Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук

Бакалаврська кваліфікаційна робота на тему:

ПРОГРАМНИЙ МОДУЛЬ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ ДЛЯ СИСТЕМ АВТОМАТИЗАЦІЇ ВИРОБНИЧИХ ПРОЦЕСІВ

Виконав: студент 4 курсу, групи 1КН-21Б
спеціальності 122 Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Поліщук В. Л.
(прізвище та ініціали)

Керівник: к.т.н., доцент, асистент каф. КН
Богач І. В.
(прізвище та ініціали)
«04» червня 2025 р.

Рецензент: д.т.н., професор каф.АІІТ
Квєтний Р. Н.
(прізвище та ініціали)
«05» червня 2025 р.

Допущено до захисту
Зав. кафедри Яровий А.А.
«06» червня 2025 р.

Вінниця ВНТУ – 2025 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 Інформаційні технології
Спеціальність – 122 Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Яровий А. А.
«05» травня 2025 р

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Поліщуку Володимиру Леонідовичу

1. Тема роботи: Програмний модуль керування дозволами та доступами для систем автоматизації виробничих процесів.
Керівник роботи: Богач Ілона Віталіївна, к.т.н., доцент, асистент каф. КН
Затверджені наказом вищого навчального закладу "20" березня 2025 року № 97
2. Термін подання студентом роботи 30 травня 2025 р.
3. Вихідні дані до роботи: об'єктно-орієнтована мова програмування; програмний модуль керування доступом та дозволами; функціонал для керування доступом до компонентів користувацького інтерфейсу та ендпойнтів; API для базової автентифікації користувача за допомого JWT токена та авторизація за допомогою Spring Security.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): Аналіз предметної галузі керування дозволами та доступами; Проєктування програмного модуля керування дозволами та доступами; Реалізація програмного модуля керування дозволами та доступами; Тестування та розгортання програмного модуля керування дозволами та доступами; Висновки; Список використаних джерел.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Структурна схема взаємодії компонентів системи автоматизації виробничих процесів; Схема логічного поділу ролей на типи; Схема композиції сутностей керування доступом; UML-діаграма взаємодії шарів застосунку на прикладі Permission.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Богач І.В., к.т.н., доц., асистент каф.КН		

7. Дата видачі завдання 05 травня 2025р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Аналіз предметної галузі керування дозволами та доступами	05.05.25	08.05.25	викон.
2	Проєктування програмного модуля керування дозволами та доступами	09.05.25	13.05.25	викон.
3	Реалізація програмного модуля керування дозволами та доступами	14.05.25	20.05.25	викон.
4	Тестування та розгортання програмного модуля керування дозволами та доступами	21.05.25	26.05.25	викон.
5	Оформлення пояснювальної записки.	27.05.25	29.05.25	викон.
6	Попередній захист БКР та доопрацювання.	30.05.25	01.06.25	викон.
7	Захист БКР	02.06.25	04.06.25	викон.

Студент _____ **Поліщук В. Л.**
(підпис) *(прізвище та ініціали)*

Керівник роботи _____ **Богач І. В.**
(підпис) *(прізвище та ініціали)*

АНОТАЦІЯ

Бакалаврська кваліфікаційна робота складається з 103 сторінок формату А4, на яких є 23 рисунки, список використаних джерел містить 25 найменувань.

В роботі представлено інформаційну технологію для керування дозволами та доступами з використанням Spring Security та JWT token на основі реляційної моделі даних. В даній бакалаврській роботі розроблені механізм автентифікації на основі Spring Security та JWT token, сконфігуровано CORS, TLS та реалізовано гнучку архітектуру керуванням дозволами та доступами з використанням мови програмування Java та Spring фреймворком.

Реалізація системи розглянута на прикладі забезпечення керування дозволами та доступами для системи управління виробничими процесами.

Розроблену систему було протестовано в ізольованому середовищі Docker контейнера з використанням інтеграційних та юніт-тестів.

Ключові слова: автентифікація, авторизація, підходи керування доступом, дозволи, компоненти користувацького веб-інтерфейсу, REST ендпойнт.

ABSTRACT

The Bachelor's degree qualification work consists of 103 pages of A4 format, which have 23 figures, the list of references contains 25 titles.

The work presents an information technology for managing permissions and accesses using Spring Security and JWT token based on a relational data model. In this bachelor's thesis, an authentication mechanism based on Spring Security and JWT token is developed, CORS, TLS are configured, and a flexible architecture for managing access and permissions is implemented using the Java programming language and Spring framework.

The implementation of the system is considered on the example of providing access and permission management for a production process control system.

The developed system was tested in an isolated Docker container environment using integration and unit tests.

Keywords: authentication, authorization, access control approaches, permissions, user web interface components, REST endpoint.

ЗМІСТ

ВСТУП

Актуальність дослідження. Із стрімким розвитком інформаційних технологій зростає складність інформаційних систем, а також обсяги даних, що обробляються та зберігаються в них. Такі додатки розробляються з метою оперування даними, що відображають певну предметну область із реального світу, та для забезпечення їхнього формалізованого збереження у базах даних. Таким чином також зростає необхідність забезпечення високого рівня безпеки та гнучкого керування дозволами та доступами до ресурсів інформаційних систем [1].

Такі підходи керування доступом як RBAC, ABAC покликані розв’язати цю проблему. Таке розмежування доступу є складовою багатьох сучасних комп'ютерних систем. Наприклад, підхід RBAC застосовується в системах захисту СУБД, а окремі його елементи реалізуються в мережевих операційних системах. Такі підходи застосовуються в системах, користувачам яких чітко призначено коло їх посадових повноважень і обов'язків [2].

RBAC вимагає заздалегідь визначених ролей, кожна з яких має унікальний набір дозволів. Ця модель авторизації є статичною і її не можна легко оновити, коли змінюються політики доступу в організації. Крім того, RBAC створює проблеми з масштабуванням, оскільки в структурі компанії з'являються нові ролі та посади.

ABAC забезпечує набагато більшу гнучкість у визначенні того, хто отримує доступ до того чи іншого ресурсу на підприємстві, незалежно від їхнього призначення. Він також дозволяє враховувати час і місцезнаходження запиту, що дає змогу надавати доступ до певної частини ресурсу та обмежувати його в часі, коли це необхідно [3].

Проте модифікація цього механізму досить важка, оскільки передбачає зміну атрибутів в бізнес лозіці програми, що вимагає технічних знань.

Ці моделі керування доступом не передбачають створення користувацьких шаблонів для дозволів та ролей, а також різноманітне поєднання дозволів. Крім того авторизація на основі цих методів як правило забезпечує лише керування доступом до компонентів користувацького інтерфейсу, не беручи до уваги обмеження доступу до окремих ендпойнтів REST API.

З вищевказаного можна зробити висновок, що розробка гнучкого механізму автентифікації та авторизації є актуальною та потребує подальшого дослідження.

Метою роботи є розширення функціональних можливостей механізму керування дозволами та доступами. На відміну від аналогічних підходів, які обмежуються фіксованими схемами та недостатньою деталізацією налаштувань, в роботі пропонується застосувати поєднання Spring Security, JWT Token та власноруч розроблену реляційну архітектуру сутностей, що забезпечує гнучкість і багаторівневий контроль доступу.

Об’єктом дослідження є процес керування дозволами та доступами до ресурсів системи користувачів.

Предметом дослідження є програмні засобі організації керування дозволами та доступами користувачів.

Задачі дослідження:

1. Здійснити аналіз предметної галузі керування дозволами та доступами.

2. Спроєктувати програмний модуль керування дозволами та доступами.

3. Реалізувати програмний модуль керування дозволами та доступами, розробити логіку програмного модуля та здійснити базове налаштування механізмів безпеки.

4. Виконати тестування програми та провести аналіз отриманих результатів;

5. Розробити інструкцію користувача.

Апробація роботи.

Результати роботи було представлено на LIV Всеукраїнській науково-технічній конференції підрозділів Вінницького національного технічного університету (2025) [4] та на Міжнародній науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2025)» [5].

Публікації: електронні тези конференції LIV Всеукраїнській науково-технічній конференції підрозділів Вінницького національного технічного університету (2025) [4]; Міжнародній науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2025)».

Впровадження. Результати роботи планується використати в розробках та управлінні проєктами ТОВ «ІННОВІННПРОМ» (додаток Д) та подано дві заявки на реєстрацію авторського права на твір (с202505171, с202505172).

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ

Керування доступом – це фундаментальна концепція інформаційної безпеки, яка використовується для визначення правил та механізмів, за якими користувачі отримують доступ до ресурсів інформаційної системи. Він гарантує, що доступ до даних або систем мають лише авторизовані та уповноважені особи, що дозволяє забезпечити конфіденційність, цілісність і доступність даних, а також мінімізувати ризики витоку інформації або несанкціонованого втручання.

По суті, механізми керування доступом передбачає проходження послідовних етапів: ідентифікацію, автентифікацію та авторизацію користувачів, гарантуючи, що дозволи на доступ узгоджуються з політиками організації та ролями користувачів. У найпростішій формі контроль доступу можна розглядати як комбінацію паролів і дозволів користувачів; однак сучасні системи контролю доступу включають в себе більш складні технології, такі як біометрична верифікація, багатофакторна автентифікація і динамічні структури на основі політик. Ці системи необхідні для управління безпекою в різних середовищах, від великих підприємств до малого бізнесу, і мають вирішальне значення для дотримання нормативних стандартів безпеки[3].

1.1 Основні підходи керування доступом

У сучасних інформаційних системах для регулювання прав користувачів і захисту ресурсів застосовують різні моделі контролю доступу, кожна з яких має свої особливості, переваги та недоліки. Нижче розглянуто п’ять базових підходів: контроль доступу на основі ролей (RBAC), дискретний контроль доступу (DAC), обов’язковий контроль доступу (MAC), атрибутний контроль доступу (ABAC) та політико-орієнтований контроль доступу (PBAC).

Контроль доступу на основі ролей (Role-Based Access Control, RBAC) став де-факто стандартом у багатьох корпоративних інформаційних системах. У цій моделі привілеї групуються в ролі, а користувачі отримують доступ, прив’язаний

до їхніх службових чи функціональних ролей. RBAC спрощує адміністрування за рахунок повторного використання наборів дозволів і дозволяє легко керувати змінами організаційних структур [6]. Проте традиційний RBAC є менш гнучким у ситуаціях, коли необхідно враховувати контекст запитів або складні умови доступу.

Переваги:

- Спрощення адміністрування: дозволи призначаються ролям, а не окремим користувачам.
- Покращена безпека: користувачі отримують лише ті дозволи, які необхідні для виконання їхніх обов'язків.
- Масштабованість: легко адаптується до змін в організаційній структурі.
- Покращення відповідності нормативним вимогам.

Недоліки:

- Може призвести до появи надмірної кількості мало відмінних ролей у великих системах.
- Обмежена гнучкість при необхідності врахування контексту доступу.
- Потребує ретельного планування та підтримки.

Дискретний контроль доступу (Discretionary Access Control, DAC) передбачає, що кожен власник об’єкта (файлу, пристрою, записи в базі даних) може самостійно визначати, хто отримає до нього доступ і з якими привілеями. Ця модель проста в реалізації та добре підходить для систем, де довірчі відносини між користувачами є ключовими, однак вона відкриває можливості помилкового чи зловмисного розповсюдження прав, адже користувачі можуть передавати свої дозволи іншим особам [7].

Переваги:

- Простота реалізації та розуміння.
- Гнучкість у наданні доступу.

Недоліки:

- Високий ризик несанкціонованого доступу через передачу дозволів.
- Складність у забезпеченні централізованого контролю.

Обов’язковий контроль доступу (Mandatory Access Control, MAC) забезпечує

жорсткий централізований механізм, у якому політики безпеки встановлюються адміністраторами без можливості подальшого їх зміщення кінцевими користувачами. Ресурси та суб’єкти системи маркуються рівнями довіри або класифікаціями (labels), і рішення про надання доступу приймаються лише на основі зіставлення цих міток, що гарантує високий рівень захищеності та відповідність суворим нормативним вимогам (наприклад, у військових чи державних системах) [7].

Переваги:

- Високий рівень безпеки та контроль.
- Підходить для середовищ з високими вимогами до безпеки (військові, урядові установи).

Недоліки:

- Обмежена гнучкість.
- Складність у налаштуванні та підтримці.

Контроль доступу на основі атрибутів (Attribute-Based Access Control, ABAC) охоплює більш тонкі політики, де рішення про авторизацію приймається на основі сукупності атрибутів суб’єкта (користувача), об’єкта (ресурсу), дії та контексту (час, місцезнаходження тощо). Таке рішення реалізується через правила, що дозволяють формулювати доволі складні умови доступу (наприклад, «дозволити зміну документа лише власнику з рівнем допуску вище Secret і у робочий час»). ABAC вважають «наступним поколінням» моделей доступу завдяки його динамічності та високій деталізації політик [8].

Переваги:

- Висока гнучкість та деталізація політик доступу.
- Можливість врахування контексту (час, місце, тип пристрою тощо).
- Підходить для динамічних та розподілених середовищ.

Недоліки:

- Складність у реалізації та підтримці політик.
- Високі вимоги до обчислювальних ресурсів.
- Потребує надійного управління атрибутами.

Політико-орієнтований контроль доступу (Policy-Based Access Control,

PBAC) поєднує ідеї RBAC та ABAC, коли адміністративні політики централізовано визначаються та застосовуються при кожному запиті. У PBAC доступ базується на формальних політиках (policy rules), а не виключно на ролях або атрибутах, що забезпечує гнучкість і можливість швидкого оновлення правил у відповідь на зміну бізнес-вимог чи загроз [9].

Переваги:

- Гнучкість у визначенні політик доступу.
- Можливість швидкого адаптування до змін у вимогах безпеки.

Недоліки:

- Складність у реалізації та управлінні політиками.
- Потребує високого рівня технічної експертизи.

Попри ефективність кожного з класичних підходів, жоден із них самостійно не задовольняє всі сучасні вимоги: дискретні моделі є занадто відкритими, обов’язкові — надто жорсткими, RBAC не забезпечує контекстної гнучкості, ABAC і PBAC — складні в реалізації та підтримці.

Актуальність створення власного гнучкого підходу обумовлена необхідністю поєднати переваги цих моделей: ієрархічний, детальний контроль доступу як у PBAC/ABAC, зручність ролей RBAC, можливість автопоновлення й шаблонів дозволів, а також гранульоване керування доступом до окремих DOM-компонентів UI та ендпойнтів REST API. Такий комбінований механізм, реалізований через Spring Security, JWT Token та розроблену реляційну архітектуру сутностей, дозволить забезпечити як і більш високий рівень безпеки, так і оперативну адаптивність до змін у виробничих процесах.

1.2 Аналіз систем-аналогів автоматизації виробничих процесів

У сфері автоматизації виробничих процесів існує низка платформ, що забезпечують ефективне управління IoT-пристроями, обробку даних та контроль доступу. Серед них варто виділити ThingsBoard та Thinger.io, які пропонують різні підходи до реалізації функціональності та безпеки.

ThingsBoard є відкритою IoT-платформою, яка підтримує різноманітні

протоколи зв'язку, такі як MQTT, CoAP та HTTP. Вона надає гнучку систему керування доступом на основі ролей (RBAC), що дозволяє налаштовувати права доступу до різних ресурсів платформи. Користувачі можуть взаємодіяти з REST API, використовуючи JWT-токени для автентифікації та авторизації .

ThingsBoard використовує JWT (JSON Web Token) для автентифікації REST API запитів. Після входу користувача система видає пару токенів:

- Access Token — токен з коротким терміном функціонування, який використовується для здійснення API-запитів.

- Refresh Token — використовується для отримання нового Access Token після закінчення терміну дії поточного.

За замовчуванням, Access Token дійсний протягом 2,5 годин, а Refresh Token — протягом одного тижня.

У ThingsBoard реалізовано розширене керування доступом на основі ролей (RBAC) (рисунок 1.1). Користувачам призначаються ролі з певними дозволами, які визначають доступ до ресурсів платформи, таких як пристрої, дашборди, користувачі та інші. Це дозволяє гнучко налаштовувати права доступу відповідно до потреб організації [10].

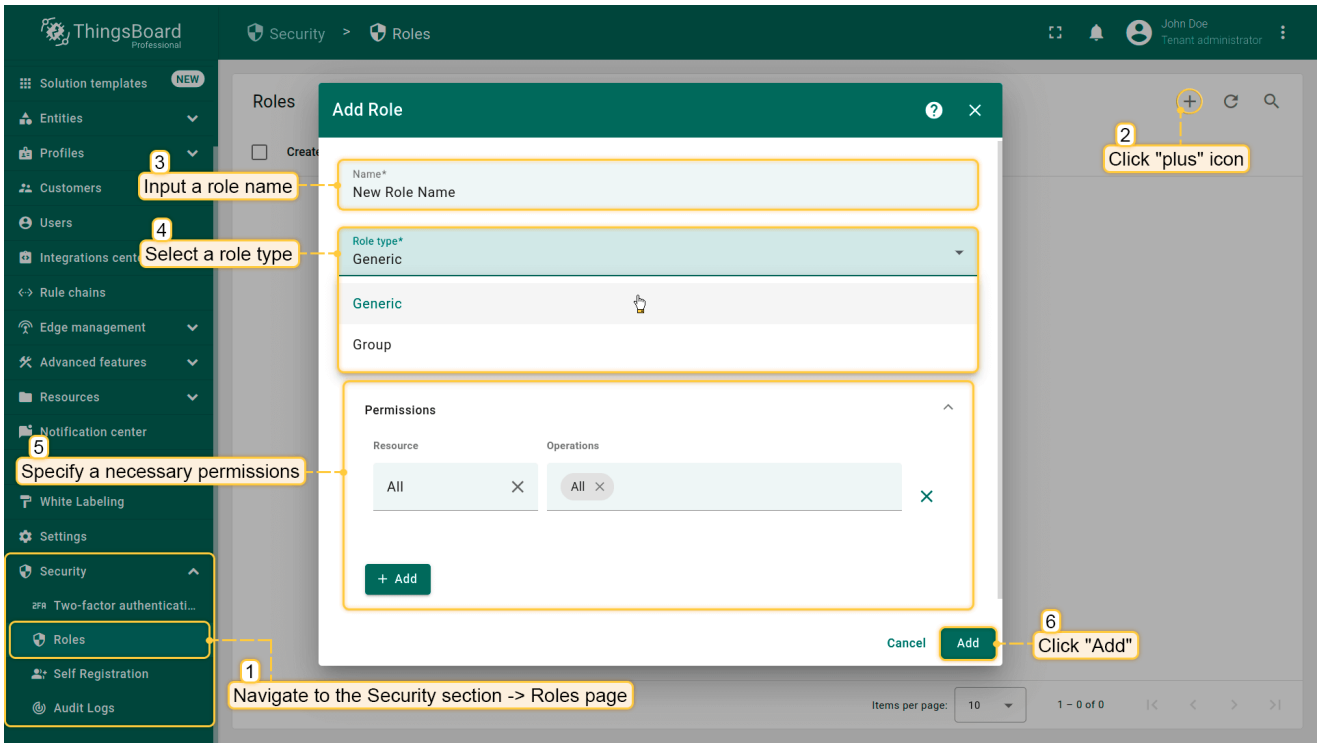

Рисунок 1.1 – Приклад створення ролей в системі ThingsBoard

У свою чергу керування доступом до пристроїв здійснюється наступним чином. Кожен пристрій у ThingsBoard може бути призначений певному користувачу або групі користувачів. Доступ до пристроїв контролюється через призначення ролей та дозволів, що забезпечує безпечне та контрольоване управління пристроями в системі.

Thinger.io також є відкритою IoT-платформою, яка фокусується на простоті використання та швидкому розгортанні рішень. Вона підтримує створення access tokens з детальними дозволами, що дозволяє контролювати доступ до пристроїв, даних та інших ресурсів платформи. Кожен токен може мати специфічні дозволи на читання, запис або експорт даних.

Усі взаємодії з підключеними пристроями в Thinger.io, включаючи REST API запити, потребують автентифікації. За замовчуванням, при взаємодії через консоль Thinger.io, всі запити підписуються access token, отриманим з вашого імені користувача та пароля. Однак для надання доступу іншим користувачам або платформам рекомендується створювати специфічні access tokens з обмеженими дозволами [11].

У Thinger.io (рисунок 1.2) кожен пристрій реєструється в системі, а доступ до нього контролюється через access tokens. Ці токени забезпечують авторизацію певних операцій з пристроєм. Деталізація дозволів здійснюється на рівні загальних операцій, без можливості гнучкого налаштування для окремих функцій пристрою.

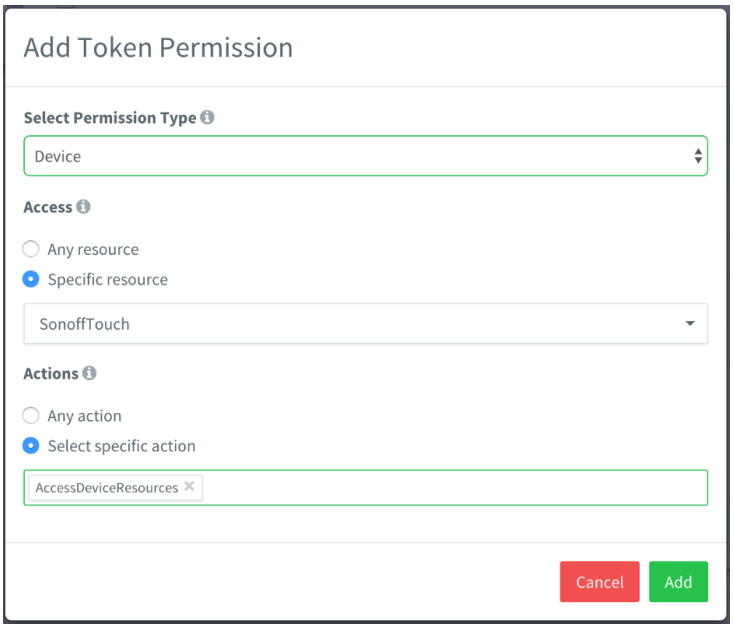

Рисунок 1.2 – Приклад призначення базових дозволів до токена в Thinger.io [12]

Платформа не надає безпосередньої підтримки керування доступом до окремих компонентів інтерфейсу користувача. Керування доступом здійснюється на рівні дашбордів, де можна обмежити доступ до певних віджетів або сторінок. Порівняльна характеристика з системами-аналогами наведена у таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика з системами-аналогами

Характеристика	Розроблена система	ThingsBoard	Thinger IoT Platform
1	2	3	4
Керування доступом до ендпойнтів REST API	Є можливість налаштовувати доступ до окремих ендпойнтів REST API через ролі та дозволи, забезпечуючи контроль доступу до API на рівні кінцевих точок. Крім того, підтримується автентифікація на основі JWT токена з цифровим підписом та секретного ключа	Підтримується через загальні ролі та дозволи, але без детального налаштування на рівні окремих ендпойнтів REST API.	Підтримується керування доступом як до окремих ендпойнтів, так і до всіх через механізм access tokens. Токени видаються для автентифікації REST API запитів, при цьому дозволи зазвичай визначаються на рівні заздалегідь визначених наборів дій або операцій.
Керування доступом до окремих UI компонентів	Ієрархічна модель доступу до DOM компонентів інтерфейсу: від сторінок/модулів до окремих кнопок, з можливістю відключення як батьківських елементів, так і вибіркового обмеження доступу до вкладених компонентів.	Немає	Безпосередньої підтримки немає, лише на рівні керування доступом до дашбордів.
Керування доступом до IoT пристроїв	Здійснюється завдяки керуванню дозволами до компонентів та ендпойнтів.	Підтримується через призначення ролей та дозволів на рівні пристроїв	Кожен пристрій реєструється в системі, а доступ до нього контролюється через access tokens, які забезпечують авторизацію певних операцій з пристроєм. Однак деталізація дозволів здійснюється на рівні загальних операцій, без можливості гнучкого налаштування для окремих функцій пристрою.

