException("User is not registered in
a system")); Object entity =
        return
modelMapper.map(taskRepository.save(task),
TaskFullInfoDto.class);
    }

@Transactiona

getEntityByType(dto.getType(), dto.getId());
if (entity instanceof Project project) {
    if

(!userProjectRepository.existsByUserIdAndProjectI
d(user.getId(), project.getId())) {
            UserProject userProject = new
UserProject();
            userProject.setUser(user);
            userProject.setProject(project);
            userProject.setRole(Role.USER);
            userProjectRepository.save(userProject);
        }
    } else if (entity instanceof Task task) {
        validateUserProjectAssignment(task.getId(),
user.getEmail());
        task.setUser(user);
        taskRepository.save(task);
    }
  }

@Transactional
public void unassignUser(AssignUserDto dto) {
User user =
userRepository.findByEmail(dto.getUserEmail())
            .orElseThrow(() -> new
EntityNotFoundException("User not found"));
Object entity = getEntityByType(dto.getType(),
dto.getId());
if (entity instanceof Project project) {
userProjectRepository.deleteByUserIdAndProjectId
(user.getId(), project.getId());
taskRepository.unassignUserFromTasksInProject(pr
oject.getId(), user.getId());
    } else if (entity instanceof Task task) {
        validateTaskUnassignment(task, user);
        task.setUser(null);
        taskRepository.save(task);
    }
  }

  @Transactional
  public void registerIfAbsent(String email, String
firstName, String lastName, String imageUrl) {
    if (!userRepository.existsByEmail(email)) {
        User user = new User();
        user.setEmail(email);
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setDisplayName(firstName + " " +
```

```
lastName);

try (InputStream in = new
URL(imageUrl).openStream()) {
            byte[] imageData = in.readAllBytes();
            user.setProfileImage(imageData);
        } catch (IOException e) {
            user.setProfileImage(null);
        }
        userRepository.save(user);
    }
  }

@Transactional
public byte[] getUserImage(Long id){
    User user = userRepository.findById(id)
            .orElseThrow(() -> new
EntityNotFoundException("User not found"));
    return user.getProfileImage();
  }
public List<ProjectFullInfoDto>
getUserProjects(String name) {
User user =
userRepository.findByEmail(name).orElseThrow(()
-> new UsernameNotFoundException("User not
found")); return
userProjectRepository.findProjectsByUserId(user.ge
tId())
            .stream()
            .map(project ->
modelMapper.map(project,
ProjectFullInfoDto.class))
            .toList();
  }

public boolean isUserAdminOfProject(Long
projectId, String email) {
Long userId = userRepository.findByEmail(email)
            .orElseThrow(() -> new
UsernameNotFoundException("User not
found")).getId();
return
userProjectRepository.existsByUserIdAndProjectId
AndRole(userId, projectId, Role.ADMIN);
  }
public boolean isUserAssignedToProject(Long
projectId, String email) {
    Long userId =
userRepository.findByEmail(email)
            .orElseThrow(() -> new
UsernameNotFoundException("User not found"))
            .getId();
return
userProjectRepository.existsByUserIdAndProjectId
(userId, projectId);
  }
public boolean
isUserAssignedToProjectByTask(Long taskId,
String email) {
    Task task = taskRepository.findById(taskId)
            .orElseThrow(() -> new
EntityNotFoundException("Task not found"));
    Long projectId = task.getProject().getId();
    return isUserAssignedToProject(projectId,
email);
  }

private Object getEntityByType(AssignmentType
type, Long id) {
return switch (type) {
        case PROJECT ->
projectRepository.findById(id)
                .orElseThrow(() -> new
EntityNotFoundException("Project not found"));
        case TASK -> taskRepository.findById(id)
                .orElseThrow(() -> new
EntityNotFoundException("Task not found"));
    };
  }

private void validateTaskUnassignment(Task task,
User user) {
    Optional.ofNullable(task.getUser())
            .orElseThrow(() -> new
IllegalStateException("Task is not assigned to any
user"));

    if (!task.getUser().equals(user)) {
        throw new IllegalStateException("Task is
assigned to a different user");
    }
  }

private void validateUserProjectAssignment(Long
id, String email) {
    if (!isUserAssignedToProjectByTask(id,
email)) {
        throw new IllegalStateException("User is
not assigned to project related to this task");
    }
  }
}

@Repository
public interface ProjectRepository extends
CrudRepository<Project, Long> {

@Override
Optional<Project> findById(Long id);
}

@Repository
public interface TaskRepository extends
CrudRepository<Task, Long> {
```

```
@Modifying
@Transactional
@Query("DELETE FROM Task t WHERE
t.project.id = :projectId")
   void deleteByProjectId(@Param("projectId")
Long projectId);

   @Modifying
@Query("UPDATE Task t SET t.user = null
WHERE t.project.id = :projectId AND t.user.id =
:userId")
   void
unassignUserFromTasksInProject(@Param("project
Id") Long projectId, @Param("userId") Long
userId);

List<Task> findByEndPoint(LocalDate endpoint);
}
@Repository
public interface UserProjectRepository extends
CrudRepository<UserProject, UserProjectId> {

@Query("SELECT COUNT(up) > 0 FROM
UserProject up " +
         "WHERE up.id.userId = :userId AND
up.id.projectId = :projectId")
   boolean
existsByUserIdAndProjectId(@Param("userId")
Long userId,
@Param("projectId") Long projectId);

@Query("SELECT COUNT(up) > 0 FROM
UserProject up " +
"WHERE up.id.userId = :userId AND
up.id.projectId = :projectId AND up.role = :role")
   boolean
existsByUserIdAndProjectIdAndRole(@Param("us
erId") Long userId, @Param("projectId") Long
projectId, @Param("role") Role role);
@Query("select u.project from UserProject u where
u.id.userId = ?1")
   List<Project> findProjectsByUserId(Long
userId);
void deleteByUserIdAndProjectId(Long userId,
Long projectId);


@Modifying
@Transactional
@Query(value = "DELETE FROM user_project
WHERE project_id = :projectId", nativeQuery =
true)
void deleteByProjectId(@Param("projectId") Long
projectId);
@Query("select u.user from UserProject u where
u.id.projectId = ?1")
   List<User> getUsersByProjectId(Long id);
}

@Repository
public interface UserRepository extends
CrudRepository<User, Long> {
Optional<User> findByEmail(String username);
boolean existsByEmail(String email);

boolean existsByEmailAndIdNot(String email,
Long id);
}

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {
@Bean
public SecurityFilterChain
securityFilterChain(HttpSecurity http,
CustomJwtAuthenticationConverter
customConverter,

CustomOAuth2UserService
customOAuth2UserService,

OAuth2AuthenticationSuccessHandler
oAuth2SuccessHandler,

ClientRegistrationRepository
clientRegistrationRepository,

CookieBearerTokenResolver tokenResolver,
CustomEntryPoint customEntryPoint,

CustomAccessDeniedHandler
accessDeniedHandler) throws Exception {
http.authorizeHttpRequests(requests -> requests
                  .requestMatchers("/", "/doc",
"/v3/api-docs/**", "/swagger-ui/**", "/swagger-
ui.html", "/static/**").permitAll()
                  .requestMatchers("/login",
"/oauth2/**", "/logout.html", "/unauthorized",
"/forbidden").permitAll()
                  .requestMatchers("/api/**",
"/ui/logout", "/home", "/classroom-auth",
"/classroom/**").authenticated()
            )
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolic
y.STATELESS))

            .oauth2ResourceServer(oauth2 -> oauth2

.bearerTokenResolver(tokenResolver)
```

```
                .jwt(jwt ->
jwt.jwtAuthenticationConverter(customConverter::
convert)))
.oauth2Login(oauth2 ->
oauth2.authorizationEndpoint(endpoint ->
endpoint.authorizationRequestResolver(
new
CustomAuthorizationRequestResolver(clientRegistr
ationRepository)))
.userInfoEndpoint(userInfo ->
userInfo.userService(customOAuth2UserService))
.successHandler(oAuth2SuccessHandler)
            )

            .exceptionHandling(e -> e

.authenticationEntryPoint(customEntryPoint)

.accessDeniedHandler(accessDeniedHandler)
            );

        return http.build();
    }

    @Bean
    public KeyPair keyPair() throws
NoSuchAlgorithmException {
        var keyPairGenerator =
KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        return keyPairGenerator.generateKeyPair();
    }

    @Bean
    public JwtDecoder jwtDecoder(KeyPair keyPair)
{
        return
NimbusJwtDecoder.withPublicKey((RSAPublicKey
) keyPair.getPublic()).build();
    }

    @Bean
    public JwtEncoder jwtEncoder(KeyPair
keyPair){
        RSAKey jwk = new
RSAKey.Builder((RSAPublicKey)
keyPair.getPublic()).privateKey(keyPair.getPrivate(
)).build();
        var jwkSet = new ImmutableJWKSet<>(new
JWKSet(jwk));
        return new NimbusJwtEncoder(jwkSet);
    }

    @Bean
    public JwtGrantedAuthoritiesConverter
jwtAuthenticationConverter() {
        JwtGrantedAuthoritiesConverter
grantedAuthoritiesConverter = new
JwtGrantedAuthoritiesConverter();

grantedAuthoritiesConverter.setAuthorityPrefix("");

grantedAuthoritiesConverter.setAuthoritiesClaimNa
me("authorities");
        return grantedAuthoritiesConverter;
    }
}
```