Продовження таблиці 1.1

1	2	3	4
Мультитенантність	Підтримується	Підтримується	Підтримується
Створення шаблонів для дозволів та автопризначення	Є можливість створювати шаблони дозволів системного рівня, які автоматично призначаються новим користувачам системи, спрощуючи процес управління доступом та забезпечуючи консистентність налаштувань безпеки.	Немає прямої підтримки створення шаблонів дозволів з автопризначенням новим користувачам.	Не підтримується: Thinger IoT Platform дозволяє створювати access tokens, але механізм шаблонів дозволів та автопризначення для нових користувачів відсутній.
Автооновлення дозволів	Є можливість автоматичного оновлення дозволів при зміні ролей або політик безпеки, забезпечуючи актуальність прав доступу без необхідності ручного втручання. (зміна шаблонів супроводжується зміною породжених ролей та дозволів)	Немає прямої підтримки автооновлення дозволів; зміни в ролях та дозволах вимагають ручного оновлення або повторного призначення.	Не підтримується: налаштування access tokens проводиться вручну, і автоматичне оновлення дозволів не передбачено.
Хто може створювати/редагувати ролі з дозволами	Адміністратор та датаменеджер підприємства	Адміністратор орендаря	Створення та редагування access tokens здійснюється адміністраторами платформи або власниками акаунтів, що відповідають за управління безпекою.

Аналізуючи функціональні можливості ThingsBoard та Thinger.io, можна визначити кілька напрямків для подальшого вдосконалення систем автоматизації виробничих процесів:

1. Гранульоване керування доступом до UI-компонентів: Впровадження ієрархічної моделі доступу до елементів інтерфейсу дозволить більш точно налаштовувати права користувачів, забезпечуючи безпеку та зручність використання.

2. Шаблони дозволів та автопризначення: Створення механізму шаблонів дозволів, які автоматично призначаються новим користувачам, спростить процес управління доступом та забезпечить консистентність налаштувань безпеки.

3. Автоматичне оновлення дозволів: Реалізація функціоналу автоматичного оновлення дозволів при зміні ролей або політик безпеки забезпечить актуальність прав доступу без необхідності ручного втручання.

4. Гнучке керування доступом до REST API: Деталізоване налаштування доступу до окремих ендпойнтів REST API дозволить точніше контролювати взаємодію з системою та забезпечить додатковий рівень безпеки.

5. Розширене керування доступом до IoT-пристроїв: Можливість гнучкого налаштування дозволів для окремих функцій пристроїв забезпечить більш точний контроль та управління ресурсами.

Застосування цих вдосконалень дозволить подолати обмеження існуючих рішень і реалізувати гнучку, масштабовану та безпечну систему керування доступом у виробничих середовищах.

1.3 Аналіз технологій, що використовуються в системах-аналогах

1.3.1 Spring Security

Spring Security — це потужний фреймворк для забезпечення безпеки в Java-додатках, який надає гнучкі механізми автентифікації та авторизації. Його архітектура базується на фільтрах, що дозволяє ефективно обробляти HTTP-запити та забезпечувати захист веб-додатків.

У центрі Spring Security знаходиться SecurityFilterChain — впорядкований ланцюг фільтрів, через який проходять всі вхідні HTTP-запити. Кожен фільтр у цьому ланцюжку виконує специфічну функцію, таку як автентифікація, авторизація, захист від CSRF-атак тощо. Ці фільтри конфігуруються за допомогою API SecurityFilterChain, що дозволяє налаштовувати безпеку для різних частин додатку незалежно. Структура ланцюгів SecurityFilterChain зображено на рисунку 1.3.

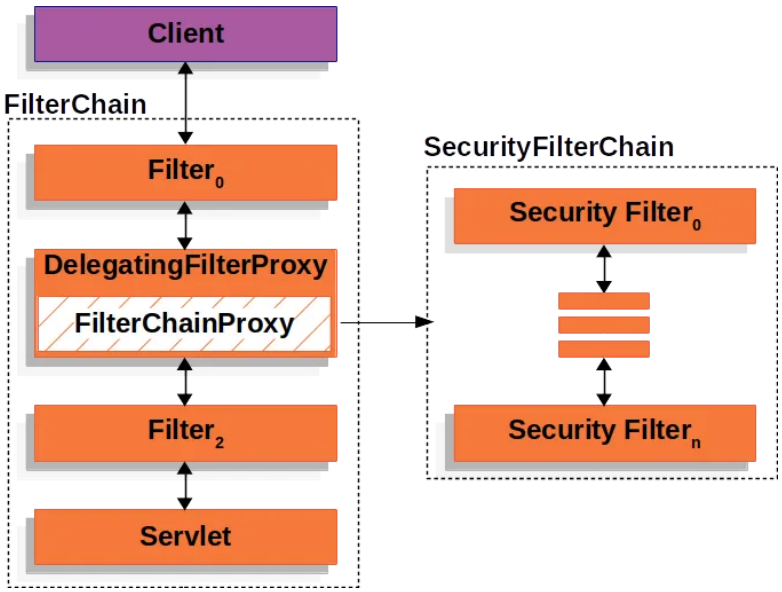

Рисунок 1.3 – Структура ланцюгів SecurityFilterChain

Spring Security зберігає інформацію про автентифікованого користувача в об'єкті SecurityContext, який, у свою чергу, зберігається в SecurityContextHolder. Це дозволяє отримувати дані про поточного користувача в будь-якій частині додатку. Після успішної автентифікації об'єкт Authentication, що містить деталі користувача, зберігається в SecurityContext.

Після автентифікації Spring Security використовує список GrantedAuthority, асоційований з користувачем, для визначення його прав доступу. Ці повноваження можуть представляти ролі або специфічні дозволи. Конфігурація правил авторизації здійснюється через DSL (Domain Specific Language), що дозволяє гнучко налаштовувати доступ до ресурсів, методів або URL-адрес. Наприклад, можна вказати, що доступ до певного ресурсу дозволений лише користувачам з конкретною роллю.

1.3.2 Spring Security OAuth2 Client

Spring Security OAuth2 Client — це модуль Spring Security, який реалізує роль клієнта відповідно до специфікації OAuth 2.0 Authorization Framework. Він дозволяє інтегрувати автентифікацію через сторонні провайдери (наприклад, Google, Facebook, GitHub) та забезпечує захищений доступ до ресурсів, що потребують авторизації.

Ключові функції:

- Підтримка різних грантів: Authorization Code, Refresh Token, Client Credentials, Resource Owner Password Credentials, JWT Bearer.
- Інтеграція з WebClient та RestTemplate: Можливість здійснювати запити до захищених ресурсів з автоматичним управлінням токенами.
- Конфігурація через DSL: Гнучке налаштування клієнтів OAuth 2.0 за допомогою DSL-конфігурації.

Spring Security OAuth2 Client базується на конфігурації через HttpSecurity.oauth2Client(), яка дозволяє налаштовувати основні компоненти клієнта OAuth 2.0. Це включає налаштування ClientRegistrationRepository, AuthorizedClientRepository та AuthorizedClientService, які відповідають за реєстрацію клієнтів, збереження авторизованих клієнтів та управління ними [13]. Архітектура OAuth2 Client наведена на рисунку 1.4.

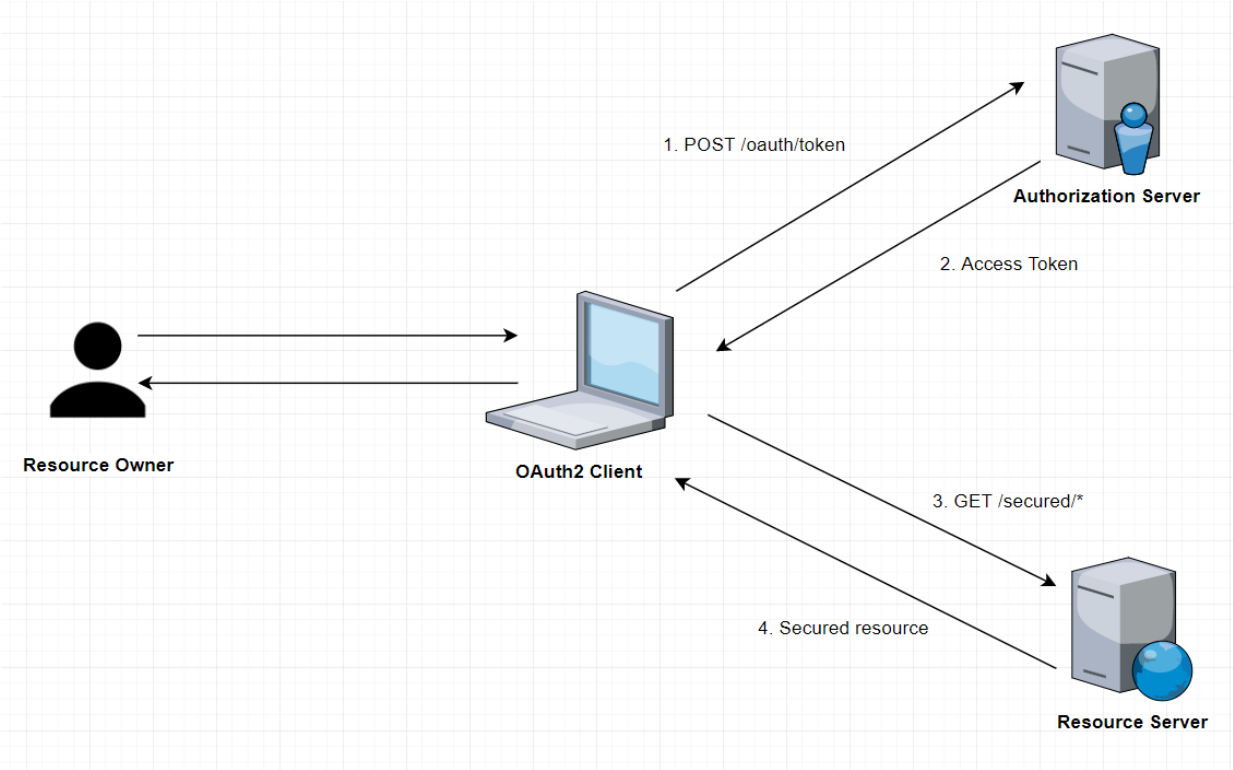

Рисунок 1.4 – Архітектура OAuth2 Client

Модуль підтримує кілька типів грантів, визначених в OAuth 2.0:

- Authorization Code Grant: використовується для отримання авторизаційного коду, який обмінюється на токен доступу.

- Refresh Token Grant: дозволяє оновлювати токен доступу без повторної автентифікації користувача.
- Client Credentials Grant: використовується для автентифікації клієнтів без участі користувача.
- Resource Owner Password Credentials Grant: дозволяє клієнту отримувати токен доступу, використовуючи облікові дані користувача.
- JWT Bearer Token Grant: дозволяє обмінювати JWT на токен доступу .

Spring Security OAuth2 Client забезпечує інтеграцію з WebClient та RestTemplate, дозволяючи здійснювати запити до захищених ресурсів з автоматичним управлінням токенами доступу. Це спрощує реалізацію клієнтів, які взаємодіють з ресурсами, що потребують авторизації .

Компонент OAuth2AuthorizedClientManager відповідає за управління авторизацією (або повторною авторизацією) клієнтів OAuth 2.0. Він працює у співпраці з одним або кількома OAuth2AuthorizedClientProvider, які реалізують логіку отримання та оновлення токенів доступу.

1.3.3 Java Json Web Token (JJWT)

Веб-токен JSON (JWT) є широко розповсюдженим і перевіреним засобом безпечного обміну інформацією у вигляді об'єктів JSON між суб'єктами. Токени цього типу вирізняються компактністю, що робить JWT гарним вибором для використання в середовищах HTML та HTTP, сумісністю з URL-адресами, підтримкою цифрового підпису за допомогою HMAC, що забезпечує розширені функції безпеки, а отже, є надійним варіантом для автентифікації в сучасних веб-додатках без сесії.

JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису (signature). Заголовок містить інформацію про тип токена та алгоритм підпису, напри-клад, {"alg": "HS256", "typ": "JWT"}. Корисне навантаження включає набір тверджень (claims) про суб'єкт токена, такі як {"sub": "1234567890", "name": "John Doe", "admin": true}. Твердження — це частини інформації, що заявляються про суб'єкта токена. Вони представлені у вигляді пар "ключ-значення". Специфікація JWT визначає сім зарезервованих тверджень, які

не є обов'язко-вими, але рекомендуються для забезпечення сумісності зі сторонніми додатками: iss (issuer — видавець токена), sub (subject — суб'єкт токена), aud (audience — аудиторія), exp (expiration time — час закінчення дії), nbf (not before time — час, раніше якого токен не є дійсним), iat (issued at time — час видачі) та jti (JWT ID — унікальний ідентифікатор токена). Підпис у JWT створюєть-ся шляхом використання секретного ключа та алгоритму підпису (наприклад, HMAC-SHA256). Спочатку секретний ключ декодується з формату Base64 у байтовий масив, після чого з нього створюється ключ HMAC. Далі цей ключ використовується для криптографічного підпису токе-на, що гарантує його цілісність та автентичність [4].

В результаті три частини кодуються за допомогою Base64url та об'єднуються крапками, утворюючи повний JWT, наприклад: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c. Закодований та декодований JWT токен наведено на рисунку 1.5.

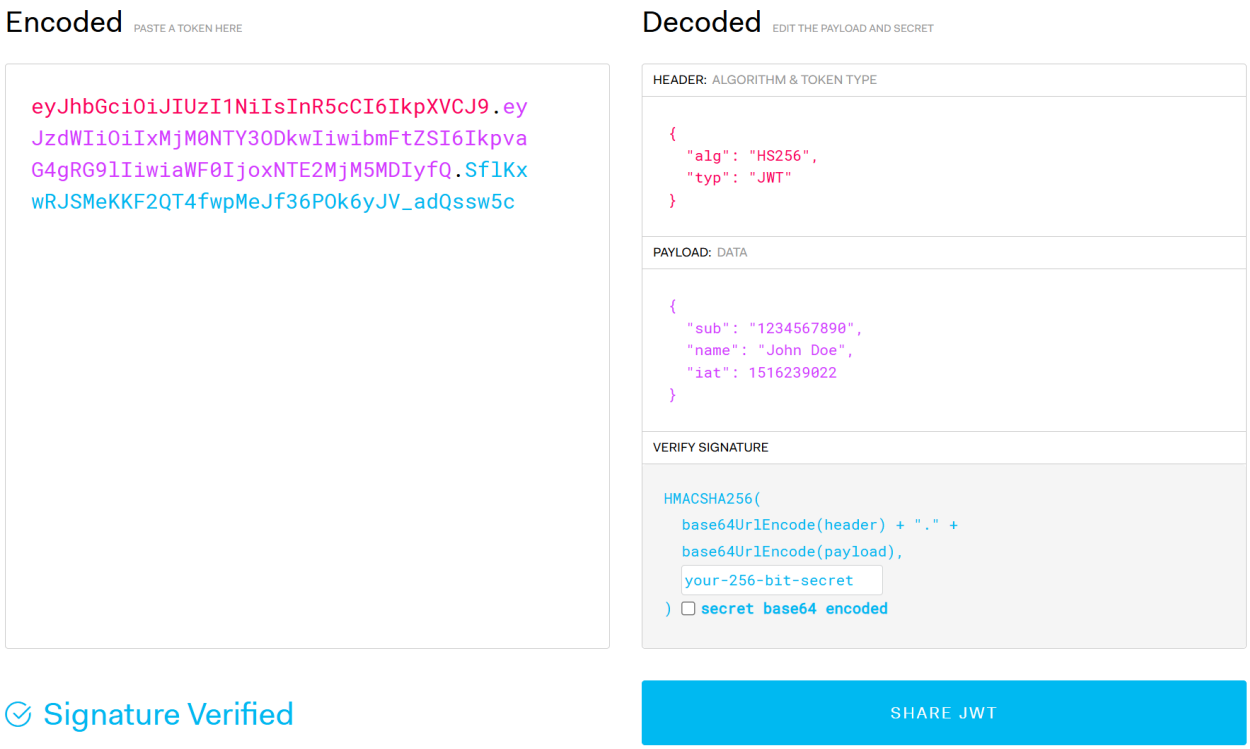

Рисунок 1.5 – Закодований та декодований JWT токен

Генерація охоплює такі кроки:

1) Встановлення імені поточного користувача як subject (sub) токена: твердження sub (subject) використовується для ідентифікації основного суб'єкта токена, тобто користувача, для якого цей токен було видано. Зазвичай у цьому полі зберігається унікальний ідентифікатор користувача, такий як ім'я користувача.

2) Встановлення дати створення та дати закінчення терміну дії токена: твердження iat (issued at) вказує час, коли токен було створено, і зазвичай представлений у форматі UNIX-часу. Твердження exp (expiration time) визначає момент, після якого токен вважається недійсним.

3) Підписання токена на основі секретного ключа з використанням алгоритму HS256: після визначення заголовка та корисного навантаження токена, вони кодуються за допомогою Base64url та об'єднуються через крапку. Отриманий рядок підписується з використанням алгоритму HMAC-SHA256 (HS256) та секретного ключа. Алгоритм HS256 використовує спільний секретний ключ як для створення, так і для перевірки підпису.

4) Встановлення додаткових тверджень: окрім стандартних, можна додавати власні кастомні твердження для передачі специфічної інформації, наприклад, ролі користувача або його прав доступу.

1.4 Висновки до розділу 1

В результаті дослідження було виявлено необхідність розширення функціональних можливостей механізму керування дозволами та доступами, що на відміну від аналогічних підходів, які обмежуються фіксованими схемами та недостатньою деталізацією налаштувань, забезпечує більшу гнучкість і багаторівневий контроль доступу

Під час аналізу аналогів було досліджено специфіку їхніх архітектурних рішень, зокрема засоби автентифікації та авторизації, інтеграцію з зовнішніми сервісами. Окрему розглянуто технології, які широко застосовуються для побудови безпечних веб-застосунків: Spring Security як основа авторизаційної логіки, Spring OAuth2 Client для взаємодії з зовнішніми провайдерами, а також JJWT.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ПРОГРАМНОГО МОДУЛЯ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ

2.1 Розробка архітектури системи

Першим етапом проєктування інформаційної системи є вибір базової архітектурної моделі, яка забезпечить необхідну функціональність, гнучкість, масштабованість і підтримку сучасних стандартів розробки. Для побудови даної системи обрано клієнт-серверну архітектуру, яка слугує фундаментом для більш складних багаторівневих структур і дозволяє чітко розділити функціональні обов’язки між окремими компонентами застосунку.

Клієнт-серверна архітектура (рис. 2.1) передбачає поділ системи на дві основні частини: клієнтську та серверну. Клієнтська частина відповідає за взаємодію з користувачем, включаючи збирання введених даних, відображення інформації та ініціацію запитів до серверної частини. У межах розроблюваної системи роль клієнта виконує фронтенд-застосунок, реалізований як веб-інтерфейс, що виконується в браузері користувача. У загальному випадку клієнт може також бути представлений мобільними або десктопними застосунками з графічним інтерфейсом.

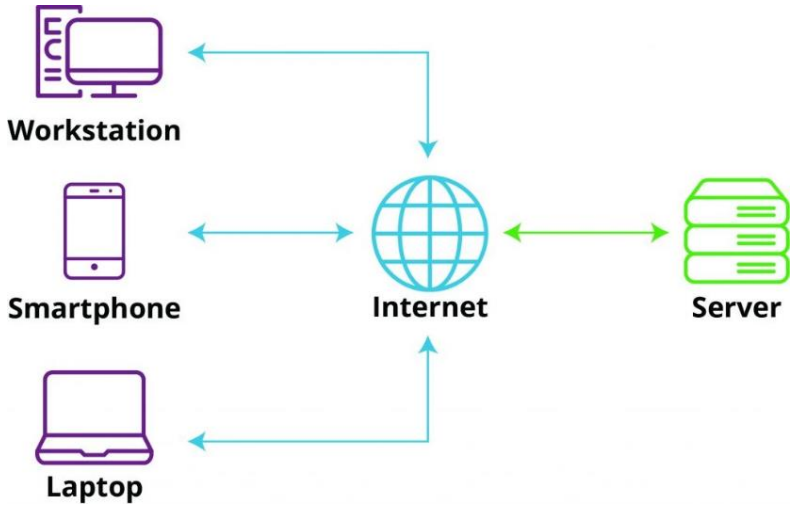

Рисунок 2.1 – Клієнт-серверна архітектура

Серверна частина відповідає за обробку запитів, реалізацію бізнес-логіки,

керування транзакціями та взаємодію з підсистемами зберігання даних. Взаємодія між клієнтським і серверним компонентами здійснюється з використанням стандартних мережевих протоколів, переважно HTTP або HTTPS, що забезпечує сумісність, масштабованість і платформонезалежність.

Однією з ключових переваг клієнт-серверної моделі є можливість незалежного розвитку клієнтської та серверної частин, а також масштабування серверної інфраструктури без впливу на кінцевих користувачів. Це дозволяє адаптувати систему до зростаючого навантаження, розгортати сервери у хмарних середовищах та оптимізувати продуктивність за рахунок розподілу обчислень. Крім того, централізація обробки запитів та зберігання даних на сервері спрощує впровадження засобів контролю доступу, шифрування і автентифікації [14].

2.1.1 Трирівнева архітектура

Трирівнева архітектура (Three-Tier Architecture) є поширеним архітектурним шаблоном у розробці програмного забезпечення, який забезпечує чітке розділення відповідальностей між різними компонентами системи. Цей підхід сприяє підвищенню масштабованості, підтримуваності та гнучкості застосунків, особливо в умовах складних корпоративних систем [15].

У межах трирівневої архітектури система поділяється на три основні рівні:

1. Презентаційний рівень (Presentation Layer): Цей рівень відповідає за взаємодію з користувачем, включаючи відображення інформації та збирання введених даних. У веб-застосунках це зазвичай реалізується за допомогою HTML, CSS та JavaScript, які виконуються в браузері користувача. Презентаційний рівень надсилає запити до рівня бізнес-логіки та відображає результати обробки.

2. Рівень бізнес-логіки (Business Logic Layer): Цей рівень містить основну бізнес-логіку та правила, які визначають поведінку системи. Він обробляє запити від презентаційного рівня, взаємодіє з рівнем доступу до даних для отримання або збереження інформації та повертає результати назад. Бізнес-логіка інкапсулює функціональність, специфічну для конкретного застосунку, забезпечуючи її незалежність від деталей реалізації інтерфейсу користувача або зберігання даних.

3. Рівень доступу до даних (Data Access Layer): Цей рівень відповідає за зберігання та отримання даних з бази даних або інших джерел. Він надає інтерфейси для взаємодії з даними, абстрагуючи деталі зберігання від рівня бізнес-логіки. Це дозволяє змінювати або оновлювати механізми зберігання без впливу на інші частини системи. Трирівнева архітектура програмного забезпечення зображена на рисунку 2.2.

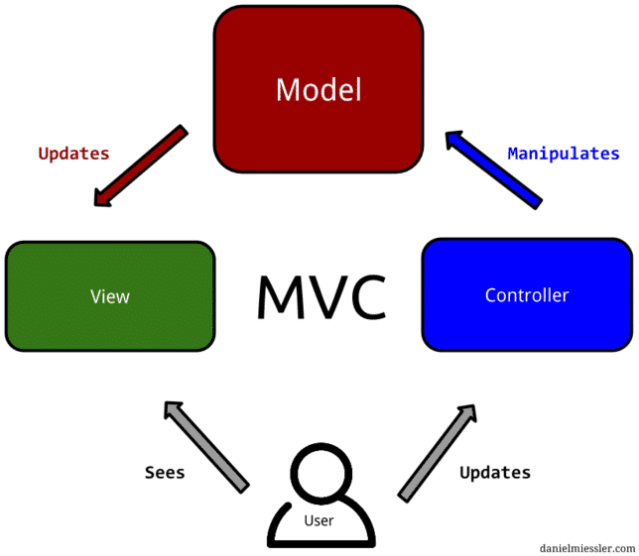

Рисунок 2.2 – Трирівнева архітектура програмного забезпечення

Така структура дозволяє кожному рівню розвиватися та масштабуватися незалежно, що сприяє підвищенню гнучкості та спрощує процес розробки та підтримки системи. Крім того, чітке розділення відповідальностей між рівнями забезпечує кращу організацію коду та полегшує тестування окремих компонентів.

2.1.2 Модель розгортання

У виборі моделі розгортання ключовим є баланс між простотою керування, масштабованістю та відповідністю вимогам до безпеки та продуктивності системи. У контексті даної роботи доцільно розглянути два основні підходи: монолітну реалізацію та мікросервісну архітектуру, порівняти їх переваги та обмеження і аргументувати вибір оптимальної моделі.

Монолітна архітектура. У монолітній моделі увесь функціонал системи упаковується в один єдиний виконуваний артефакт, де презентаційний шар, бізнес-

логіка та доступ до даних розгортаються й масштабуються як одне ціле. Така архітектура спрощує початкове розгортання і забезпечує просту організацію коду та єдиний контекст виконання. Завдяки зниженню міжпроцесорних викликів у межах одного процесу, моноліти можуть мати меншу затримку у внутрішній взаємодії компонентів. Однак масштабування моноліту вимагає розгортання нових екземплярів повного застосунку, що нерідко призводить до надмірного споживання ресурсів. Крім того, внесення змін, навіть у невеликий фрагмент логіки, потребує повторної збірки та розгортання всього застосунку, що збільшує час релізу та ризики регресій.

Мікросервісна архітектура. Мікросервіси розбивають застосунок на низку незалежних сервісів, кожен із власним контекстом даних і життєвим циклом розгортання. Кожен сервіс відповідає за окрему бізнес-функцію і взаємодіє з іншими через легковагі протоколи (наприклад, HTTP/REST або gRPC). Цей підхід дозволяє масштабу-вати лише ті сервіси, що відчувають підвищене навантаження, оптимізуючи використання ресурсів. Крім того, команда розробників може обирати найпідходящі технології для кожного сервісу, що підвищує гнучкість і прискорює впровадження інновацій. Однак мікросервіси ускладнюють тестування, моніторинг і трасування запитів через розподілену природу системи, а також потребують додаткових механізмів управління конфігурацією та безпекою міжсервісної взаємодії [16].

Порівняння підходів наведено в таблиці 2.1.

Таблиця 2.1 – Порівняльна характеристика моделей розгортання

	Моноліт	Мікросервіси
1	2	3
Складність розгортання	Просте: один артефакт, одна операція розгортання	Складне: кожен сервіс розгортається окремо; потрібні CI/CD конвеєри для кожного сервісу

Продовження таблиці 2.1.

1	2	3
Масштабованість	Масштабування всього застосунку цілком	Гранулярне масштабування окремих сервісів
Час релізу	Триваліший: упаковка та тест всього застосунку	Швидший: незалежні розгортання окремих сервісів
Управління залежностями	Легке: єдиний стек технологій	Складне: різні технології, версії, оновлення в кожному сервісі
Безпека	Централізована: єдина точка контролю доступу	Розподілена: необхідні механізми аутентифікації та авторизації між сервісами (mTLS, JWT)
Складність підтримки	Нижча на початковому етапі; з часом зростає через розміри коду	Висока: потребує системи оркестрації (Kubernetes), CI/CD, моніторингу, логування

Враховуючи мету роботи – розроблення гнучкої та масштабованої системи безпеки з детальним контролем доступу на рівні UI, REST API та даних – монолітна архітектура забезпечує простоту розгортання та централізацію механізмів безпеки, що полегшує реалізацію шаблонів дозволів та автооновлення політик. Водночас у рамках моноліту можливо впровадити трирівневу структуру, що дозволить модульно організувати презентаційний, бізнес- та шар доступу до даних, а також застосувати ключові патерни проєктування (наприклад, MVC, Strategy та Singleton). Складність мікросервісної архітектури та потреба у розподілених механізмах безпеки міжсервісної аутентифікації й авторизації можуть відволікати від головної мети – детального контролю доступу та гнучкого управління політиками у єдиному контексті. Таким чином, вибір монолітного розгортання із трирівневою структурою найбільш відповідає задачам даного дослідження, забезпечуючи баланс між гнучкістю, безпекою та ефективністю розробки.

2.1.3 Вибір патернів проєктування

Дотримання загальноприйнятих патернів проєктування є запорукою зрозумілого, підтримуваного й розширюваного коду. Патерни дозволяють формалізувати досвід і найкращі практики, зменшуючи зв’язність між компонентами та підвищуючи повторне використання рішень. У контексті системи з гнучкою моделлю безпеки й багаторівневою архітектурою обрані патерни допоможуть розділити зони відповідальності, спростити тестування й пришвидшити розвиток нових функціональних можливостей.

MVC (Model–View–Controller). Патерн MVC розділяє додаток на три логічні компоненти: модель, подання та контролер, що спрощує розробку та тестування інтерфейсу користувача. Модель інкапсулює дані та бізнес-логіку, подання відповідає за відображення отриманих з моделі даних користувачеві, а контролер обробляє вхідні запити, ініціює зміни в моделі та оновлює подання . Цей патерн дозволяє ізолювати логіку представлення від бізнес-логіки, зменшуючи залежності між компонентами та полегшуючи підтримку коду.

Dependency Inversion Principle. Принцип Dependency Inversion (DIP) стверджує, що високорівневі модулі не повинні залежати від низькорівневих; обидва мають залежати від абстракцій, а абстракції не повинні залежати від деталей . Таким чином, бізнес-логіка (високий рівень) залежить від інтерфейсів (абстракцій), а реалізації (низький рівень) — від тих же інтерфейсів, що дозволяє підміняти модулі без зміни клієнтського коду . Застосування DIP підвищує тестованість і знижує зв’язність компонентів системи, сприяючи гнучкості й зручності впровадження нових механізмів доступу чи безпеки.

Builder (для DTO). Патерн Builder використовується для поетапного конструювання складних об’єктів, таких як DTO (Data Transfer Object), без необхідності передавати до конструктора численні параметри . Він дозволяє задавати лише необхідні атрибути, лаконічно викликаючи методи withXxx(), а потім формувати об’єкт викликом build(), що підвищує читаність коду та попереджає помилки від невірно переданих параметрів . Це особливо корисно для передачі даних між рівнями системи, коли потрібно інкапсулювати безпечні поля та уникнути некоректної ініціалізації об’єктів .

Facade (для інтерфейсів бізнес-логіки). Патерн Facade визначає єдиний інтерфейс до набору інтерфейсів підсистеми, спрощуючи взаємодію з бізнес-логікою та приховуючи складність внутрішніх компонентів . Впровадження фасаду забезпечує клієнтським класам простий доступ до бізнес-процесів (наприклад, авторизації, керування ролями), зменшуючи специфікацію залежностей і підвищуючи рівень абстракції . Це дозволяє змінювати внутрішню реалізацію бізнес-логіки без впливу на клієнтський код, що особливо важливо для систем із частими оновленнями політик безпеки та доступу .

2.2 Вибір архітектурного стилю

У процесі розробки сучасних веб-застосунків важливим аспектом є вибір відповідного архітектурного стилю для побудови API. Серед найпоширеніших підходів виділяють REST (Representational State Transfer) та GraphQL. Кожен з них має свої переваги та недоліки, які слід враховувати при прийнятті рішення.

REST — це архітектурний стиль, який використовує стандартні HTTP-методи (GET, POST, PUT, DELETE) для взаємодії між клієнтом і сервером. Основною ідеєю REST є представлення ресурсів через URL та використання стандартних методів для їх обробки.

Переваги REST:

- Простота та зрозумілість: REST використовує стандартні HTTP-методи, що робить його легким для розуміння та впровадження.

- Кешування: REST дозволяє ефективно використовувати кешування, що зменшує навантаження на сервер і покращує продуктивність.

- Масштабованість: Завдяки безстанності (statelessness) REST-підходу, сервер не зберігає інформацію про стан клієнта, що спрощує масштабування системи.

- Широка підтримка: REST підтримується багатьма фреймворками та інструментами, що полегшує його інтеграцію в різні системи.

Недоліки REST:

- Перевантаження або недостатність даних: REST може повертати або надмірну, або недостатню кількість даних, що призводить до необхідності робити додаткові запити.

- Фіксовані структури відповідей: REST має жорстко визначені структури відповідей, що може бути незручним при потребі отримання специфічних даних.

- Відсутність типізації: REST не забезпечує вбудованої типізації, що може ускладнювати валідацію даних на клієнтській стороні.

GraphQL — це мова запитів для API, розроблена Facebook, яка дозволяє клієнтам запитувати саме ті дані, які їм потрібні. На відміну від REST, де кожен ресурс має свій URL, GraphQL використовує єдиний ендпоінт для обробки всіх запитів.

Переваги GraphQL:

- Гнучкість запитів: Клієнт може запитувати лише ті поля, які йому потрібні, що зменшує обсяг переданих даних.

- Єдиний ендпоінт: Усі запити обробляються через один ендпоінт, що спрощує структуру API.

- Типізація: GraphQL використовує схеми з чіткою типізацією, що полегшує валідацію та автогенерацію документації.

- Зручність для клієнта: Клієнт має більше контролю над тим, які дані він отримує, що покращує ефективність взаємодії.

Недоліки GraphQL:

- Складність реалізації: Впровадження GraphQL може вимагати більше зусиль, особливо при міграції з REST.

- Відсутність кешування на рівні HTTP: Через використання єдиного ендпоінту стандартне HTTP-кешування стає менш ефективним.

- Проблеми з безпекою: Гнучкість запитів може призвести до складнощів у контролі доступу та захисті від зловживань [17].

Вибір між REST та GraphQL залежить від конкретних потреб проєкту. REST є зрілим та стабільним підходом, який добре підходить для більшості стандартних веб-застосунків. GraphQL, у свою чергу, надає більшу гнучкість та ефективність у

роботі з даними, але вимагає більшої уваги до реалізації та безпеки. У контексті даного проєкту, враховуючи його масштаб, вимоги до безпеки та необхідність швидкої реалізації, було прийнято рішення використовувати REST як основний архітектурний стиль для побудови API. Це дозволить забезпечити стабільність, простоту обслуговування та легкість інтеграції з іншими системами.

Висновок. Клієнт-серверна архітектура, трирівнева архітектура та архітектура REST — це основоположні концепції розробки програмного забезпечення. Їхнє розуміння допомагає проєктувати, створювати та підтримувати надійні й масштабовані застосунки. Архітектура вашого застосунку, ймовірно, поєднує кілька з цих шаблонів, щоб задовольнити конкретні вимоги та забезпечити оптимальний користувацький досвід.

Структура нашого проєкту розроблена таким чином, щоб сприяти організації коду, його підтримуваності та командній роботі. Вона забезпечує чіткий розподіл відповідальностей, дотримується стандартів кодування та включає необхідні конфігураційні файли для інструментів розробки та збірки.

2.3 Вибір інструментів розробки

2.3.1 Мова програмування

У межах реалізації програмного модуля для керування доступом у веб-застосунку було прийнято рішення використовувати мову програмування Java. Це вибір, обґрунтований як технічними, так і концептуальними міркуваннями, що безпосередньо пов’язані з вимогами до стабільності, безпеки, масштабованості та підтримки сучасних фреймворків.

Java — це мова програмування загального призначення, яка працює за принципом "Write Once, Run Anywhere" (WORA), тобто один раз написаний код можна запускати на будь-якій платформі, що підтримує JVM (Java Virtual Machine). Завдяки цьому Java є однією з найпопулярніших мов для серверної розробки, особливо у корпоративному середовищі [18].

Однією з основних переваг Java є зріла екосистема та широкий набір бібліотек і фреймворків. Крім того, Java має добре опрацьовану документацію,

високу якість інструментів розробки, потужну підтримку тестування та інтеграції з базами даних, що є критичним для побудови надійних backend-систем.

Важливим фактором на користь вибору Java є її типізація та об’єктно-орієнтована модель програмування, які забезпечують високу читабельність коду, покращене тестування та простішу підтримку проєкту в довгостроковій перспективі. Завдяки строгій типізації, Java дозволяє уникнути багатьох помилок ще на етапі компіляції, що позитивно впливає на стабільність програмного забезпечення.

Крім того, Java активно розвивається: сучасні версії (починаючи з Java 17 і вище) підтримують лаконічніші конструкції, покращене керування пам’яттю та нові можливості, які наближають її до функціональних мов, не втрачаючи класичних переваг. Це робить мову ще більш ефективною для розробки як великих, так і середніх веб-систем.

У межах цієї роботи Java обрана як основна мова для серверної частини веб-застосунку, оскільки вона:

- активно підтримується великою спільнотою, що забезпечує наявність численних бібліотек, інструментів і рішень для широкого спектра задач, зокрема у сфері веб-розробки та розробки корпоративних застосунків;
- дозволяє реалізувати ООП архітектурні патерни (наприклад, MVC, фасад тощо);
- забезпечує стабільність і масштабованість програмного рішення;
- підтримується більшістю сучасних інструментів CI/CD і хмарних платформ.

2.3.2 Середовище розробки

Для ефективної реалізації програмного забезпечення важливо обрати сучасне інтегроване середовище розробки (IDE), яке забезпечує не лише зручне написання та рефакторинг коду, але й підтримку збірки проєкту, керування залежностями, тестування, налагодження, інтеграцію з системами контролю версій тощо. Правильно обране середовище розробки дозволяє підвищити продуктивність розробника, зменшити ймовірність помилок та спростити процес супроводу

проєкту.

У рамках розробки даної системи як основне середовище було обрано IntelliJ IDEA — одну з найпопулярніших серед Java-розробників інтегрованих середовищ розробки, розроблену компанією JetBrains. IntelliJ IDEA підтримує всі ключові аспекти розробки застосунків на Java, Kotlin та інших JVM-мовах, пропонуючи глибоку інтеграцію з екосистемою Java та сучасними інструментами розробки [19].

Серед переваг IntelliJ IDEA варто виокремити наступні:

- інтелектуальний аналіз коду — IDE пропонує автоматичні підказки, перевірку синтаксису, швидкі виправлення помилок, а також рефакторинг на рівні проєкту, що значно полегшує розробку та супровід коду;
- зручна навігація — завдяки потужному механізму індексації, IntelliJ IDEA забезпечує миттєвий перехід між класами, методами, інтерфейсами, а також пошук використань, що дозволяє ефективно працювати навіть з великими кодовими базами;
- вбудовані інструменти для роботи з системами контролю версій (наприклад, Git) — забезпечують зручну візуалізацію історії змін, конфліктів злиття та можливість роботи з репозиторіями безпосередньо з IDE;
- підтримка інструментів керування залежностями — таких як Maven і Gradle, які є стандартними у світі Java-розробки;
- підтримка JUnit і інших фреймворків для модульного тестування — дозволяє запускати тести, переглядати покриття коду та налагоджувати тестову логіку без виходу з IDE;
- можливість розширення — завдяки великій кількості плагінів, доступних у JetBrains Marketplace, IDE можна адаптувати до специфіки будь-якого проєкту.

Також варто зазначити, що IntelliJ IDEA підтримує як локальні проєкти, так і хмарні рішення, має функціонал для роботи з базами даних, а також дозволяє зручно працювати з REST API та JSON-даними — усе це робить середовище універсальним інструментом для реалізації як простих, так і складних клієнт-серверних систем. Обраний інструмент дозволив забезпечити стабільний та ефективний процес розробки протягом усього життєвого циклу створення застосунку — від проєктування до впровадження.

2.3.3 Фреймворки

Під час створення сучасних веб-застосунків ключову роль відіграє вибір відповідного фреймворку — інструментарію, що забезпечує готову інфраструктуру для реалізації типових завдань програмування, спрощує конфігурацію, підвищує безпеку та скорочує час розробки. Для реалізації серверної частини застосунку в даному проєкті було обрано фреймворк Spring, а для забезпечення безпеки — Spring Security, що є його окремим модулем.

Spring Framework — це потужний і розширюваний фреймворк з відкритим кодом, орієнтований на побудову корпоративних Java-застосунків. Він є одним із найбільш поширених рішень у Java-екосистемі завдяки своїй модульній структурі, активній спільноті, широкій підтримці стандартів Java EE та високій сумісності з іншими інструментами й технологіями [20].

Серед основних переваг використання Spring варто відзначити:

- Інверсія управління (IoC) — дозволяє ефективно керувати залежностями між компонентами застосунку за допомогою механізмів впровадження залежностей (dependency injection), що підвищує гнучкість і модульність системи;
- Модульність — розділення на окремі компоненти (Spring Core, Spring Web, Spring Data тощо) дозволяє використовувати лише ті частини, які дійсно потрібні, без перевантаження проєкту зайвими залежностями;
- Підтримка REST API /— Spring MVC надає зручний механізм для реалізації контролерів REST, обробки HTTP-запитів, серіалізації/десеріалізації даних, керування помилками та валідації;
- Простота тестування — завдяки чіткій структурі та використанню IoC контейнери Spring легко піддаються юніт-тестуванню.

Для реалізації функціоналу автентифікації, авторизації та загального контролю доступу у системі додатково інтегровано модуль Spring Security. Цей фреймворк надає гнучкі засоби для керування доступом на основі ролей та правил, захисту REST-інтерфейсів, зберігання паролів у безпечному вигляді, інтеграції з JWT (JSON Web Token), OAuth2 або іншими механізмами безпеки.

Основні переваги Spring Security:

- Гнучка система авторизації — підтримка як декларативної, так і програмної перевірки прав доступу;

- Механізми автентифікації — як базові (логін/пароль), так і розширені (через токени, сторонні провайдери тощо);

- Захист від поширених атак — CSRF, Session Fixation, Clickjacking, Brute-force login тощо;

- Простота інтеграції — повна сумісність із рештою модулів Spring, зокрема Spring MVC та Spring Boot.

У межах проєкту також використовується Hibernate — ORM-бібліотека (Object-Relational Mapping) для Java, яка виступає стандартною реалізацією Java Persistence API (JPA). Взаємодія з Hibernate в проєкті здійснюється через модуль Spring Data JPA, що забезпечує зручний та абстрагований доступ до бази даних без потреби у написанні великої кількості SQL-запитів вручну.

Основною метою використання Hibernate є перетворення об'єктно-орієнтованої моделі застосунку на реляційну модель зберігання в базі даних та навпаки. Це дозволяє:

- автоматизувати збереження, оновлення, видалення та отримання сутностей (Entity);

- уникнути прямого написання SQL-запитів завдяки використанню HQL (Hibernate Query Language) або Spring Data репозиторіїв;

- працювати з транзакціями, каскадними операціями, зв’язками (one-to-many, many-to-one, many-to-many) та lazy-loading без необхідності вручного керування кожним кроком;

- зменшити кількість шаблонного коду завдяки автоматичному створенню реалізацій CRUD-методів.

Spring Data JPA надбудовується над Hibernate і надає потужний рівень абстракції: достатньо створити інтерфейс-репозиторій і описати лише сигнатури методів, а їх реалізація буде згенерована автоматично під час виконання. Крім того, підтримуються кастомні SQL-запити, проекції DTO, пагінація, сортування та інші можливості [21].

Вибір саме цих фреймворків зумовлений необхідністю побудови

масштабованого, безпечного та підтримуваного веб-застосунку. Їх широке використання в індустрії, активна підтримка та добре задокументована функціональність забезпечують надійну основу для реалізації проєкту в межах обраної клієнт-серверної архітектури.

2.3.4 Інструменти керування проєктами та їх залежностями

У процесі розробки програмного забезпечення важливою складовою є ефективне керування залежностями, конфігурацією збірки та структурами проєкту. З цією метою у межах даного проєкту було обрано Apache Maven — один із найпоширеніших інструментів керування проєктами для Java-платформи.

Maven є інструментом автоматизації збірки, що ґрунтується на концепції декларативного підходу. Замість написання численних скриптів збірки, як це передбачено в більш низькорівневих інструментах, конфігурація Maven здійснюється за допомогою XML-файлу pom.xml, де описується структура проєкту, його залежності, плагіни, профілі, життєвий цикл збірки та інші параметри.

Серед ключових переваг використання Maven у даному проєкті можна виділити [22]:

- Керування залежностями: Maven дозволяє централізовано керувати бібліотеками та фреймворками, завантажуючи їх автоматично з віддалених репозиторіїв (наприклад, Maven Central). Це значно спрощує оновлення та підключення сторонніх компонентів.
- Стандартизована структура проєкту: Maven встановлює чітку організацію каталогів (src/main/java, src/test/java тощо), що полегшує підтримку та розуміння проєкту іншими розробниками.
- Масштабованість: Maven підтримує багато-модульні проєкти, що дозволяє легко керувати великими системами, розділеними на окремі компоненти.
- Плагіни для розширення функціональності: існує велика кількість офіційних та сторонніх плагінів, які дозволяють інтегрувати інструменти тестування, генерації документації, складання архівів (JAR/WAR), запуску серверів тощо.

- Інтеграція з IDE: Maven підтримується усіма сучасними середовищами розробки, зокрема IntelliJ IDEA, що дозволяє повністю автоматизувати процес збірки, оновлення залежностей та запуску застосунку без потреби в додатковій ручній конфігурації.

Таким чином, Apache Maven став логічним вибором для проєкту, побудованого на базі Java, оскільки забезпечує гнучкість, повторюваність, стандартизацію процесу збірки та зменшує ймовірність помилок, пов’язаних із ручним налаштуванням середовища розробки.

2.4 Висновки до розділу 2

У результаті другого розділу було спроєктовано архітектуру програмного забезпечення. Обґрунтовано вибір трирівневої архітектури як такої, що забезпечує розділення відповідальностей між рівнями Presentation, Business і Persistence, спрощує супровід, масштабування та повторне використання компонентів. Розглянуто можливі варіанти моделі розгортання, зокрема традиційний моноліт, мікросервісний підхід та використання контейнеризації, що дало змогу вибрати оптимальну стратегію з урахуванням гнучкості, продуктивності та потенційного розширення системи.

Також було обґрунтовано вибір архітектурного стилю та ключових патернів проєктування, які покращують структурованість та масштабованість проєкту. Використання таких патернів, як MVC, Dependency Inversion, Builder і Facade, дозволяє підвищити гнучкість модифікацій та зрозумілість коду. На основі аналізу інструментів розробки було обрано технологічний стек, що включає Java як основну мову програмування, фреймворки Spring Boot, Spring Security, Hibernate, а також інтегроване середовище IntelliJ IDEA та систему керування залежностями Maven. Такий вибір забезпечує високу продуктивність розробки, безпеку та можливість легкої інтеграції із зовнішніми компонентами.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО МОДУЛЯ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ

3.1 Налаштування механізму автентифікації та конфігурація захищеного доступу

У сучасних веб-застосунках безпека є критичним аспектом, що впливає не лише на довіру користувачів, а й на цілісність даних та стабільність усієї системи. Забезпечення захищеного доступу до ресурсів передбачає впровадження надійних механізмів автентифікації, авторизації та шифрування каналів зв’язку. З огляду на загрози, пов’язані з несанкціонованим доступом, перехопленням даних або міжсайтовими атаками, проєктування безпекового шару потребує уважного аналізу та реалізації перевірених підходів.

Під час реалізації даного програмного забезпечення було прийнято рішення побудувати систему захисту, спираючись на стандарти індустрії та відкриті технології, які зарекомендували себе як ефективні та масштабовані. Зокрема, було обрано токен-орієнтовану автентифікацію, конфігурацію CORS для контролю міждоменної взаємодії, використання TLS/SSL для захисту переданих даних, а також інтеграцію Spring Security як основного інструменту керування доступом на рівні застосунку.

3.1.1 Налаштування автентифікації на основі JWT токена

Для автентифікації користувачів в програмному модулі передбачено використання токенів JSON Web Token (JWT). Такий підхід забезпечує безпечну, незалежно від сесії та масштабовану автентифікацію користувачів у контексті веб-архітектури.

Перед налаштуванням JWT-автентифікації необхідно підготувати такі ресурси:

- Файл ключа name_key.der, який буде використовуватись для підпису та перевірки JWT-токенів. Цей файл має бути розміщений у директорії resources/keys проєкту.

- Файл конфігурації application-{profile}.yml, у якому задаються параметри налаштування JWT-механізму.

Далі потрібно налаштувати основні параметри конфігурації JWT за допомогою наступних властивостей конфігураційного файлу Spring Boot застосунку:

1. Шлях до приватного ключа. Властивість private-key-path повинна містити шлях до директорії, де зберігається файл ключа name_key.der, що використовується для підпису токенів.

private-key-path: dir1/dir2/

2. Назва приватного ключа У параметрі private-key-name задається точна назва файлу приватного ключа.

private-key-name: name_key.der

3. Час дії токена Визначаються два типи параметрів терміну дії токенів:

- token-validity-in-seconds — тривалість дії стандартного JWT-токена в секундах.

- token-validity-in-seconds-for-remember-me — тривалість дії токена з опцією "запам’ятати мене". Приклад конфігурації:

token-validity-in-seconds: 3600 # 1 година

token-validity-in-seconds-for-remember-me: 604800 # 7 днів

Механізм автентифікації на основі JWT було реалізовано у кілька послідовних етапів:

1. Імплементація утилітного класу JWT. Розроблено окремий сервіс для генерації та перевірки JWT-токенів з використанням приватного ключа. У межах цього класу реалізовані методи для шифрування та розшифрування токенів відповідно до обраного алгоритму.

2. Інтеграція з Spring Security. Spring Security налаштовано на обробку JWT-токенів шляхом додавання фільтра, який перехоплює HTTP-запити, перевіряє наявність токена та валідність його підпису.

3. Логіка автентифікації та авторизації. Реалізація механізму керування дозволами та доступами детально описана в наступному підрозділі 3.2.

4. Захист ресурсів. Доступ до захищених ендпоінтів реалізовано шляхом

використання відповідних анотацій Spring Security (@PreAuthorize, @Secured) або за допомогою конфігураційного класу безпеки, що визначає правила авторизації.

Реалізація автентифікації на основі JWT у Spring Boot забезпечує безпечний та безстанний спосіб керування доступом користувачів. Правильне налаштування файлу application-(ваш профіль).yml та дотримання найкращих практик гарантує надійність і цілісність вашої системи автентифікації.

3.1.2 CORS-конфігурація (CORS Configuration)

Cross-Origin Resource Sharing (CORS) — це механізм, який дозволяє браузеру контролювати, чи можуть ресурси одного джерела (origin) використовуватись вебзастосунками, що завантажені з іншого джерела. У рамках розробленого програмного забезпечення реалізовано конфігурацію CORS для забезпечення доступу до серверної частини лише з дозволених джерел, а також для дотримання сучасних вимог до безпечного обміну даними між фронтендом і бекендом.

Конфігурація в application-*.yml Налаштування параметрів CORS у Spring Boot застосунку здійснено у файлі application-(профіль).yml, що відповідає середовищу розгортання (наприклад, dev, stage або prod). Фрагмент конфігураційного блоку має такий вигляд:

```
cors:
allowed-origins:
'http://localhost:8100,https://localhost:8100,http://localhost:9000,https://localhost:9000,http://dev.innovinnprom.com,https://dev.innovinnprom.com,http://stage.innovinnprom.com,https://stage.innovinnpron.com,http://cloud.innovinnprom.com,https://cloud.innovinnprom.com'
#allowed-origin-patterns: 'https://*'
allowed-methods: '*'
allowed-headers: '*'
exposed-headers: 'Authorization,Link,X-Total-Count'
allow-credentials: true
max-age: 1800
```

У цій конфігурації передбачено наступні параметри:

- allowed-origins — визначає список дозволених джерел (URL-адрес), з яких допускаються запити до ресурсів сервера. Перелік включає як локальні адреси для розробки, так і тестові та staging-домени для інтеграційного доступу.

- allowed-methods — вказує, які HTTP-методи дозволені у кросдоменно-запитах. Символ "*" дозволяє всі методи (GET, POST, PUT, DELETE тощо).

- allowed-headers — перелік HTTP-заголовків, які допускаються у запитах з інших джерел. "*" означає дозвіл на всі можливі заголовки.

- exposed-headers — визначає заголовки, які стають доступними JavaScript-коду на клієнтській стороні. У цьому випадку: Authorization, Link, X-Total-Count.

- allow-credentials — за умови значення true дозволяється передавання куків, заголовків авторизації та інших облікових даних під час кросдоменної взаємодії.

- max-age — встановлює час життя (у секундах) результатів preflight-запитів (OPTIONS), після якого браузер знову перевірятиме дозволи. У прикладі значення 1800 відповідає 30 хвилинам.

Після налаштування, CORS-конфігурація автоматично інтегрується в механізм обробки запитів Spring Boot. Завдяки цьому, вебзастосунки, що працюють на визначених доменах, можуть взаємодіяти з API серверної частини у межах дозволених політик. Це забезпечує контрольований доступ до ресурсів, а також дотримання вимог безпеки під час взаємодії клієнта з сервером.

Застосована конфігурація дозволяє зберігати баланс між зручністю інтеграції різних частин застосунку (наприклад, фронтенду та бекенду) і дотриманням обмежень, які гарантують безпечну обробку HTTP-запитів з інших джерел.

3.1.3 Конфігурація TLS/SSL

Для забезпечення захищеного каналу зв’язку між клієнтом і сервером у Spring Boot застосунку реалізовано підтримку шифрування з використанням протоколу TLS/SSL. Відповідні параметри конфігурації задаються у спеціальному файлі середовища — application-tls.yml, який активується при запуску застосунку з відповідним профілем. Основні параметри конфігурації Конфігураційний блок server.ssl містить ключові параметри, необхідні для активації TLS/SSL у серверній частині застосунку:

```
server:
  ssl:
    key-store: classpath:config/tls/keystore.p12
    key-store-password: password
    key-store-type: PKCS12
    key-alias: selfsigned
```

```
    ciphers: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
    enabled-protocols: TLSv1.2
  http2:
    enabled: true
```

Опис параметрів:

- key-store — шлях до файлу сховища ключів (keystore), що містить закритий ключ і сертифікат сервера. У прикладі файл розміщено в директорії classpath:config/tls/.

- key-store-password — пароль доступу до сховища ключів, необхідний для ініціалізації SSL-контексту.

- key-store-type — тип сховища ключів. Формат PKCS12 є одним із рекомендованих стандартів, сумісним з більшістю серверних платформ.

- key-alias — псевдонім (alias) ключа, який використовується для створення SSL-з’єднань. У прикладі вказано самопідписаний сертифікат з alias selfsigned.

- ciphers — список дозволених криптографічних шифрів, які можуть бути використані під час встановлення TLS-з'єднання. Перелік обрано з урахуванням сучасних рекомендацій щодо безпеки.

- enabled-protocols — визначає перелік підтримуваних протоколів. У цьому випадку увімкнено лише TLS 1.2 як безпечну версію протоколу, яка відповідає актуальним вимогам.

Підтримка HTTP/2 Параметр server.http2.enabled: true активує підтримку протоколу HTTP/2, який забезпечує кращу ефективність передавання даних завдяки мультиплексуванню запитів, зменшенню затримок та покращеній продуктивності у порівнянні з HTTP/1.1. Це особливо корисно при взаємодії клієнта з REST API або при завантаженні великої кількості ресурсів.

Файл application-tls.yml є критичним елементом налаштування безпеки у Spring Boot застосунку. Активація TLS/SSL дозволяє гарантувати конфіденційність та цілісність даних під час передавання, знижує ризики атак типу «man-in-the-

middle» і є обов’язковою вимогою при обробці чутливої інформації. Рекомендується зберігати файл keystore.p12 та пароль до нього у захищеному середовищі (наприклад, у системі керування секретами або середовищі CI/CD), а також дотримуватися кращих практик з налаштування SSL у виробничому середовищі.

3.1.4 Конфігурація Spring Security

Конфігурація безпеки визначає основні правила автентифікації, авторизації, обробки винятків, керування сесіями, а також параметри CORS і HTTP Basic Authentication Spring Boot застосунку.

Нижче наведено декларативну конфігурацію за допомогою Spring Security DSL.

```
http
        .csrf()
        .ignoringAntMatchers("/h2-console/**")
        .disable()
        .addFilterBefore(corsFilter, UsernamePasswordAuthenticationFilter.class)
        .exceptionHandling()
        .authenticationEntryPoint(problemSupport)
        .accessDeniedHandler(problemSupport)
.and()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
.and()
        .authorizeRequests()
        .antMatchers(HttpMethod.OPTIONS, "/**").permitAll()
        .antMatchers("/swagger-ui/**").permitAll()
        .antMatchers("/api/asrce/route-state").permitAll()
        .antMatchers("/api/asrce/equipment-state").permitAll()
        .antMatchers("/api/asrce/config").permitAll()
        .antMatchers("/api/user/enterprises/silage/**").permitAll()
        .antMatchers("/test/**").permitAll()
        .antMatchers("/h2-console/**").permitAll()
        .antMatchers("/api/authenticate").permitAll()
        .antMatchers("/api/register").permitAll()
        .antMatchers("/api/activate").permitAll()
        .antMatchers("/api/account/reset-password/init").permitAll()
        .antMatchers("/api/account/reset-password/finish").permitAll()
        .antMatchers("/api/admin/**").hasAuthority(AuthoritiesConstants.ADMIN)
        .antMatchers("/api/**").authenticated()
        .antMatchers("/management/health").permitAll()
        .antMatchers("/management/health/**").permitAll()
        .antMatchers("/management/info").permitAll()
        .antMatchers("/management/prometheus").permitAll()
        .antMatchers("/management/**").hasAuthority(AuthoritiesConstants.ADMIN)
.and()
        .httpBasic()
```

```
.and()
        .apply(securityConfigurerAdapter());
```

Опис функціонального призначення параметрів конфігурації:

- Захист від CSRF-атак:

*.csrf().ignoringAntMatchers("/h2-console/**").disable()* — вимикає CSRF-захист для шляху /h2-console/**, який зазвичай використовується для доступу до H2 Database Console. CSRF-захист використовується для запобігання атакам міжсайтової підміни запитів.

- Конфігурація CORS:

.addFilterBefore(corsFilter, UsernamePasswordAuthenticationFilter.class) — додає фільтр CORS для обробки запити з різних доментів. Цей фільтр застосовується перед стандартним UsernamePasswordAuthenticationFilter у ланцюгу фільтрів.

- Конфігурація обробки винятків:

.exceptionHandling()

.authenticationEntryPoint(problemSupport)

.accessDeniedHandler(problemSupport)

Налаштовує обробку виняткових ситуацій під час автентифікації та при відмові в доступі. problemSupport реалізує відповідну поведінку для обох випадків — неавторизований доступ та недостатні права.

- Керування сесією:

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) — використовується модель керування сесіями без збереження стану, яка відповідає REST-принципам — сервер не зберігає інформації про стан користувача між запитами.

- Правила авторизації:

.authorizeRequests()

.antMatchers(...).permitAll()

*.antMatchers("/api/admin/**").hasAuthority(AuthoritiesConstants.ADMIN)*

*.antMatchers("/api/**").authenticated()*

*.antMatchers("/management/**").hasAuthority(AuthoritiesConstants.ADMIN)*

Визначає, які маршрути (шляхи) доступні без авторизації, які потребують автентифікації, а також які — виключно для користувачів з роллю ADMIN.

Наприклад:

- *permitAll()* — доступ без авторизації (наприклад, до Swagger UI, H2 Console, реєстрації, активації облікового запису).
- *authenticated()* — доступ дозволено лише автентифікованим користувачам.
- *hasAuthority(...)* — доступ тільки для користувачів з відповідними правами.

- HTTP Basic Authentication

.httpBasic() – Увімкнення базової HTTP-автентифікації, яка дозволяє клієнту передавати облікові дані (логін і пароль) у заголовках запиту.

- Застосування додаткового конфігураційного адаптера безпеки

.apply(securityConfigurerAdapter()) – підключення додаткового конфігураційного модуля, який реалізує інтерфейс SecurityConfigurerAdapter та містить інші специфічні налаштування безпеки (наприклад, JWT-фільтри або інтеграцію з OAuth2).

Конфігурація Spring Security відіграє ключову роль у захисті застосунку, забезпечуючи контроль доступу до API, обробку запитів від неавторизованих користувачів, керування сесіями, захист від атак та налаштування механізмів автентифікації. Вона повинна бути адаптована відповідно до вимог до безпеки конкретного середовища використання, зокрема: які саме ролі існують, які маршрути мають бути загальнодоступними, а які — обмеженими.

3.2 Реалізація гнучкої архітектури безпеки та керування доступами і дозволами

3.2.1 Повноваження

Для розмежування доступу груп користувачів на загальному рівні було

реалізовано розділення користувачів на повноваження. Ці серверні ролі включають наступні: ADMIN, DATA_MANAGER, USER і ANONYMOUS.

Головна мета такого поділу — визначення меж доступу до ендпоінтів для виконання певних задач.

- ADMIN: Надає повний доступ до всіх налаштувань системи. Ця роль може призначатися виключно співробітникам компанії.
- DATA_MANAGER: Призначена для працівників, які керують інформацією в межах підприємства або холдингу. Відповідає за обробку, організацію та захист інформації. Призначається користувачам з боку клієнта.
- USER: Призначена для звичайних користувачів системи. Доступний лише інтерфейс користувача.
- ANONYMOUS: Використовується лише для сторінки входу, оскільки система автоматизації підприємства є закритою. Також застосовується для валідації JWT-токенів.

3.2.2 Сутність для керування доступом до компонентів UI

З метою обмеження доступу до елементів користувацького інтерфейсу до схеми БД було додано таблицю ComponentAccess. Призначення цієї сутності — надання прав доступу (CRUD) до визначених компонента веб-інтерфейсу. Компонент у цьому випадку представляє елемент структури HTML DOM. Сама структура компонента являє собою N-арне дерево. Структура сутності компонента наведена на рисунку 3.1.

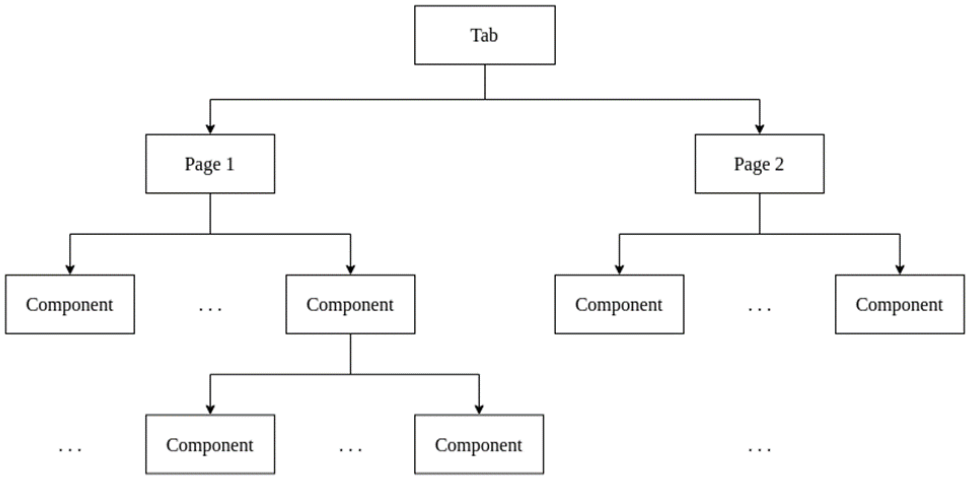

Рисунок 3.1 – Структура сутності компонента

Така ієрархічна структура дозволяє керувати доступом на широкому діапазоні рівнів: від вкладок чи модулів верхнього рівня — до окремих кнопок всередині інших компонентів. У табл. 3.1 наведено перелік полів сутності ComponentAccess в БД та їхній опис.

Таблиця 3.1 – Опис полів сутності ComponentAccess

Поле	Опис
id	унікальний ідентифікатор
name	назва компоненту
key	унікальний ідентифікатор React Компонента
available	увімкнено
enableCreate	дозволити операцію створення
enableRead	дозволити операцію читання
enableUpdate	дозволити операцію оновлення
enableDelete	дозволити операцію видалення
enableByDefault	активний за замовчуванням
allowAutoUpdate	увімкнути функцію автоматичного оновлення дочірніх елементів
paidFeature	платна функція
template	є шаблоном
annotation	короткий опис
childs	посилання на дочірній компонент
parent	посилання на батьківський компонент
permissions	посилання на сутність Permission

Екземпляри сутності ComponentAccess поділяються на типи залежно від значень відповідних полів. Такий логічний поділ дає змогу класифікувати об’єкти за їх призначенням та функціональністю в системі.

Серед типів класифікуються наступні:

– Шаблон для серверної ролі ADMIN без пов’язаного Permission: ComponentAccess позначено як шаблон (isTemplate = true), не прив’язано до жодного підприємства (enterprise == null).

– Шаблон для серверної ролі DATA_MANAGER без пов’язаного Permission: ComponentAccess позначено як шаблон (isTemplate = true), прив’язано до певного

підприємства (enterprise != null).

– Шаблон для серверної ролі USER без прив’язаного Permission: ComponentAccess не є шаблоном (isTemplate = false), прив’язано до певного підприємства (enterprise != null). Такий тип сутності передбачає, що ComponentAccess може бути як батьківським, так і дочірнім — оскільки структура компонентів є деревоподібною.

3.2.3 Сутності керування доступом до REST ендпойнтів

Сутності EndpointAccess та EndpointGroup використовуються для визначення доступу до серверних кінцевих точок (ендпойнтів) та їх логічного групування. Кожен об’єкт EndpointGroup може містити кілька пов’язаних EndpointAccess, при цьому кожен EndpointAccess може бути пов’язаний лише з однією групою.

Ці сутності також мають визначені поля, що задають дозволи на виконання CRUD-операцій, статуси активності, шаблонності тощо. Вони слугують механізмом централізованого контролю доступу до функціональних можливостей бекенд-сервісів (табл. 3.2, 3.3).

Таблиця 3.2 – Опис полів сутності EndpointAccess

Поле	Опис
Id	унікальний ідентифікатор
Name	назва EndpointAccess
Path	шлях до кінцевої точки в межах URL
httpMethod	HTTP-метод запиту (GET, POST тощо)
Available	увімкнено
enableByDefault	увімкнено за замовчуванням
allowAutoUpdate	увімкнути функцію автоматичного оновлення дочірніх елементів
paidFeature	платна функція
Template	є шаблоном
Annotation	короткий опис
Groups	посилання на EndpointGroup

Таблиця 3.3 – Опис полів сутності EndpointGroup

Поле	Опис
Id	унікальний ідентифікатор
name	назва EndpointGroup
available	увімкнено
enableByDefault	увімкнено за замовчуванням
allowAutoUpdate	увімкнути функцію автоматичного оновлення дочірніх елементів
paidFeature	платна функція
template	є шаблоном
annotation	короткий опис
endpoints	посилання на EndpointAccesses в межах групи
permissions	посилання до відповідних Permissions

Логічний поділ екземплярів EndpointAccess і EndpointGroup на шаблони різного призначення виконується за аналогічним підходом, як і для сутності ComponentAccess — на основі значень відповідних полів, зокрема isTemplate, enterprise та permission. Це дозволяє розмежовувати доступи для різних категорій користувачів і ролей у системі.

3.2.4 Агрегація сутностей керування доступами

З метою централізованого керування дозволами та доступами до ендпойнтів та елементів користувацького інтерфейсу було розроблено сутність-агрегатор. Сутності Permission відображають ролі користувачів. Вони являють собою агрегати доступів за моделлю CRUD до компонентів (ComponentAccess) і доступів до ендпойнтів (EndpointGroup та EndpointAccess), оскільки містить посилання на них та на батьківську з дочірніми сутностями, зберігаючи властивості ієрархічної структури-дерева. Така структура стане у нагоді для керування шаблонами ролів та автооновлення дочірніх елементів у разі внесення змін до батьківського елемента.

Після призначення користувачу, такі дозволи забезпечують йому доступ до різних типів даних і функціоналу системи (табл. 3.4).

Таблиця 3.4 – Опис полів сутності

Поле	Опис
id	унікальний ідентифікатор
name	назва Permission
activated	увімкнено
enableByDefault	увімкнено за замовчуванням
allowAutoUpdate	увімкнути функцію автоматичного оновлення дочірніх елементів
paidFeature	платна функція
annotation	короткий опис
childs	посилання на дочірні Permission
componentAccesses	посилання на ComponentAccesses
endpoints	посилання на EndpointGroups
parent	посилання на батьківський Permission
user	посилання на користувача, якому призначено дозвіл

Сутність може бути трьох типів (логічний поділ за наявними полями сутності):

– Шаблон для серверної ролі ADMIN (рівень ADMIN): не має батьківського елемента (parent == null), не прив’язаний до жодного підприємства (enterprise == null), не прив’язаний до жодного користувача (userAttribute == null).

– Шаблон для серверної ролі DATA_MANAGER (рівень DATA_MANAGER): має батьківський елемент (parent != null), прив’язаний до певного підприємства (enterprise != null), не прив’язаний до жодного користувача (userAttribute == null).

– Шаблон для серверної ролі USER (рівень USER): має батьківський елемент (parent != null), прив’язаний до певного підприємства (enterprise != null), прив’язаний до конкретного користувача (userAttribute != null).

Логічний поділ ролей на типи наведено на рисунку 3.2. Схема сутностей керування дозволами наведена на рисунку 3.3. Схема композиції сутностей керування доступом наведена на рисунку 3.4

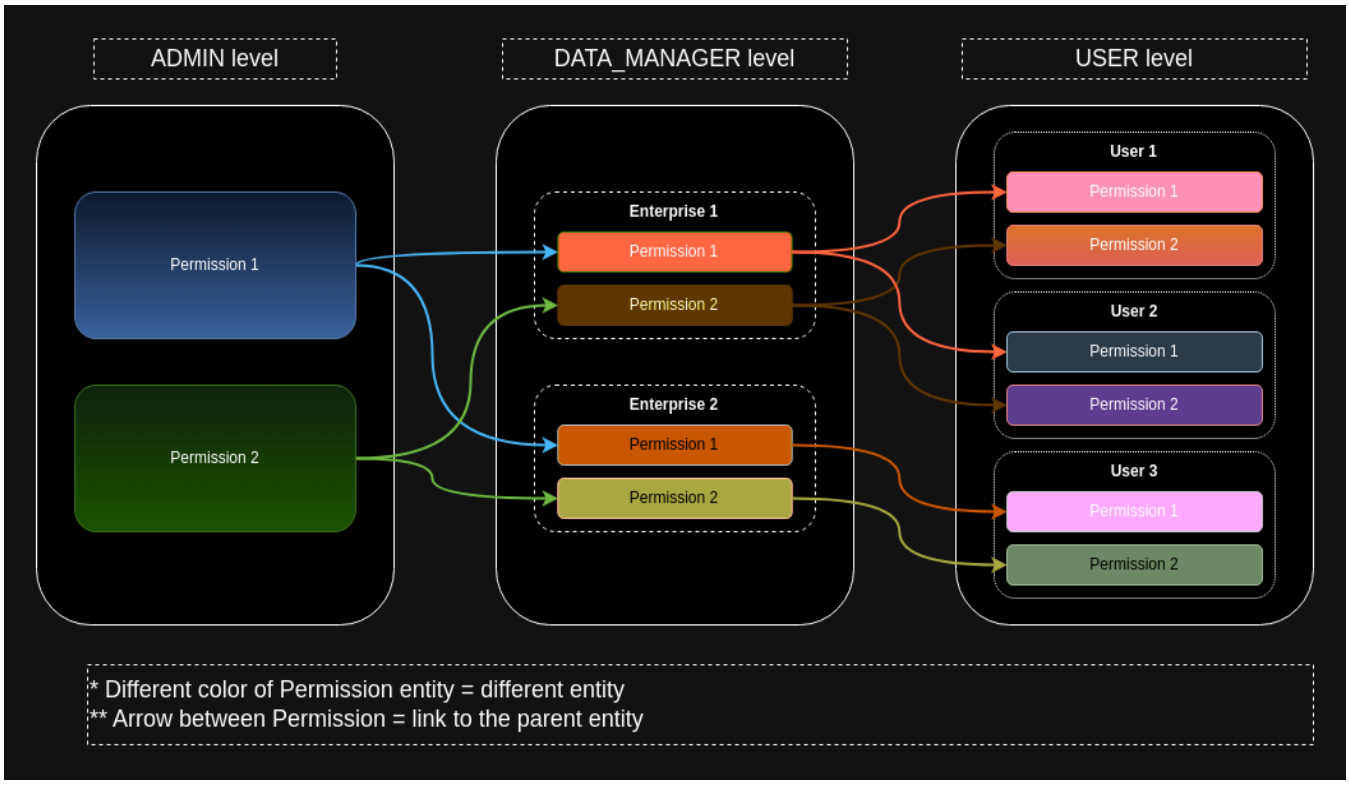

Рисунок 3.2 – Логічний поділ ролей на типи

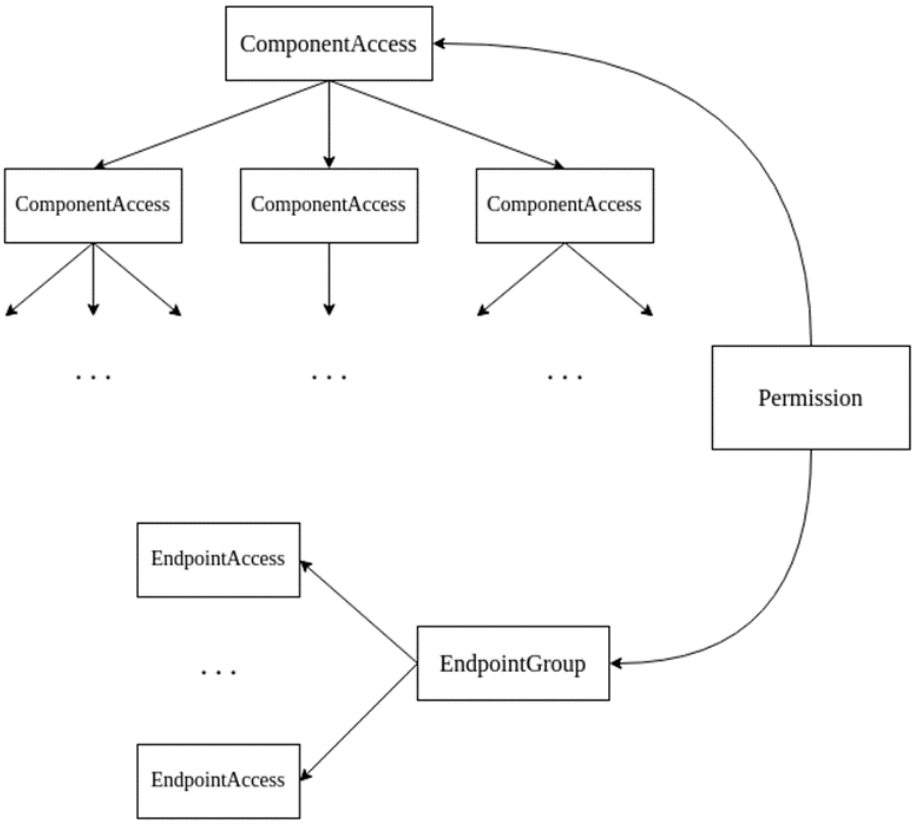

Рисунок 3.3 – Схема сутностей керування дозволами

Альтернативне представлення цієї структури можливе у формі композиції.

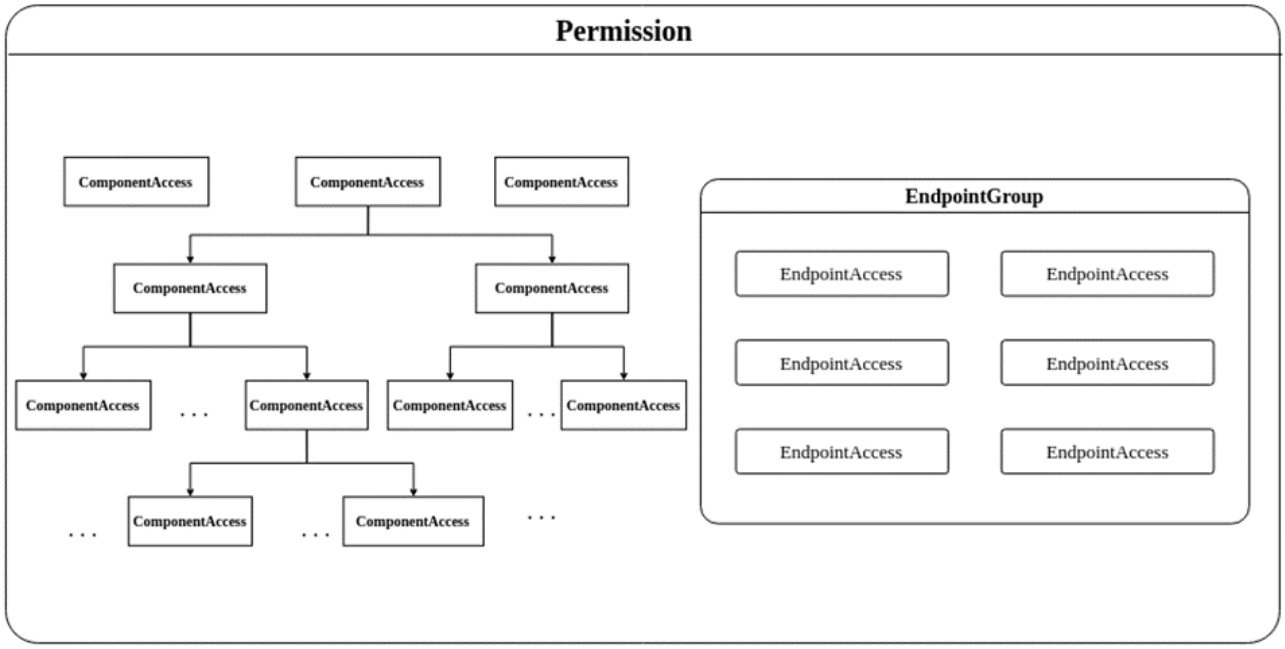

Рисунок 3.4 – Схема композиції сутностей керування доступом

Процес призначення дозволів користувачу

На початковому етапі система надає набір шаблонних дозволів. Алгоритм перетворення шаблону на призначений дозвіл, прив’язаний до користувача, складається з кількох етапів:

1. Виконати поверхневе копіювання екземпляра Permission.

2. Встановити посилання на батьківський шаблон у полі parent нової копії.

3. Виконати глибоке копіювання об’єктів EndpointGroup разом з усіма внутрішніми EndpointAccess, при цьому для всіх встановити значення властивості template = FALSE.

4. Рекурсивно скопіювати дерево пов’язаних ComponentAccess, зберігаючи ієрархію батьківських/дочірніх елементів у копії, та також встановити template = FALSE.

5. Призначити всі створені копії відповідному Permission.

6. Зберегти всі копії до бази даних.

Відповідно сутність Permission пов’язана з користувачем у базі даних відношенням багато до одного, що дозволяє кожному користувачу мати власну конфігурацію прав доступу. На фронтенді можна отримати ці дозволи поточно

користувача для керування доступом до інтерфейсних компонентів і до REST-ендпойнтів.

3.2.5 Реалізація бізнес логіки

Базовий механізм безпеки веб-застосунку реалізовано з використанням фреймворку Spring Security. Його основне призначення полягає в забезпеченні автентифікації та авторизації користувачів, а також у захисті REST API за допомогою JWT-токенів та відповідних фільтрів. Архітектура модуля побудована на основі конфігураційних класів, фільтрів та допоміжних компонентів, що спільно формують єдиний механізм контролю доступу до ресурсів системи.

Центральним елементом у конфігурації безпеки є клас *SecurityConfiguration,* який відповідає за визначення політик безпеки HTTP-запитів. У ньому активовано підтримку безпеки на рівні як веб-запитів, так і окремих методів. Основним компонентом є ланцюг фільтрів безпеки, в якому передбачено використання безсесійної моделі доступу, що є характерною для REST-архітектури. У межах цього ж класу визначаються правила доступу до API: частина маршрутів доступна без автентифікації, інші — лише авторизованим користувачам або користувачам з відповідними ролями.

Аутентифікація користувачів здійснюється на основі JSON Web Token (JWT). За створення та перевірку токенів відповідає окремий компонент *TokenProvider*, який реалізує логіку генерації, підпису, валідації та розбору токена. Підтримується використання як асиметричного алгоритму (RS256), що передбачає застосування пари відкритого та закритого ключів, так і симетричного (HS512), який використовується як резервний. У разі успішної перевірки токена здійснюється побудова об’єкта автентифікації, який передається до контексту безпеки застосунку.

Функцію перехоплення запитів та інтеграції токенів у систему безпеки виконує окремий фільтр JWTFilter. Його завданням є вилучення токена з HTTP-заголовка, перевірка його дійсності та, у разі успіху, встановлення об’єкта автентифікації до поточного контексту. Цей фільтр інтегрується до загального ланцюга безпеки за допомогою відповідного конфігуратора.

Окрему роль у функціонуванні системи відіграє механізм обробки CORS-запитів. Відповідний компонент формує CORS-фільтр на основі конфігураційних параметрів, що дозволяє контролювати походження запитів до API та забезпечити сумісність з фронтенд-частиною системи.

Допоміжні класи *SecurityUtils* та *TokenUtils* модуля безпеки реалізують утилітарну функціональність, зокрема доступ до даних автентифікованого користувача, перевірку ролей, а також автоматичну фіксацію дій користувача у базі даних у рамках механізму аудиту. Крім того, у системі визначено набір констант, що представляють рольові повноваження користувачів, що сприяє стандартизації авторизації на всіх рівнях застосунку.

Усі ці компоненти взаємодіють між собою в рамках єдиної системи контролю доступу, забезпечуючи надійний захист ресурсів веб-застосунку відповідно до сучасних вимог безпеки.

Взаємодія класів безпеки наведено на рисунку 3.5.

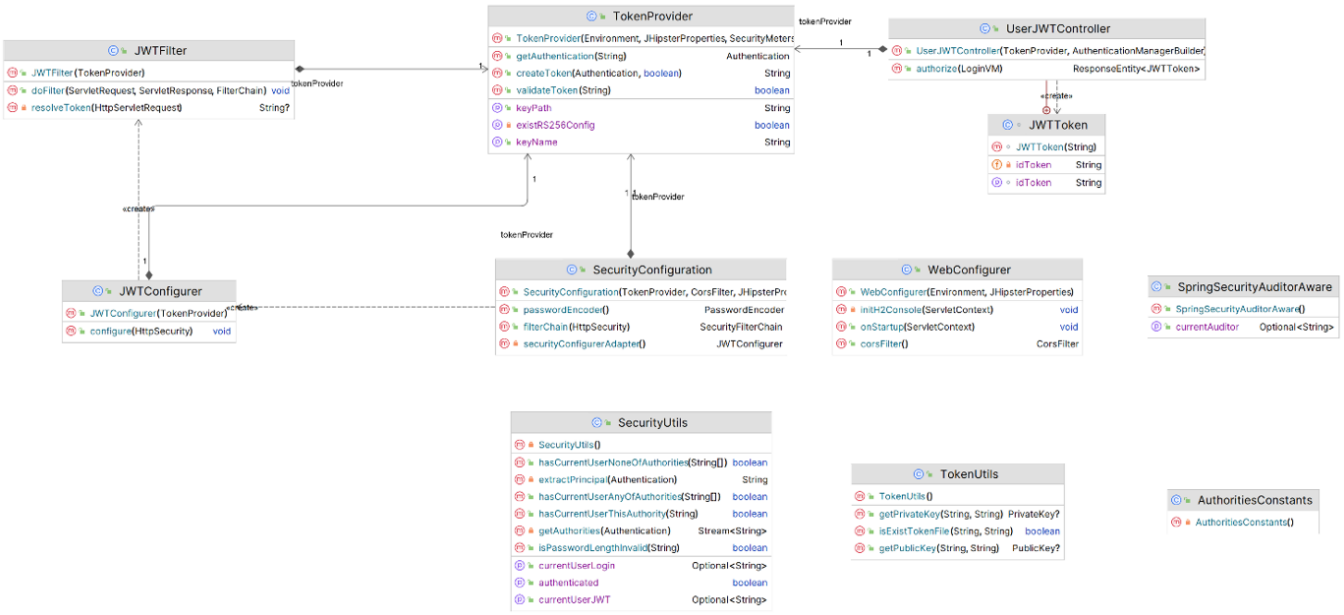

Рисунок 3.5 – Взаємодія класів безпеки

У межах реалізації кастомного механізму керування доступом у системі було розроблено набір спеціалізованих DTO-класів (Data Transfer Object), які слугують

проміжною структурою для передачі і зберігання даних щодо дозволів користувача, компонентів інтерфейсу, а також доступу до REST-ендпоінтів. Ці об'єкти використовуються як на рівні бекенду — для збереження, обробки та перевірки дозволів, — так і при формуванні відповідей до фронтенду для відображення дозволеного функціоналу.

UML-діаграма шару DTO модулю безпеки наведена на рисунку 3.6.

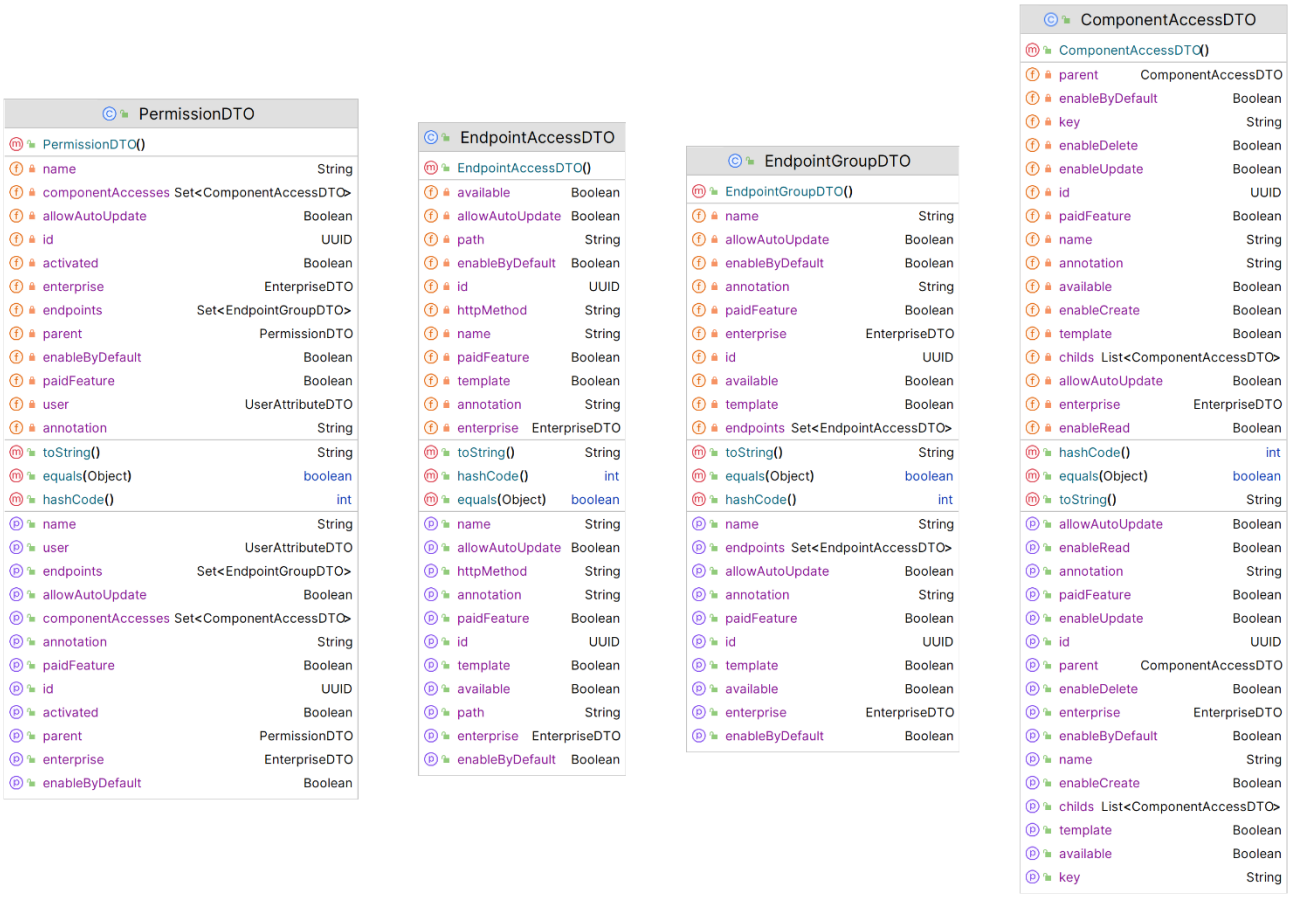

Рисунок 3.6 – UML-діаграма шару DTO модулю безпеки

Бізнес-логіка сервісів модуля безпеки реалізована, використовуючи шаблон проєктування Facade. Спочатку було розроблено набір методів в межах інтерфейсів, а потім реалізовано логіку шляхом імплементації інтерфейсів. UML-діаграма сервісів наведена на рисунку 3.7.

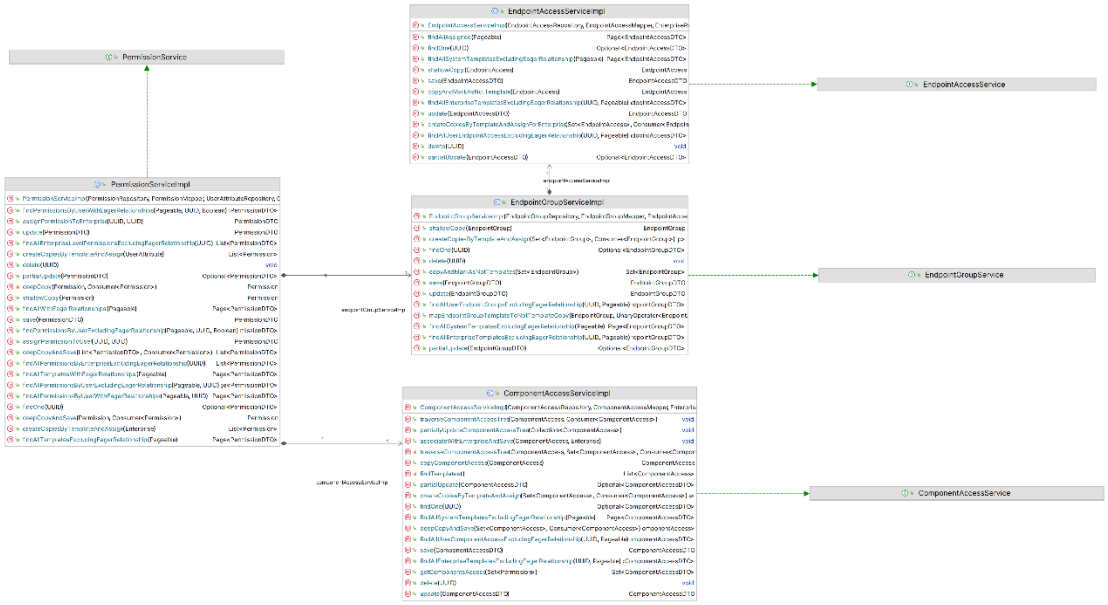

Рисунок 3.7 – UML-діаграма сервісів

Загальний принцип функціонування відбувається таким чином: REST контролери отримують запити та опрацьовують їх, викликаючи інкапсульовані компоненти сервісів. Сервіси у свою чергу реалізують основну бізнес-логіку. Вони містять логіку створення копій шаблонів, глибокого/часткового копіювання, асоціації з підприємствами, користувачами тощо. Сервіси оперують DTO-об'єктами, маперами та взаємодіють з репозиторіями. UML-діаграма взаємодії шарів застосунку на прикладі Permission наведена на рисунку 3.8.

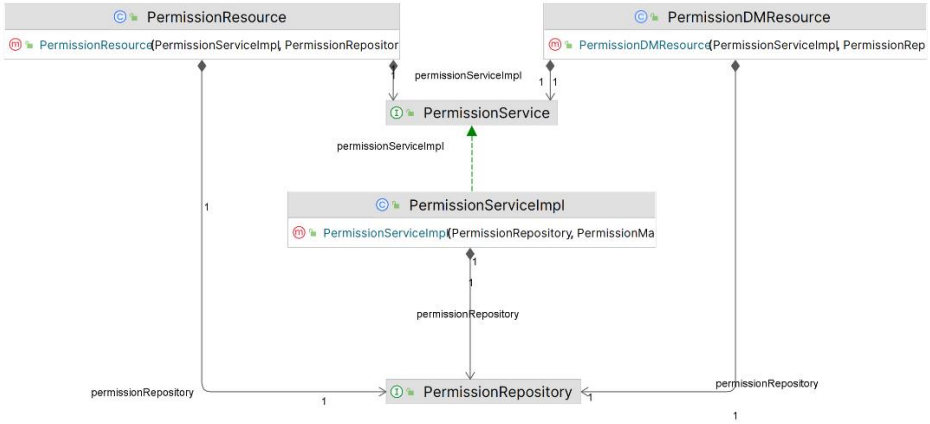

Рисунок 3.8 – UML-діаграма взаємодії шарів застосунку на прикладі Permission

Репозиторії забезпечують доступ та взаємодію з БД, реалізуючи CRUD операції над сутностями предметної області. Також активно застосовуються WithBagRelationships-репозиторії — спеціалізовані компоненти, що дозволяють уникати N+1 проблем при завантаженні зв'язаних колекцій та дочірніх елементів ієрархічних структур. UML-діаграма класів шару репотизорію наведена на рис.3.9.

Рисунок 3.9 – UML-діаграма класів шару репотизорію

3.3 Висновки до розділу 3

У третьому розділі було реалізовано механізм автентифікації та авторизації користувачів. Налаштовано JWT-автентифікацію з використанням приватних ключів, CORS та TLS/SSL для захищеної взаємодії. Конфігурація Spring Security забезпечує чітке розмежування доступу до ресурсів системи.

Окремо реалізовано модель керування дозволами з високою деталізацією. Розроблено сутності для керування доступом до UI-компонентів, REST-ендпоінтів, а також їх агрегацію. Система підтримує шаблони, автопризначення та оновлення прав, що забезпечує гнучке та масштабоване керування доступом.

4 ТЕСТУВАННЯ ТА РОЗГОРТАННЯ ПРОГРАМНОГО МОДУЛЯ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ

4.1 Тестування програмного модуля за допомогою Postman

Перед використанням ресурсів системи автоматизації виробничих процесів користувач повинен пройти етап автентифікації. Для цього здійснюється HTTP POST-запит на ендпойнт контролера, що виконує автентифікацію на основі ім’я користувача та паролю, що містяться в тілі запиту у форматі JSON. Запит на отримання особистого токена представлено на рисунку 4.1.

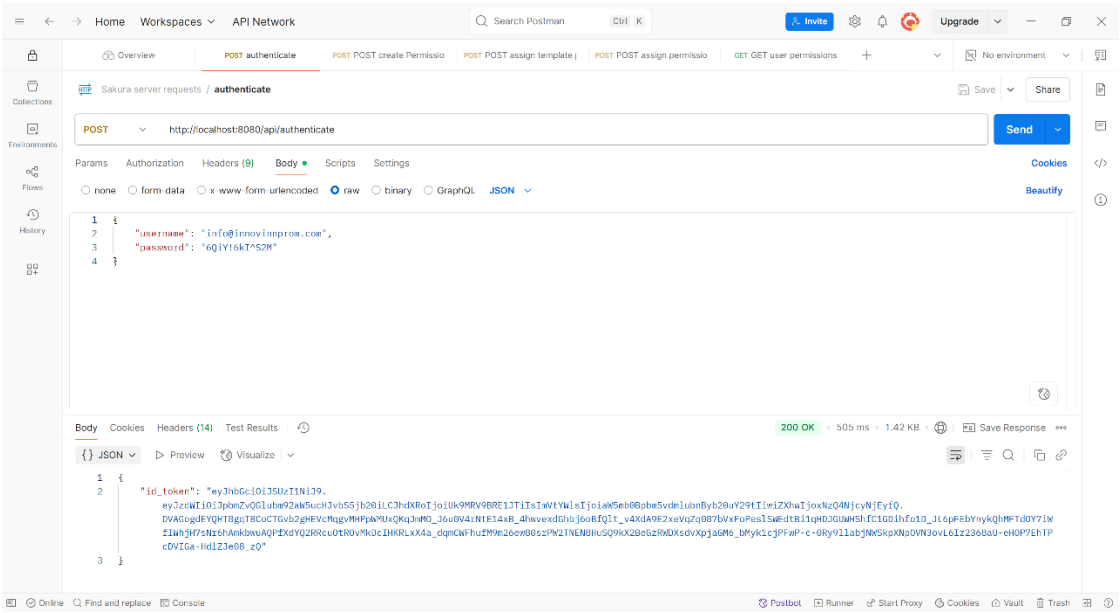

Рисунок 4.1 – Запит на отримання особистого токена

У результаті запиту користувач отримує власний JWT токен. Тепер користувач може використовувати токен для доступу до ресурсів веб-додатку, додаючи до кожного HTTP-запиту заголовок «Authorization» зі значенням «Bearer <особистий токен>».

Далі було протестовано основні методи контролера для оперування ролями користувачів та відповідними дозволами:

```
@PostMapping("/permissions")
public ResponseEntity<PermissionDTO> createPermission(@RequestBody
PermissionDTO permissionDTO) throws URISyntaxException
```

Метод отримує дані, що відображають шаблон сутності ролі, яка буде збережена до бази даних без прив’язки до підприємств та користувачів. Результат запиту на створення шаблону ролі наведено на рисунку 4.2.

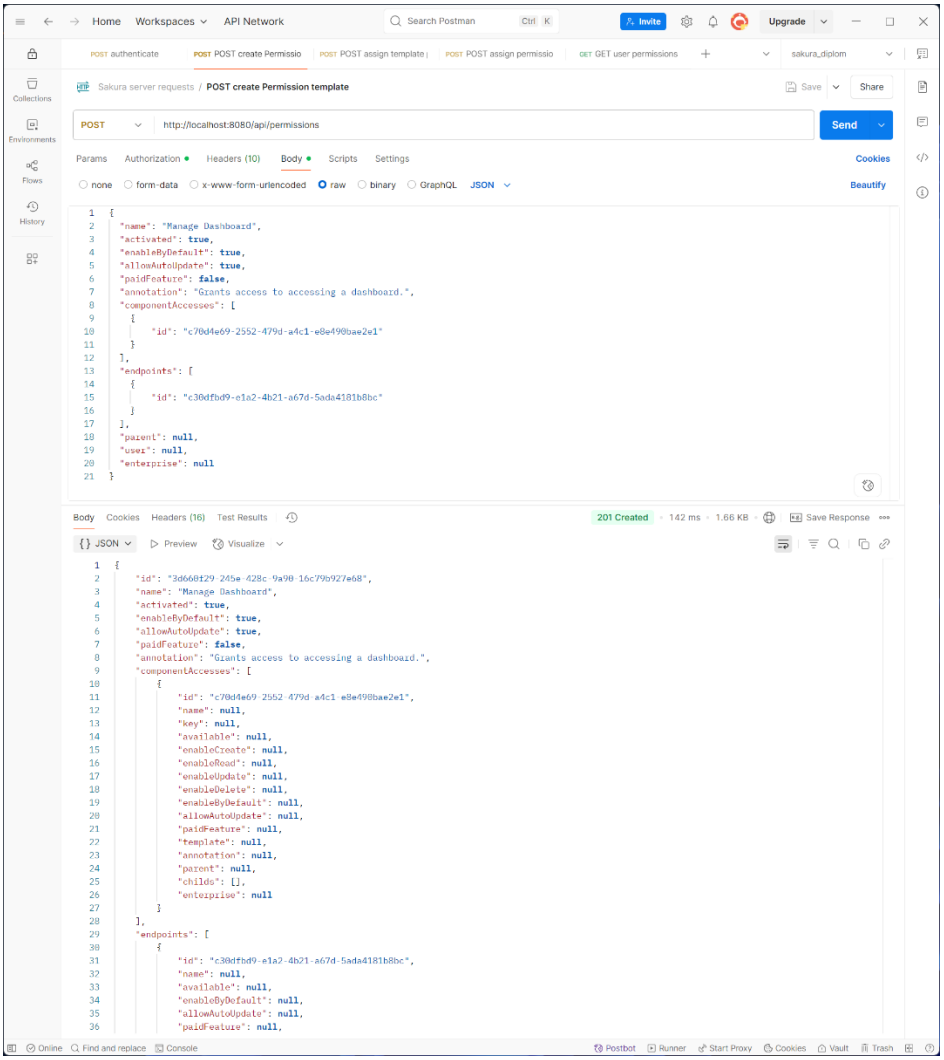

Рисунок 4.2 – Створення шаблону ролі

```
@PostMapping("/permissions/enterprise-template/{enterpriseId}")
public ResponseEntity<PermissionDTO> assignPermissionToEnterprise(
        @PathVariable(value = "enterpriseId") final UUID enterpriseId,
        @RequestParam("permissionId") UUID permissionId
)
```

Метод очікує змінну шляху, що містить унікальний ідентифікатор підприємства в БД та ідентифікатор існуючого шаблону ролі. В результаті створюється копія ролі з прив’язкою з підприємства, що є дочірньою по відношенню до шаблону (рис.4.3).

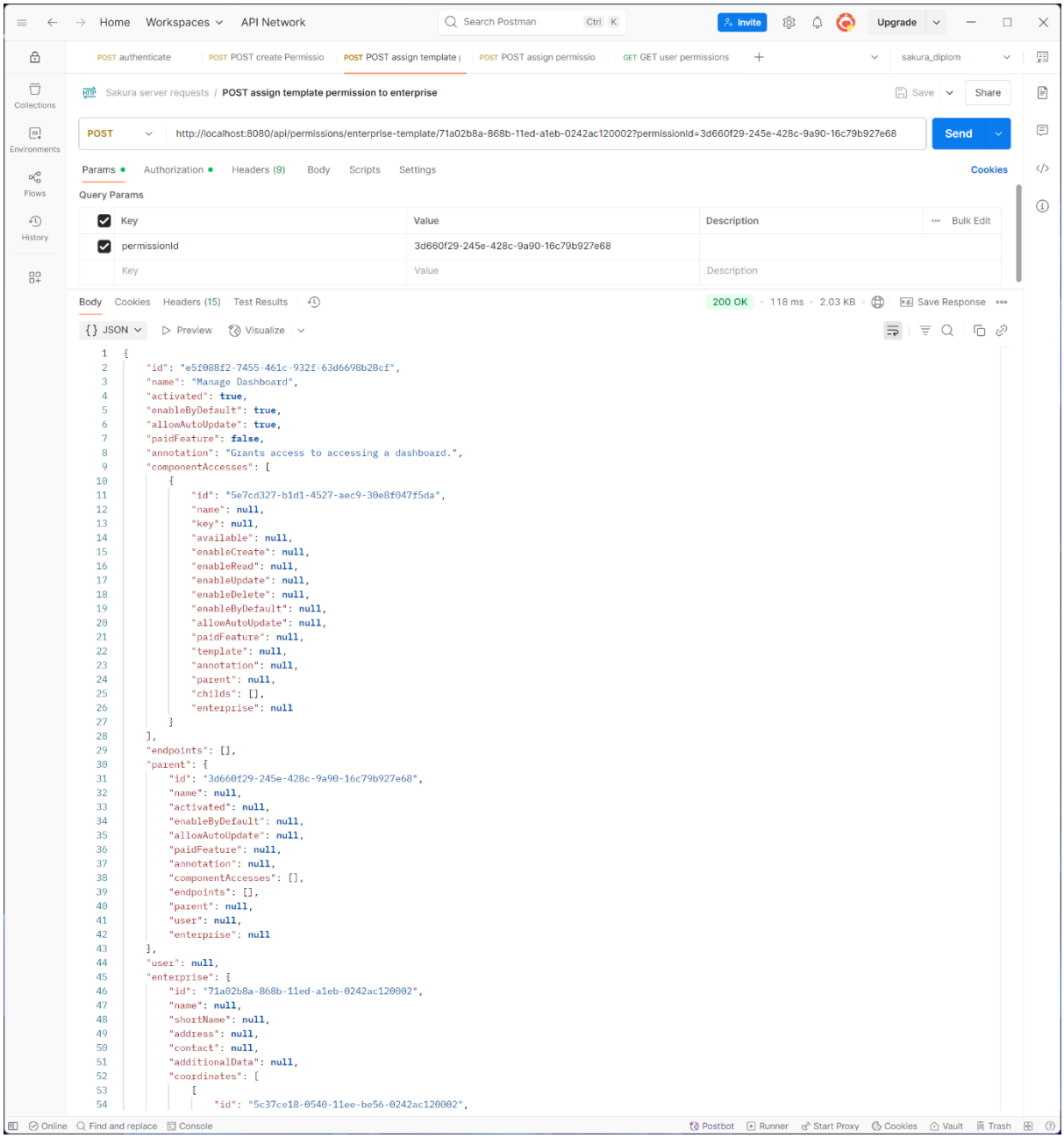

Рисунок 4.3 – Реєстрація ролі в програмному модулі з прив’язкою до підприємства

```
@PostMapping("/permissions/user-template/{userId}")
public ResponseEntity<PermissionDTO> assignPermissionToUser(
        @PathVariable(value = "userId") final UUID userAttributeId,
        @RequestParam("permissionId") UUID permissionId
)
```

Метод отримує ідентифікатор користувача за допомогою змінної шляху та ідентифікатор зареєстрованої ролі (рис 4.4). В результаті здійснюється копіювання ролі та присвоєння відповідному користувачу.

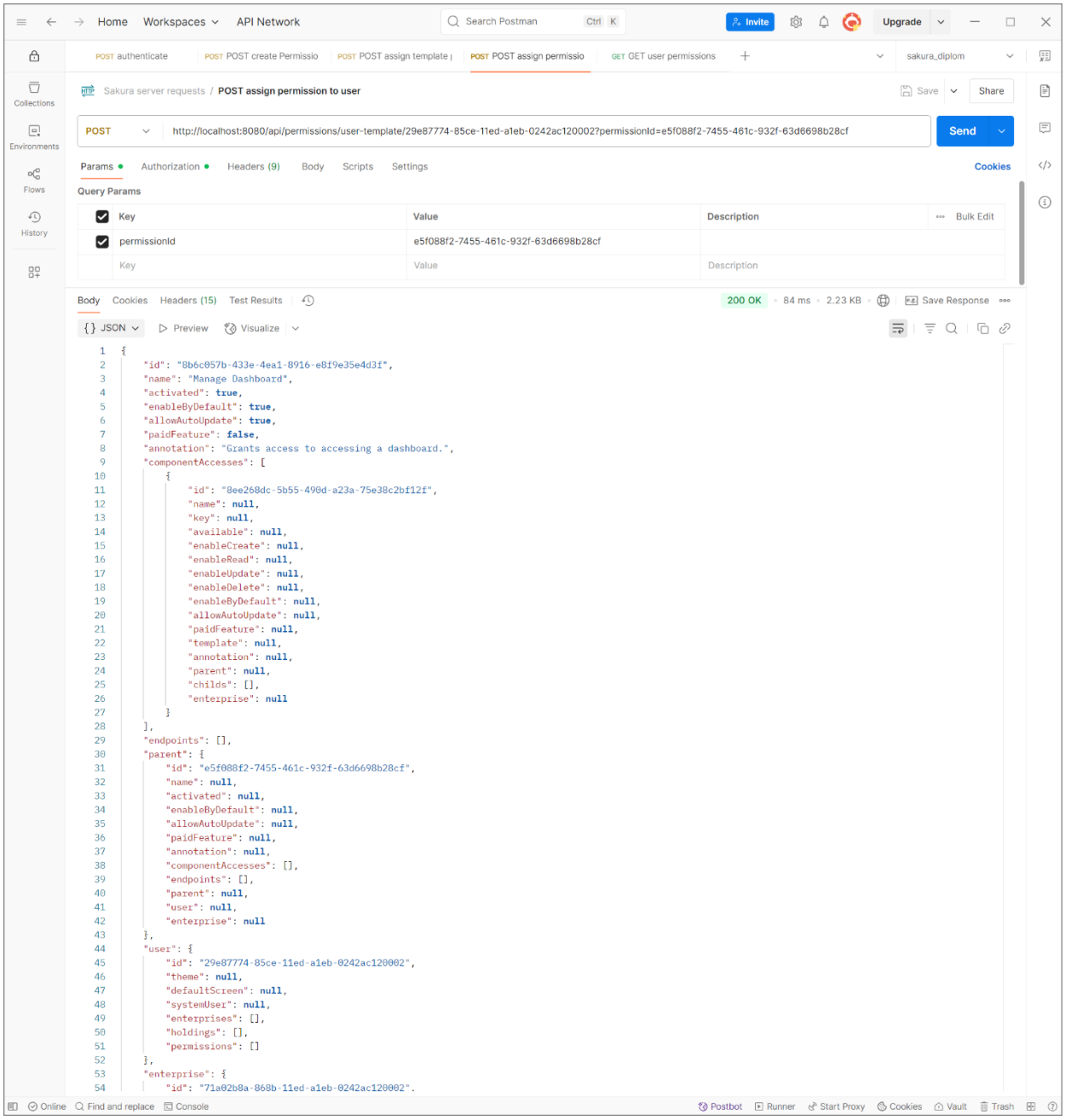

Рисунок 4.4 – Присвоєння ролі користувачу

```
@GetMapping("/permissions/user-template/{userId}")
public ResponseEntity<List<PermissionDTO>> getAllUserPermissions(
        @org.springdoc.api.annotations.ParameterObject Pageable pageable,
        @PathVariable(value = "userId") final UUID userId
)
```

Метод отримує ідентифікатор користувача та повертає всі ролі, присвоєні відповідному користувачу програмного модулю в межах підприємства (рис. 4.5).

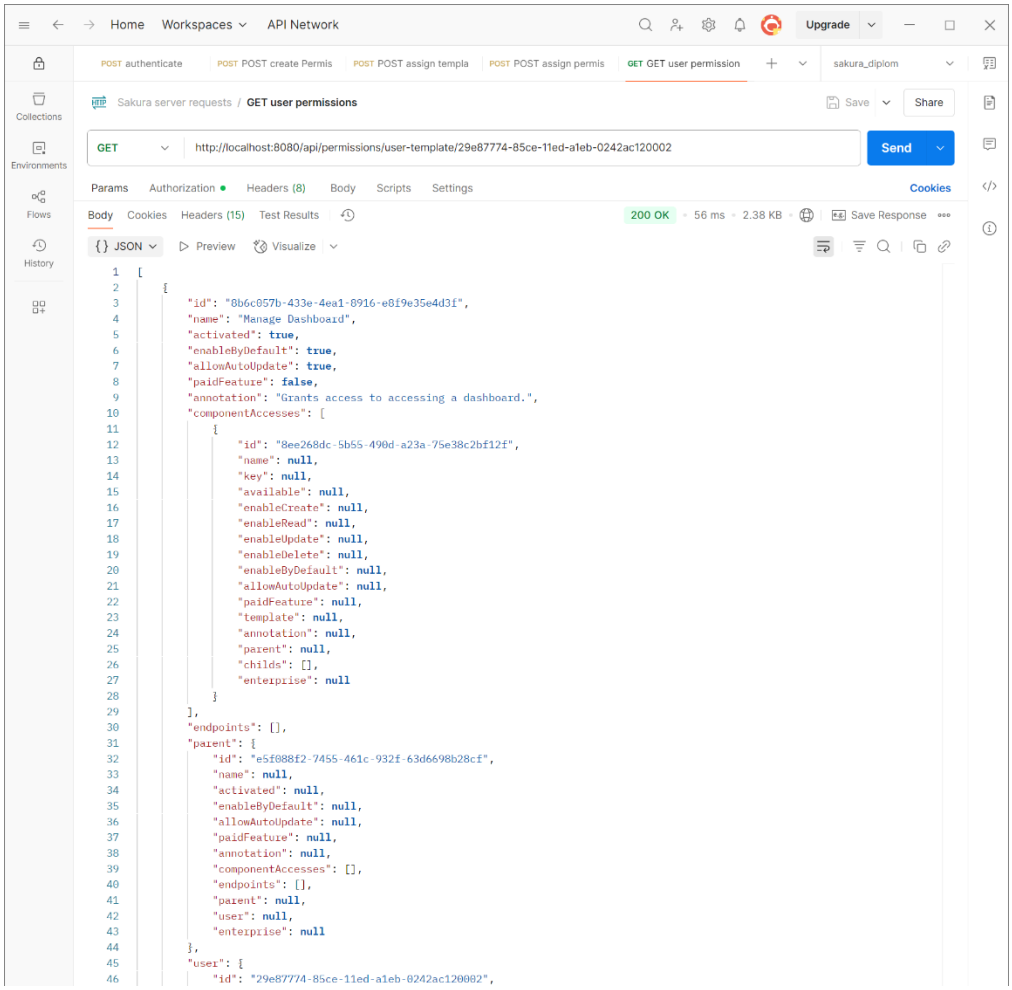

Рисунок 4.5 – Отримання ролей користувача

В результаті тестування було доведено, що програмний модуль працює так, як було спроєктовано і відповідає початковим вимогам.

4.2 Розгортання Spring Boot застосунку в Docker контейнері

Контейнеризація є сучасним підходом до розгортання програмного забезпечення, який дозволяє ізолювати застосунок разом із усіма його залежностями в окремому середовищі — контейнері. Завдяки цьому забезпечується стабільна й відтворювана робота застосунку в різних інфраструктурах, незалежно від особливостей операційної системи чи конфігурації середовища. Одним із найпоширеніших інструментів для контейнеризації є Docker, який дає змогу пакувати застосунки в легкі та портативні контейнери, що легко транспортуються, масштабуються та супроводжуються.

У розгортанні Spring Boot застосунків використання Docker дозволяє не лише

автоматизувати процес побудови і запуску, а й спростити керування конфігураціями, полегшити CI/CD-процеси та уніфікувати середовище виконання. Для цього необхідно створити або використати вже готовий Dockerfile — спеціальний файл, що описує інструкції для побудови контейнера із застосунком [23].

Першим кроком є написання `Dockerfile` — він має бути присутній у складі вихідного коду або надається розробниками окремо. Зазвичай файл розміщується в кореневій директорії проєкту, де зберігається код застосунку та конфігураційні файли. Далі, за наявності встановленого Docker, виконується побудова Docker-образу за допомогою команди `docker build`. Для цього відкривається термінал, виконується перехід до директорії, де знаходиться `Dockerfile`, і вводиться команда:

docker build -t registry.company.com/server:dev .

Параметр `-t` дозволяє призначити тег (ім'я) для створюваного образу. Крапка наприкінці команди вказує на поточний каталог як контекст побудови, тобто місце, звідки Docker буде брати ресурси. У процесі виконання Docker інтерпретує інструкції з Dockerfile та створює образ, який містить усі необхідні залежності та артефакти для запуску Spring Boot застосунку.

Після завершення побудови образу можна переконатися в його наявності за допомогою команди: *docker images*.

Виведений список образів має містити щойно створений образ із відповідним тегом. Далі застосунок запускається у вигляді окремого контейнера за допомогою команди:

docker run -d -p 8080:8080 registry.company.com/server:dev

Ключ `-d` вказує на запуск у фоновому режимі, а параметр `-p 8080:8080` встановлює перенаправлення порту: порт 8080 усередині контейнера зв’язується з портом 8080 хост-машини. У результаті застосунок стає доступним за адресою `http://localhost:8080`.

Таким чином, контейнер з Spring Boot застосунком функціонує як самостійний процес, повністю ізольований від системного середовища, але при цьому доступний для клієнтів через відкритий порт. Це забезпечує однакову

поведінку застосунку в середовищах розробки, тестування та продуктивної експлуатації, значно спрощуючи його розгортання та масштабування.

4.3 Рекомендація застосування методології неперервної інтеграції та розгортання (CI/CD)

Методологія неперервної інтеграції та розгортання (CI/CD) є критично важливою складовою сучасної розробки програмного забезпечення, що дозволяє автоматизувати процеси збирання, тестування, перевірки та доставлення коду до цільового середовища. Неперервна інтеграція (Continuous Integration) забезпечує регулярне об'єднання змін до головної гілки репозиторію з автоматичним виконанням тестів, тоді як неперервне розгортання (Continuous Deployment/Delivery) автоматизує доставку зміненого програмного продукту на сервери або у хмарне середовище. Такий підхід мінімізує людські помилки, пришвидшує цикл випуску нових версій і покращує надійність програмного забезпечення.

Інтеграція CI/CD у проєкт на базі GitLab передбачає використання вбудованих механізмів для конфігурації змінних середовища, налаштування доступу до приватних сховищ (наприклад, GitLab Registry), а також забезпечення безпечного з'єднання з віддаленими серверами через SSH [24].

Для збереження секретних значень, таких як токени доступу, SSH-ключі, паролі або змінні конфігурації, використовуються GitLab Variables. Вони налаштовуються у розділі меню Settings → CI/CD → Variables. У цьому інтерфейсі зручно організовується зберігання як звичайних змінних середовища, так і файлів, наприклад приватних SSH-ключів. Для середовищ stage та production доцільно створити окремі пари SSH-ключів, які будуть використовуватись при деплої.

Для доступу до приватного GitLab Registry застосовується механізм “deploy tokens”. Ці токени генеруються у відповідному розділі репозиторію і дозволяють безпечно виконувати авторизацію без необхідності зберігати основні облікові дані. До змінних середовища додається ім’я користувача (логін токена) і сам токен, які використовуються під час доступу до реєстру контейнерів.

Окрему увагу слід приділити правильному збереженню SSH-ключів.

Приватний ключ, згенерований заздалегідь, має бути повністю скопійований у змінну типу “File”, із обов’язковим додаванням порожнього рядка в кінці. Відсутність порожнього рядка може спричинити помилку формату ключа (`SSH key invalid format`) під час виконання pipeline.

У результаті налаштування було реалізовано структуру CI/CD-завдань (див. рис. 4.6), яка автоматизує збирання Docker-образу, його завантаження до приватного реєстру, а також розгортання застосунку на сервері. Логіка pipeline представлена у вигляді послідовності етапів, що охоплюють перевірку, збірку, публікацію та деплой.

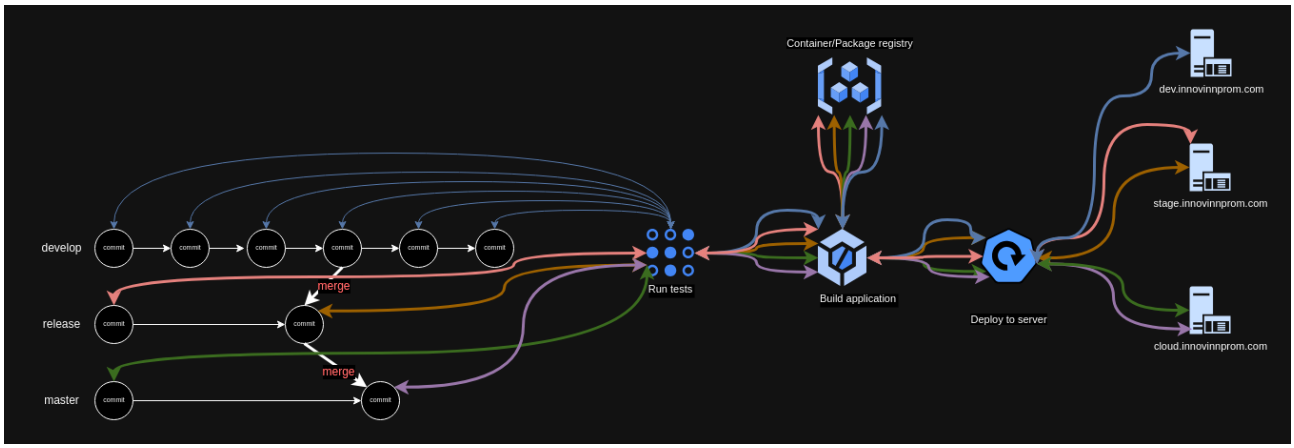

Рисунок 4.6 – Структура CI/CD-процесу для Spring Boot застосунку

Такий підхід дозволяє забезпечити відтворюваність усіх етапів розробки, скоротити час між розробкою та постачанням функціональності користувачам, а також гарантує, що кожна версія застосунку проходить уніфіковану процедуру збірки та перевірки перед публікацією.

4.4 Висновки до розділу 4

У четвертому розділі проведено тестування REST-інтерфейсів та механізмів авторизації з використанням Postman. Отримані результати підтвердили коректну роботу запитів, правильну обробку JWT-токенів.

Також здійснено розгортання застосунку в Docker-контейнері, що забезпечило середовище ізольованого тестування. Застосовано методологію CI/CD, що дозволяє автоматизувати процес розробки.

ВИСНОВКИ

Поставлені перед бакалаврською кваліфікаційною роботою задачі виконано в повному обсязі, а саме:

- здійснено аналіз предметної галузі керування дозволами та доступами;
- спроєктовано програмний модуль керування дозволами та доступами;
- реалізовано програмний модуль керування дозволами та доступами, розробити логіку програмного модуля та здійснити базове налаштування механізмів безпеки.
- виконано тестування програми та провести аналіз отриманих результатів;
- розроблено інструкцію користувача.

У ході виконання бакалаврської роботи було проведено всебічний аналіз предметної галузі керування доступом у автоматизованих системах управління технологічними процесами. У першому розділі розглянуто основні моделі контролю доступу (RBAC, ABAC, DAC, MAC та PBAC) із визначенням їх переваг і недоліків, а також проаналізовано існуючі промислові рішення (ThingsBoard та Thinger IoT Platform) з точки зору гнучкості безпекових механізмів. Оцінка використаних технологій зокрема Spring Security, Spring Security OAuth2 Client та бібліотеки JJWT дозволила виокремити прогалини в підходах до керування правами доступу на рівні окремих UI-компонентів та REST-ендпойнтів.

У другому розділі було обґрунтовано застосування трирівневої архітектури та монолітного розгортання. Обрано набір патернів (MVC, Dependency Inversion, Builder, Facade), які забезпечують чисту організацію коду та гнучкість змін. Детальна оцінка мов програмування, середовища розробки, фреймворків і систем керування проєктом підтвердила перевагу Java зі Spring Boot/Spring Security і Hibernate у поєднанні з IntelliJ IDEA та Maven.

У третьому розділі реалізовано безпековий шар та основні механізми автентифікації і авторизації. Налаштовано безсесійний JWT-механізм з використанням приватних ключів і фільтрів Spring Security, сконфігуровано CORS та TLS/SSL для захищеного каналу, а також деталізовано правила доступу в SecurityFilterChain. Далі побудовано гнучку модель управління дозволами: описано сутності ComponentAccess, EndpointAccess, EndpointGroup і Permission, їх DTO-

шари та бізнес-логіку на рівні контролерів, сервісів і репозиторіїв. Це дозволило реалізувати ієрархічне керування доступом до окремих DOM-компонентів і REST-ендпойнтів із підтримкою шаблонів, автопризначення й автоновлення прав.

У четвертому розділі проведено тестування розробленого програмного модуля за допомогою Postman, що підтвердило коректність реалізованих REST-інтерфейсів та механізмів безпеки. Розгортання Spring Boot застосунку в Docker-контейнері продемонструвало відтворюваність середовища виконання та спрощення доставки образів, а рекомендації з використання CI/CD в GitLab забезпечують автоматизацію збирання, тестування і розгортання, що підвищує ефективність та надійність процесу розробки.

Поставлена мета, а саме розширення функціональних можливостей механізму керування дозволами та доступами була досягнута за рахунок поєднання Spring Security, JWT Token та розробленої реляційної архітектури сутностей. Це дозволило розробити програмний модуль керування дозволами та доступами з гнучким та багаторівневим механізмом контролю дозволів, а також з можливістю керування доступом до REST-ендпойнтів та компонентів користувацького інтерфейсу для систем автоматизації виробничих процесів.

Звіт про виконану роботу було складено відповідно до вимог методичних вказівок, що гарантує його структурованість, логічність викладення матеріалу та відповідність академічним стандартам [25].

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Управління інформаційною безпекою. URL: https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fela.kpi.ua%2Fserver%2Fapi%2Fcore%2Fbitstreams%2Fcd74918a-3d93-4c32-b562-0ca71e4630e7%2Fcontent (дата звернення 21.05.2025).

2. Ferraiolo D. F., Kuhn D. R. Role-Based Access Control // Proceedings of the 15th National Computer Security Conference. – Baltimore, MD, USA, 13–16 October 1992. – P. 554–563. URL: https://csrc.nist.gov/files/pubs/conference/1992/10/13/rolebased-access-controls/final/docs/ferraiolo-kuhn-92.pdf (дата звернення: 21.05.2025).

3. What are the different types of access control systems? – SailPoint. URL: https://www.sailpoint.com/identity-library/what-are-the-different-types-of-access-control-systems (дата звернення 21.05.2025).

4. Реалізація механізму автентифікації та авторизації REST API на основі JWT Token та Spring Security /В.Л. Поліщук, І.В. Богач // Матеріали LIV науково-технічної конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ-2025). Збірник доповідей [Електронний ресурс], Вінниця : ВНТУ, 2025. Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fksa/all-fksa-2025/paper/view/23707 [Тези 2]

5. Реалізація авторизації HTTP-запитів до REST-ендпойнтів за допомогою Spring Security DSL /В.Л.Поліщук, І.В.Богач // Матеріали міжнародної науково-практичної Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: дослідження, проблеми, перспективи (МН-2025)». Збірник доповідей [Електронний ресурс], Вінниця: ВНТУ, 2025. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25415

6. Access Control Models – Escape.tech. URL: https://escape.tech/blog/access-control-models (дата звернення 21.05.2025).

7. Access Control Models – Twingate. URL: https://www.twingate.com/blog/other/access-control-models (дата звернення 21.05.2025).

8. Attribute-Based Access Control (ABAC). National Institute of Standards and Technology (NIST). URL: https://csrc.nist.gov/Projects/Attribute-Based-Access-Control (дата звернення: 21.05.2025).

9. Policy-Based Access Control – Ping Identity, NextLabs. URL: https://www.pingidentity.com/en/resources/blog/post/policy-based-access-control.html; https://www.nextlabs.com/blogs/what-is-policy-based-access-control-pbac/ (дата звернення 21.05.2025).

10. Role-Based Access Control в ThingsBoard. URL: https://thingsboard.io/docs/pe/user-guide/rbac/ (дата звернення 21.05.2025).

11. Thinger.io – Devices Administration. URL: https://github.com/thinger-io/Docs/blob/main/features/devices-administration.md (дата звернення 21.05.2025).

12. Thinger.io – Access Tokens. URL: https://docs.thinger.io/features/access-tokens (дата звернення 21.05.2025).

13. Spring Security – OAuth 2.0 Client (Reactive) Documentation. URL: https://docs.spring.io/spring-security/reference/reactive/oauth2/client/index.html (дата звернення 21.05.2025).

14. Client-server architecture – Britannica. URL: https://www.britannica.com/technology/client-server-architecture (дата звернення 21.05.2025).

15. Client-server architecture – Britannica. URL: https://www.britannica.com/technology/client-server-architecture (дата звернення 21.05.2025).

16. Microservices vs. Monolith – Atlassian. URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (дата звернення 21.05.2025).

17. GraphQL vs REST – AIMultiple. URL: https://research.aimultiple.com/graphql-vs-rest/ (дата звернення 21.05.2025).

18. Oracle Java Technologies – Oracle. URL: https://www.oracle.com/java/ (дата звернення 21.05.2025).

19. IntelliJ IDEA – Офіційний сайт JetBrains. URL: https://www.jetbrains.com/idea/ (дата звернення 21.05.2025).

20. Spring Framework Documentation. URL: https://docs.spring.io/spring-framework/ (дата звернення 21.05.2025).

21. Hibernate ORM Documentation. URL: https://hibernate.org/orm/documentation/ (дата звернення 21.05.2025).

22. Apache Maven – Maven Features. URL: https://maven.apache.org/maven-features.html (дата звернення 21.05.2025).

23. Dockerfile reference – Docker Documentation. URL: https://docs.docker.com/reference/dockerfile/ (дата звернення 21.05.2025).

24. Build your application – GitLab Documentation. URL: https://docs.gitlab.com/topics/build_your_application/ (дата звернення 21.05.2025).

25. Методичні вказівки до виконання бакалаврських кваліфікаційних робіт для студентів спеціальності 122 «Комп’ютерні науки» [Електронний ресурс] / уклад.: А. А. Яровий, О. В. Сілагін, І. В. Богач. – Вінниця : ВНТУ, 2023. – (PDF, 54 с.).

ДОДАТКИ

Додаток А (обов'язковий)

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Програмний модуль керування дозволами та доступами для систем автоматизації виробничих процесів

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

Підрозділ кафедра комп'ютерних наук
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 4,58 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

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

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

Яровий А.А., зав. каф. КН _____ _____
(прізвище, ініціали, посада) (підпис)

Колесницький О.К., проф. каф КН _____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Озеранський В.С.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Богач І.В.
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Поліщук В.Л.
(підпис) (прізвище, ініціали)

Додаток Б

(обов'язковий)

Лістинг програмного забезпечення

Permission.java

```
package com.prodautomation.domain;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import javax.persistence.*;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import java.util.UUID;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity
@Table(name = "permission")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@SuppressWarnings("common-java:DuplicatedBlocks")
public class Permission implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    @Column(name = "id")
    private UUID id;

    @Column(name = "name")
    private String name;

    @Column(name = "activated")
    private Boolean activated;

    @Column(name = "enable_by_default")
    private Boolean enableByDefault;

    @Column(name = "allow_auto_update")
    private Boolean allowAutoUpdate;

    @Column(name = "paid_feature")
    private Boolean paidFeature;
```

```
    @Column(name = "annotation")
    private String annotation;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "childs", "componentAccesses", "endpoints",
"parent", "user", "enterprise" }, allowSetters = true)
    private Set<Permission> childs = new HashSet<>();

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "rel_permission__component_access",
        joinColumns = @JoinColumn(name = "permission_id"),
        inverseJoinColumns = @JoinColumn(name = "component_access_id")
    )
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "enterprise", "permissions" }, allowSetters = true)
    private Set<ComponentAccess> componentAccesses = new HashSet<>();

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
        name = "rel_permission__endpoints",
        joinColumns = @JoinColumn(name = "permission_id"),
        inverseJoinColumns = @JoinColumn(name = "endpoints_id")
    )
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "enterprise", "permissions" }, allowSetters = true)
    private Set<EndpointGroup> endpoints = new HashSet<>();

    @ManyToOne(fetch = FetchType.LAZY)
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "childs", "componentAccesses", "endpoints",
"parent", "user", "enterprise" }, allowSetters = true)
    private Permission parent;

    @ManyToOne(fetch = FetchType.LAZY)
    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @JsonIgnoreProperties(value = { "premissions", "enterprises", "holdings" },
allowSetters = true)
    private UserAttribute user;

    @ManyToOne
    @JsonIgnoreProperties(
        value = {
            "coordinates",
            "equipments",
            "asrces",
```

```
        "energyIotGateways",
        "contracts",
        "customOptions",
        "permissions",
        "componentsAccesses",
        "endpointGroups",
        "endpointsAccesses",
        "grainCropsTypes",
        "grainCropsTypeResolvers",
        "equipmentTypes",
        "equipmentTypeResolvers",
        "routeOperations",
        "routeOperationResolvers",
        "deliveryOperationTypes",
        "deliveryProviders",
        "holding",
        "userAttributes",
    },
    allowSetters = true
)
private Enterprise enterprise;

public Set<Permission> getChilds() {
    return this.childs;
}

public void setChilds(Set<Permission> permissions) {
    if (this.childs != null) {
        this.childs.forEach(i -> i.setParent(null));
    }
    if (permissions != null) {
        permissions.forEach(i -> i.setParent(this));
    }
    this.childs = permissions;
}

public Permission childs(Set<Permission> permissions) {
    this.setChilds(permissions);
    return this;
}

public Permission addChilds(Permission permission) {
    this.childs.add(permission);
    permission.setParent(this);
    return this;
}
```

```
    public Permission removeChilds(Permission permission) {
        this.childs.remove(permission);
        permission.setParent(null);
        return this;
    }

    // інші геттери, сеттери, білдер методи; equals(), hashCode() та toString().
```

PermissionRepositoryWithBagRelationships.java

```
package com.prodautomation.repository;

import java.util.List;
import java.util.Optional;
import java.util.UUID;
import com.prodautomation.domain.Permission;
import org.springframework.data.domain.Page;

public interface PermissionRepositoryWithBagRelationships {
    Optional<Permission> fetchBagRelationships(Optional<Permission> permission);

    List<Permission> fetchBagRelationships(List<Permission> permissions);

    Page<Permission> fetchBagRelationships(Page<Permission> permissions);
}
```

PermissionRepository

```
package com.prodautomation.repository;

import com.prodautomation.domain.Permission;
import com.prodautomation.domain.UserAttribute;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;
import java.util.UUID;


@Repository
public interface PermissionRepository extends
```

```
PermissionRepositoryWithBagRelationships, JpaRepository<Permission, UUID> {

    @Query("select permissions from Permission permissions left join
permissions.enterprise enterprise where enterprise.id = :enterpriseId and
permissions.activated = true and permissions.name not in (select permissions2.name
from Permission permissions2 left join permissions.user user2 where user2.id
= :userUUID)")
    Page<Permission>
findAvailablePermissionsForUserExcludingEagerRelationship(@Param("enterpriseId")
UUID enterpriseId, @Param("userUUID") UUID userUUID, Pageable pageable);

    default Page<Permission>
findAvailablePermissionsForUserWithEagerRelationship(UUID enterpriseId, UUID
userUUID, Pageable pageable) {
        return
this.fetchBagRelationships(this.findAvailablePermissionsForUserExcludingEagerRelati
onship(enterpriseId, userUUID, pageable));
    }

    @Query("select permissions from Permission permissions where permissions.parent
is null and permissions.enterprise is null")
    Page<Permission> findAllTemplatesExcludingEagerRelationship(Pageable
pageable);

    @Query("select permissions from Permission permissions where permissions.parent
is not null and permissions.enterprise is null")
    Page<Permission> findAllAssignedExcludingEagerRelationship(Pageable pageable);

    default Page<Permission> findAllTemplatesWithEagerRelationships(Pageable
pageable) {
        return
this.fetchBagRelationships(this.findAllTemplatesExcludingEagerRelationship(pageable
));
    }

    default Page<Permission> findAllAssignedWithEagerRelationships(Pageable
pageable) {
        return
this.fetchBagRelationships(this.findAllAssignedExcludingEagerRelationship(pageable)
);
    }

    @Query("select permissions from Permission permissions where permissions.user
= :user and permissions.activated = :activatedOnly")
    Page<Permission>
findPermissionsByUserExcludingEagerRelationship(@Param("user") UserAttribute
user, @Param("activatedOnly") Boolean activatedOnly, Pageable pageable);
```

```
    default Page<Permission>
findPermissionsByUserWithEagerRelationships(UserAttribute user, Boolean
activatedOnly, Pageable pageable) {
        return
this.fetchBagRelationships(this.findPermissionsByUserExcludingEagerRelationship(us
er, activatedOnly, pageable));
    }

    @Query("select permissions from Permission permissions where permissions.user
= :user")
    Page<Permission> findAllPermissionsByUser(@Param("user") UserAttribute user,
Pageable pageable);

    default Page<Permission>
findAllPermissionsByUserWithEagerRelationships(UserAttribute user, Pageable
pageable) {
        return this.fetchBagRelationships(this.findAllPermissionsByUser(user, pageable));
    }

    default Optional<Permission> findOneWithEagerRelationships(UUID id) {
        return this.fetchBagRelationships(this.findById(id));
    }

    default List<Permission> findAllWithEagerRelationships() {
        return this.fetchBagRelationships(this.findAll());
    }

    default Page<Permission> findAllWithEagerRelationships(Pageable pageable) {
        return this.fetchBagRelationships(this.findAll(pageable));
    }

    @Query("select permissions from Permission permissions where
permissions.enterprise.id = :enterpriseId")
    List<Permission>
findAllPermissionsByEnterpriseExcludingEagerRelationship(@Param("enterpriseId")
UUID enterpriseId);

    @Query("select permissions from Permission permissions where permissions.parent
is not null and permissions.user is null and permissions.enterprise.id = :enterpriseId")
    List<Permission>
findAllEnterpriseTemplatePermissionsExcludingEagerRelationship(@Param("enterpris
eId") UUID enterpriseId);
}
```

PermissionRepositoryWithBagRelationshipsImpl.java

```
package com.prodautomation.repository;

import java.util.*;
import java.util.stream.Collectors;

import com.prodautomation.domain.Permission;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

public class PermissionRepositoryWithBagRelationshipsImpl implements
PermissionRepositoryWithBagRelationships {

   private final ComponentAccessRepository componentAccessRepository;

   @PersistenceContext
   private EntityManager entityManager;

   public
PermissionRepositoryWithBagRelationshipsImpl(@Qualifier("componentAccessRepos
itory") ComponentAccessRepository componentAccessRepository) {
      this.componentAccessRepository = componentAccessRepository;
   }

   @Override
   public Optional<Permission> fetchBagRelationships(Optional<Permission>
permission) {
      return permission.map(this::fetchComponentAccessesAndEndpoints);
   }

   @Override
   public Page<Permission> fetchBagRelationships(Page<Permission> permissions) {
      return new PageImpl<>(fetchBagRelationships(permissions.getContent()),
permissions.getPageable(), permissions.getTotalElements());
   }

   @Override
   public List<Permission> fetchBagRelationships(List<Permission> permissions) {
      return
Optional.of(permissions).map(this::fetchComponentAccessesAndEndpoints).orElse(Col
lections.emptyList());
   }
```

```
    List<Permission> fetchComponentAccessesAndEndpoints(List<Permission>
permissions) {
        return
permissions.stream().map(this::fetchComponentAccessesAndEndpoints).collect(Collect
ors.toList());
    }

    Permission fetchComponentAccessesAndEndpoints(Permission permission) {
        Permission result = entityManager
                .createQuery(
                        "select permission from Permission permission left join fetch
permission.componentAccesses left join fetch permission.endpoints endpointGroups
left join fetch endpointGroups.endpoints where permission.id = :id",
                        Permission.class
                )
                .setParameter("id", permission.getId())
                .getSingleResult();
componentAccessRepository.fetchBagRelationships(result.getComponentAccesses());

        return result;
    }
}
```

PermissionService.java

```
package com.prodautomation.service;

import com.prodautomation.domain.Enterprise;
import com.prodautomation.domain.Permission;
import com.prodautomation.domain.UserAttribute;
import com.prodautomation.service.dto.PermissionDTO;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;

import java.util.List;
import java.util.Optional;
import java.util.UUID;
import java.util.function.Consumer;

public interface PermissionService {
    PermissionDTO save(PermissionDTO permissionDTO);
    PermissionDTO update(PermissionDTO permissionDTO);
    Optional<PermissionDTO> partialUpdate(PermissionDTO permissionDTO);
    Page<PermissionDTO> findAllWithEagerRelationships(Pageable pageable);
    Page<PermissionDTO> findAllTemplatesExcludingEagerRelationship(Pageable
```

```
pageable);
   Page<PermissionDTO> findAllTemplatesWithEagerRelationships(Pageable
pageable);
   Page<PermissionDTO> findAllPermissionsByUserWithEagerRelationships(Pageable
pageable, UUID id);
   Page<PermissionDTO>
findAllPermissionsByUserExcludingEagerRelationship(Pageable pageable, UUID id);
   Page<PermissionDTO>
findPermissionsByUserExcludingEagerRelationship(Pageable pageable, UUID
userAttributeId, Boolean activatedOnly);
   Page<PermissionDTO> findPermissionsByUserWithEagerRelationships(Pageable
pageable, UUID userAttributeId, Boolean activatedOnly);
   List<PermissionDTO>
findAllPermissionsByEnterpriseExcludingEagerRelationship(UUID enterpriseId);
   List<PermissionDTO>
findAllEnterpriseLevelPermissionsExcludingEagerRelationship(UUID enterpriseId);
   Optional<PermissionDTO> findOne(UUID id);
   void delete(UUID id);
   List<Permission> createCopiesByTemplateAndAssign(Enterprise enterprise);
   List<Permission> createCopiesByTemplateAndAssign(UserAttribute user);
   List<PermissionDTO> deepCopyAndSave(List<PermissionDTO> originalDTOs,
Consumer<Permission> additionalProcessing);
   Permission deepCopyAndSave(Permission original, Consumer<Permission>
additionalProcessing);
   Permission shallowCopy(Permission original);
   PermissionDTO assignPermissionToEnterprise(UUID permissionId, UUID
enterpriseId);
   PermissionDTO assignPermissionToUser(UUID permissionId, UUID
userAttributeId);
}
```

PermissionServiceImpl.java

```
package com.prodautomation.service;

import com.prodautomation.domain.*;
import com.prodautomation.repository.*;
import com.prodautomation.service.dto.PermissionDTO;
import com.prodautomation.service.mapper.PermissionMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
import java.util.*;
import java.util.function.Consumer;
import java.util.stream.Collectors;

@Service
@Transactional
public class PermissionServiceImpl implements PermissionService {

    private final Logger log = LoggerFactory.getLogger(PermissionServiceImpl.class);

    public static final Boolean ACTIVE_PERMISSION_ONLY = true;
    public static final Boolean INACTIVE_PERMISSION_ONLY = false;

    private final PermissionRepository permissionRepository;

    private final PermissionMapper permissionMapper;

    private final UserAttributeRepository userAttributeRepository;

    private final ComponentAccessServiceImpl componentAccessServiceImpl;

    private final EndpointGroupServiceImpl endpointGroupServiceImpl;

    private final EnterpriseRepository enterpriseRepository;


    public PermissionServiceImpl(
            PermissionRepository permissionRepository,
            PermissionMapper permissionMapper,
            UserAttributeRepository userAttributeRepository,
            ComponentAccessServiceImpl componentAccessServiceImpl,
            EndpointGroupServiceImpl endpointGroupServiceImpl,
            EnterpriseRepository enterpriseRepository) {
        this.permissionRepository = permissionRepository;
        this.permissionMapper = permissionMapper;
        this.userAttributeRepository = userAttributeRepository;
        this.componentAccessServiceImpl = componentAccessServiceImpl;
        this.endpointGroupServiceImpl = endpointGroupServiceImpl;
        this.enterpriseRepository = enterpriseRepository;
    }

    public PermissionDTO save(PermissionDTO permissionDTO) {
        log.debug("Request to save Permission : {}", permissionDTO);
        Permission permission = permissionMapper.toEntity(permissionDTO);
        permission = permissionRepository.save(permission);
        return permissionMapper.toDto(permission);
    }
```

```
    public PermissionDTO update(PermissionDTO permissionDTO) {
        log.debug("Request to update Permission : {}", permissionDTO);
        Permission permission = permissionMapper.toEntity(permissionDTO);
        permission = permissionRepository.save(permission);
        return permissionMapper.toDto(permission);
    }

    public Optional<PermissionDTO> partialUpdate(PermissionDTO permissionDTO) {
        log.debug("Request to partially update Permission : {}", permissionDTO);

        return permissionRepository
            .findById(permissionDTO.getId())
            .map(existingPermission -> {

componentAccessServiceImpl.partiallyUpdateComponentAccessTree(existingPermissio
n.getComponentAccesses());
                //TODO: move mapping inner endpoints from mapper as well
                permissionMapper.partialUpdate(existingPermission, permissionDTO);

                return existingPermission;
            })
            .map(permissionRepository::save)
            .map(permissionMapper::toDto);
    }

    public Page<PermissionDTO> findAllWithEagerRelationships(Pageable pageable) {
        return
permissionRepository.findAllWithEagerRelationships(pageable).map(PermissionMapp
er::toPermissionDto);
    }

    @Transactional(readOnly = true)
    public Page<PermissionDTO>
findAllTemplatesExcludingEagerRelationship(Pageable pageable) {
        log.debug("Request to get all Permissions");
        return
permissionRepository.findAllTemplatesExcludingEagerRelationship(pageable).map(per
missionMapper::toDto);
    }

    public Page<PermissionDTO> findAllTemplatesWithEagerRelationships(Pageable
pageable) {
        return
permissionRepository.findAllTemplatesWithEagerRelationships(pageable).map(Permis
sionMapper::toPermissionDto);
    }
```

```
    public Page<PermissionDTO>
findAllPermissionsByUserWithEagerRelationships(Pageable pageable, UUID id) {
        log.debug("Request to get all Permissions");
        return permissionRepository
                .findAllPermissionsByUserWithEagerRelationships(userAttributeRepository.f
indById(id).orElseThrow(), pageable)
                .map(PermissionMapper::toPermissionDto);
    }

    public Page<PermissionDTO>
findAllPermissionsByUserExcludingEagerRelationship(Pageable pageable, UUID id) {
        log.debug("Request to get all Permissions");
        return permissionRepository
                .findAllPermissionsByUser(userAttributeRepository.findById(id).orElseThro
w(), pageable)
                .map(permissionMapper::toDto);
    }


    @Transactional(readOnly = true)
    public Page<PermissionDTO>
findPermissionsByUserExcludingEagerRelationship(Pageable pageable, UUID
userAttributeId, Boolean activatedOnly) {
        log.debug("Request to get all Permissions");
        return permissionRepository
                .findPermissionsByUserExcludingEagerRelationship(userAttributeRepository
.findById(userAttributeId).orElseThrow(), activatedOnly, pageable)
                .map(permissionMapper::toDto);
    }

    public Page<PermissionDTO>
findPermissionsByUserWithEagerRelationships(Pageable pageable, UUID
userAttributeId, Boolean activatedOnly) {
        return permissionRepository
                .findPermissionsByUserWithEagerRelationships(userAttributeRepository.find
ById(userAttributeId).orElseThrow(), activatedOnly, pageable)
                .map(PermissionMapper::toPermissionDto);
    }

    public List<PermissionDTO>
findAllPermissionsByEnterpriseExcludingEagerRelationship(UUID enterpriseId) {
        log.debug("Request to get Permission by Enterprise id : {}", enterpriseId);
        return permissionRepository
                .findAllPermissionsByEnterpriseExcludingEagerRelationship(enterpriseId)
                .stream()
                .map(permissionMapper::toDto)
```

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

    public List<PermissionDTO>
findAllEnterpriseLevelPermissionsExcludingEagerRelationship(UUID enterpriseId) {
        log.debug("Request to get enterprise-level Permissions by Enterprise id : {}",
enterpriseId);
        return permissionRepository
            .findAllEnterpriseTemplatePermissionsExcludingEagerRelationship(enterpris
eId)
            .stream()
            .map(permissionMapper::toDto)
            .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public Optional<PermissionDTO> findOne(UUID id) {
        log.debug("Request to get Permission : {}", id);
        return
permissionRepository.findOneWithEagerRelationships(id).map(PermissionMapper::toP
ermissionDto);
    }

    public void delete(UUID id) {
        log.debug("Request to delete Permission : {}", id);
        permissionRepository.deleteById(id);
    }

    public List<Permission> createCopiesByTemplateAndAssign(Enterprise enterprise)
{
        return permissionRepository
            .findAllTemplatesWithEagerRelationships(Pageable.unpaged())
            .stream()
            .map(template -> deepCopyAndSave(template, permission ->
permission.setEnterprise(enterprise)))
            .collect(Collectors.toList());

    }

    public List<Permission> createCopiesByTemplateAndAssign(UserAttribute user) {
        return permissionRepository
            .findAllTemplatesWithEagerRelationships(Pageable.unpaged())
            .stream()
            .map(template -> deepCopyAndSave(template, permission ->
permission.setUser(user)))
            .collect(Collectors.toList());
```

```
    }

    @Transactional
    public List<PermissionDTO> deepCopyAndSave(List<PermissionDTO>
originalDTOs, Consumer<Permission> additionalProcessing) {
        if (Objects.nonNull(originalDTOs))
            return originalDTOs.stream()
                    .map(dto ->
permissionRepository.findOneWithEagerRelationships(dto.getId()).orElseThrow())
                    .map(permission -> deepCopyAndSave(permission, additionalProcessing))
                    .map(permissionMapper::toDto)
                    .collect(Collectors.toList());

        return Collections.emptyList();
    }

    public Permission deepCopyAndSave(Permission original, Consumer<Permission>
additionalProcessing) {
        return permissionRepository.saveAndFlush(deepCopy(original,
additionalProcessing));
    }

    private Permission deepCopy(Permission original, Consumer<Permission>
additionalProcessing) {
        Permission copy = shallowCopy(original);
        copy.setParent(original);
        additionalProcessing.accept(copy);

        boolean isTemplate;
        Enterprise enterprise;
        if (Objects.nonNull(copy.getUser())) {
            isTemplate = false;
            enterprise = original.getEnterprise();
            copy.setEnterprise(enterprise);
        } else {
            isTemplate = true;
            enterprise = copy.getEnterprise();
        }
        copy.setEndpoints(

endpointGroupServiceImpl.createCopiesByTemplateAndAssign(original.getEndpoints(
), endpointGroup -> {
                    endpointGroup.setTemplate(isTemplate);
                    endpointGroup.setEnterprise(enterprise);
                })
        );
        copy.setComponentAccesses(
```

```
componentAccessServiceImpl.createCopiesByTemplateAndAssign(original.getCompon
entAccesses(), componentAccess -> {
                componentAccess.setTemplate(isTemplate);
                componentAccess.setEnterprise(enterprise);
            })
    );
    return copy;
  }

  public Permission shallowCopy(Permission original) {
    return new Permission()
        .name(original.getName())
        .activated(original.getActivated())
        .enableByDefault(original.getEnableByDefault())
        .allowAutoUpdate(original.getAllowAutoUpdate())
        .paidFeature(original.getPaidFeature())
        .annotation(original.getAnnotation());
  }

  @Transactional
  public PermissionDTO assignPermissionToEnterprise(UUID permissionId, UUID
enterpriseId) {
    log.debug("Request to convert 1-st level Permission : {} to the 2-nd one and assign
to the Enterprise : {}", permissionId, enterpriseId);
    Permission permission =
permissionRepository.findOneWithEagerRelationships(permissionId).orElseThrow();
    Enterprise enterprise = enterpriseRepository.findById(enterpriseId).orElseThrow();

    Permission assignedPermission = this.deepCopyAndSave(permission, p ->
p.setEnterprise(enterprise));
    return permissionMapper.toDto(assignedPermission);
  }

  @Transactional
  public PermissionDTO assignPermissionToUser(UUID permissionId, UUID
userAttributeId) {
    log.debug("Request to convert 2-nd level Permission : {} to the 3-rd one and
assign to the UserAttribute : {}", permissionId, userAttributeId);
    Permission permission =
permissionRepository.findOneWithEagerRelationships(permissionId).orElseThrow();
    UserAttribute userAttribute =
userAttributeRepository.findById(userAttributeId).orElseThrow();

    Permission assignedPermission = this.deepCopyAndSave(permission, p ->
p.setUser(userAttribute));
    return permissionMapper.toDto(assignedPermission);
```

```
    }
}
```

PermissionDMResource.java

```
package com.prodautomation.web.rest.datamanger;

import com.prodautomation.repository.PermissionRepository;
import com.prodautomation.security.AuthoritiesConstants;
import com.prodautomation.service.EnterpriseService;
import com.prodautomation.service.PermissionService;
import com.prodautomation.service.PermissionServiceImpl;
import com.prodautomation.service.dto.PermissionDTO;
import com.prodautomation.web.rest.errors.BadRequestAlertException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import tech.jhipster.web.util.HeaderUtil;
import tech.jhipster.web.util.PaginationUtil;
import tech.jhipster.web.util.ResponseUtil;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;
import java.util.Objects;
import java.util.Optional;
import java.util.UUID;

@RestController
@RequestMapping("/api/data-manager")
@PreAuthorize("hasAuthority(\"" + AuthoritiesConstants.DATA_MANAGER + "\")")
public class PermissionDMResource {

    private final Logger log = LoggerFactory.getLogger(PermissionDMResource.class);

    private static final String ENTITY_NAME = "permission";

    @Value("${jhipster.clientApp.name}")
    private String applicationName;
```

```
    private final PermissionService permissionServiceImpl;

    private final PermissionRepository permissionRepository;

    private final EnterpriseService enterpriseService;

    public PermissionDMResource(PermissionServiceImpl permissionServiceImpl,
PermissionRepository permissionRepository, EnterpriseService enterpriseService) {
        this.permissionServiceImpl = permissionServiceImpl;
        this.permissionRepository = permissionRepository;
        this.enterpriseService = enterpriseService;
    }

    @PostMapping("/permissions")
    public ResponseEntity<PermissionDTO> createPermission(@RequestBody
PermissionDTO permissionDTO) throws URISyntaxException {
        log.debug("REST request to save Permission : {}", permissionDTO);
        if (permissionDTO.getId() != null) {
            throw new BadRequestAlertException("A new permission cannot already have
an ID", ENTITY_NAME, "idexists");
        } else if (permissionDTO.getEnterprise() == null) {
            throw new BadRequestAlertException("A new permissionDTO don`t have an
Enterprise", ENTITY_NAME, "enterprisenotfound");
        } else
if(!enterpriseService.existForCurrentUser(permissionDTO.getEnterprise().getId())) {
            throw new BadRequestAlertException("Entity unavailable for current user",
ENTITY_NAME, "unavailableforcurrentuser");
        }
        PermissionDTO result = permissionServiceImpl.save(permissionDTO);
        return ResponseEntity
            .created(new URI("/api/data-manager/permissions/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, true,
ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

    @PostMapping("/permission/user")
    public ResponseEntity<PermissionDTO> assignPermissionToUser(
            @RequestParam("permissionId") UUID permissionId,
@RequestParam("userId") UUID userAttributeId
    ) {
        log.debug("REST request to assign Permission : {} to the Enterprise : {}",
permissionId, userAttributeId);
        if (!permissionRepository.existsById(permissionId)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
```

```
        }

        PermissionDTO assignedPermission =
permissionServiceImpl.assignPermissionToUser(permissionId, userAttributeId);
        return ResponseEntity
                .ok()
                .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true,
ENTITY_NAME, assignedPermission.getId().toString()))
                .body(assignedPermission);
    }

    @PutMapping("/permissions/{id}")
    public ResponseEntity<PermissionDTO> updatePermission(
        @PathVariable(value = "id", required = false) final UUID id,
        @RequestBody PermissionDTO permissionDTO
    ) throws URISyntaxException {
        log.debug("REST request to update Permission : {}, {}", id, permissionDTO);
        if (permissionDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, permissionDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME,
"idinvalid");
        }

        if (!permissionRepository.existsById(id)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }

        if(!enterpriseService.existForCurrentUser(permissionDTO.getEnterprise().getId()))
{
            throw new BadRequestAlertException("Entity unavailable for current user",
ENTITY_NAME, "unavailableforcurrentuser");
        }

        PermissionDTO result = permissionServiceImpl.update(permissionDTO);
        return ResponseEntity
            .ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true,
ENTITY_NAME, permissionDTO.getId().toString()))
            .body(result);
    }

    @PatchMapping(value = "/permissions/{id}", consumes = { "application/json",
"application/merge-patch+json" })
    public ResponseEntity<PermissionDTO> partialUpdatePermission(
```

```
        @PathVariable(value = "id", required = false) final UUID id,
        @RequestBody PermissionDTO permissionDTO
    ) throws URISyntaxException {
        log.debug("REST request to partial update Permission partially : {}, {}", id,
permissionDTO);
        if (permissionDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, permissionDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME,
"idinvalid");
        }

        if (!permissionRepository.existsById(id)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }

        if(!enterpriseService.existForCurrentUser(permissionDTO.getEnterprise().getId()))
{
            throw new BadRequestAlertException("Entity unavailable for current user",
ENTITY_NAME, "unavailableforcurrentuser");
        }

        Optional<PermissionDTO> result =
permissionServiceImpl.partialUpdate(permissionDTO);

        return ResponseUtil.wrapOrNotFound(
            result,
            HeaderUtil.createEntityUpdateAlert(applicationName, true, ENTITY_NAME,
permissionDTO.getId().toString())
        );
    }

    @GetMapping("/permissions/enterprise/{enterpriseId}")
    public ResponseEntity<List<PermissionDTO>>
getAllEnterpriseTemplatePermissions(
            @PathVariable(value = "enterpriseId") final UUID enterpriseId
    ) {
        log.debug("REST request to get a page of enterprise-level template Permissions.
Enterprise id: {}", enterpriseId);

        if(!enterpriseService.existForCurrentUser(enterpriseId)) {
            throw new BadRequestAlertException("Entity unavailable for current user",
ENTITY_NAME, "unavailableforcurrentuser");
        }
```

```
        List<PermissionDTO> enterprisePermissions =
permissionServiceImpl.findAllEnterpriseLevelPermissionsExcludingEagerRelationship
(enterpriseId);
        return ResponseEntity
                .ok()
                .body(enterprisePermissions);
    }

    @GetMapping("/permissions/user/{userId}")
    public ResponseEntity<List<PermissionDTO>> getAllUserPermissions(
            @org.springdoc.api.annotations.ParameterObject Pageable pageable,
            @PathVariable(value = "userId") final UUID userId
    ) {
        log.debug("REST request to get a page of user's Permissions. User id: `{}`",
userId);
        Page<PermissionDTO> page =
permissionServiceImpl.findAllPermissionsByUserExcludingEagerRelationship(pageabl
e, userId);

        HttpHeaders headers =
PaginationUtil.generatePaginationHttpHeaders(ServletUriComponentsBuilder.fromCurr
entRequest(), page);
        return ResponseEntity.ok().headers(headers).body(page.getContent());
    }

    @GetMapping("/permissions/{id}")
    public ResponseEntity<PermissionDTO> getPermission(@PathVariable UUID id) {
        log.debug("REST request to get Permission : {}", id);
        Optional<PermissionDTO> permissionDTO = permissionServiceImpl.findOne(id);

        if(permissionDTO.isPresent()
&& !enterpriseService.existForCurrentUser(permissionDTO.get().getEnterprise().getId(
))) {
            throw new BadRequestAlertException("Entity unavailable for current user",
ENTITY_NAME, "unavailableforcurrentuser");
        }

        return ResponseUtil.wrapOrNotFound(permissionDTO);
    }

    @DeleteMapping("/permissions/{id}")
    public ResponseEntity<Void> deletePermission(@PathVariable UUID id) {
        log.debug("REST request to delete Permission : {}", id);
        permissionServiceImpl.delete(id);
        return ResponseEntity
            .noContent()
            .headers(HeaderUtil.createEntityDeletionAlert(applicationName, true,
```

```
ENTITY_NAME, id.toString()))
            .build();
    }
}
```

PermissionResource.java

```
package com.prodautomation.web.rest.admin;

import com.prodautomation.repository.PermissionRepository;
import com.prodautomation.security.AuthoritiesConstants;
import com.prodautomation.service.PermissionService;
import com.prodautomation.service.PermissionServiceImpl;
import com.prodautomation.service.dto.PermissionDTO;
import com.prodautomation.web.rest.errors.BadRequestAlertException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import tech.jhipster.web.util.HeaderUtil;
import tech.jhipster.web.util.PaginationUtil;
import tech.jhipster.web.util.ResponseUtil;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.List;
import java.util.Objects;
import java.util.Optional;
import java.util.UUID;

@RestController
@RequestMapping("/api")
@PreAuthorize("hasAuthority(\"" + AuthoritiesConstants.ADMIN + "\")")
public class PermissionResource {

    private final Logger log = LoggerFactory.getLogger(PermissionResource.class);

    private static final String ENTITY_NAME = "permission";
```

```
    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final PermissionService permissionServiceImpl;

    private final PermissionRepository permissionRepository;

    public PermissionResource(PermissionServiceImpl permissionServiceImpl,
PermissionRepository permissionRepository) {
        this.permissionServiceImpl = permissionServiceImpl;
        this.permissionRepository = permissionRepository;
    }

    @PostMapping("/permissions")
    public ResponseEntity<PermissionDTO> createPermission(@RequestBody
PermissionDTO permissionDTO) throws URISyntaxException {
        log.debug("REST request to save Permission : {}", permissionDTO);
        if (permissionDTO.getId() != null) {
            throw new BadRequestAlertException("A new permission cannot already have
an ID", ENTITY_NAME, "idexists");
        }
        PermissionDTO result = permissionServiceImpl.save(permissionDTO);
        return ResponseEntity
            .created(new URI("/api/permissions/" + result.getId()))
            .headers(HeaderUtil.createEntityCreationAlert(applicationName, true,
ENTITY_NAME, result.getId().toString()))
            .body(result);
    }

    @PutMapping("/permissions/{id}")
    public ResponseEntity<PermissionDTO> updatePermission(
        @PathVariable(value = "id", required = false) final UUID id,
        @RequestBody PermissionDTO permissionDTO
    ) throws URISyntaxException {
        log.debug("REST request to update Permission : {}, {}", id, permissionDTO);
        if (permissionDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, permissionDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME,
"idinvalid");
        }

        if (!permissionRepository.existsById(id)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }
```

```
        PermissionDTO result = permissionServiceImpl.update(permissionDTO);
        return ResponseEntity
            .ok()
            .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true,
ENTITY_NAME, permissionDTO.getId().toString()))
            .body(result);
    }

    @PatchMapping(value = "/permissions/{id}", consumes = { "application/json",
"application/merge-patch+json" })
    public ResponseEntity<PermissionDTO> partialUpdatePermission(
        @PathVariable(value = "id", required = false) final UUID id,
        @RequestBody PermissionDTO permissionDTO
    ) throws URISyntaxException {
        log.debug("REST request to partial update Permission partially : {}, {}", id,
permissionDTO);
        if (permissionDTO.getId() == null) {
            throw new BadRequestAlertException("Invalid id", ENTITY_NAME, "idnull");
        }
        if (!Objects.equals(id, permissionDTO.getId())) {
            throw new BadRequestAlertException("Invalid ID", ENTITY_NAME,
"idinvalid");
        }

        if (!permissionRepository.existsById(id)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }

        Optional<PermissionDTO> result =
permissionServiceImpl.partialUpdate(permissionDTO);

        return ResponseUtil.wrapOrNotFound(
            result,
            HeaderUtil.createEntityUpdateAlert(applicationName, true, ENTITY_NAME,
permissionDTO.getId().toString())
        );
    }

    @GetMapping("/permissions/system-template")
    public ResponseEntity<List<PermissionDTO>> getAllSystemTemplatePermissions(
            @org.springdoc.api.annotations.ParameterObject Pageable pageable,
            @RequestParam(required = false, defaultValue = "false") boolean eagerload
    ) {
        log.debug("REST request to get a page of system-level template Permissions");
        Page<PermissionDTO> page;
```

```
        if (eagerload) {
            page =
permissionServiceImpl.findAllTemplatesWithEagerRelationships(pageable);
        } else {
            page =
permissionServiceImpl.findAllTemplatesExcludingEagerRelationship(pageable);
        }
        HttpHeaders headers =
PaginationUtil.generatePaginationHttpHeaders(ServletUriComponentsBuilder.fromCurr
entRequest(), page);
        return ResponseEntity.ok().headers(headers).body(page.getContent());
    }

    @GetMapping("/permissions/enterprise-template/{enterpriseId}")
    public ResponseEntity<List<PermissionDTO>>
getAllEnterpriseTemplatePermissions(
            @PathVariable(value = "enterpriseId") final UUID enterpriseId
    ) {
        log.debug("REST request to get a page of enterprise-level template Permissions.
Enterprise id: `{}`", enterpriseId);
        List<PermissionDTO> enterprisePermissions =
permissionServiceImpl.findAllEnterpriseLevelPermissionsExcludingEagerRelationship
(enterpriseId);
        return ResponseEntity
                .ok()
                .body(enterprisePermissions);
    }

    @GetMapping("/permissions/user-template/{userId}")
    public ResponseEntity<List<PermissionDTO>> getAllUserPermissions(
            @org.springdoc.api.annotations.ParameterObject Pageable pageable,
            @PathVariable(value = "userId") final UUID userId
    ) {
        log.debug("REST request to get a page of user's Permissions. User id: `{}`",
userId);
        Page<PermissionDTO> page =
permissionServiceImpl.findAllPermissionsByUserExcludingEagerRelationship(pageabl
e, userId);

        HttpHeaders headers =
PaginationUtil.generatePaginationHttpHeaders(ServletUriComponentsBuilder.fromCurr
entRequest(), page);
        return ResponseEntity.ok().headers(headers).body(page.getContent());
    }

    @GetMapping("/permissions/{id}")
    public ResponseEntity<PermissionDTO> getPermission(@PathVariable UUID id) {
```

```
        log.debug("REST request to get Permission : {}", id);
        Optional<PermissionDTO> permissionDTO = permissionServiceImpl.findOne(id);
        return ResponseUtil.wrapOrNotFound(permissionDTO);
    }

    @PostMapping("/permissions/enterprise-template/{enterpriseId}")
    public ResponseEntity<PermissionDTO> assignPermissionToEnterprise(
            @PathVariable(value = "enterpriseId") final UUID enterpriseId,
            @RequestParam("permissionId") UUID permissionId
    ) {
        log.debug("REST request to assign Permission : {} to the Enterprise : {}",
permissionId, enterpriseId);
        if (!permissionRepository.existsById(permissionId)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }

        PermissionDTO assignedPermission =
permissionServiceImpl.assignPermissionToEnterprise(permissionId, enterpriseId);
        return ResponseEntity
                .ok()
                .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true,
ENTITY_NAME, assignedPermission.getId().toString()))
                .body(assignedPermission);
    }

    @PostMapping("/permissions/user-template/{userId}")
    public ResponseEntity<PermissionDTO> assignPermissionToUser(
            @PathVariable(value = "userId") final UUID userAttributeId,
            @RequestParam("permissionId") UUID permissionId
    ) {
        log.debug("REST request to assign Permission : {} to the UserAttribute : {}",
permissionId, userAttributeId);
        if (!permissionRepository.existsById(permissionId)) {
            throw new BadRequestAlertException("Entity not found", ENTITY_NAME,
"idnotfound");
        }

        PermissionDTO assignedPermission =
permissionServiceImpl.assignPermissionToUser(permissionId, userAttributeId);
        return ResponseEntity
                .ok()
                .headers(HeaderUtil.createEntityUpdateAlert(applicationName, true,
ENTITY_NAME, assignedPermission.getId().toString()))
                .body(assignedPermission);
    }
```

```
    @DeleteMapping("/permissions/{id}")
    public ResponseEntity<Void> deletePermission(@PathVariable UUID id) {
        log.debug("REST request to delete Permission : {}", id);
        permissionServiceImpl.delete(id);
        return ResponseEntity
            .noContent()
            .headers(HeaderUtil.createEntityDeletionAlert(applicationName, true,
ENTITY_NAME, id.toString()))
            .build();
    }
}
```

Додаток В

(обов'язковий)

ІЛЮСТРАТИВНА ЧАСТИНА
«ПРОГРАМНИЙ МОДУЛЬ КЕРУВАННЯ ДОЗВОЛАМИ ТА ДОСТУПАМИ ДЛЯ СИСТЕМ АВТОМАТИЗАЦІЇ ВИРОБНИЧИХ ПРОЦЕСІВ»

Виконав: студент 4 курсу,групи 1КН-21б
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Поліщук В. Л.
(прізвище та ініціали)

Керівник: к.т.н., доцент
Богач І.В.
(прізвище та ініціали)
«03» червня 2025 р.

Структурна схема взаємодії компонентів системи автоматизації виробничих процесів

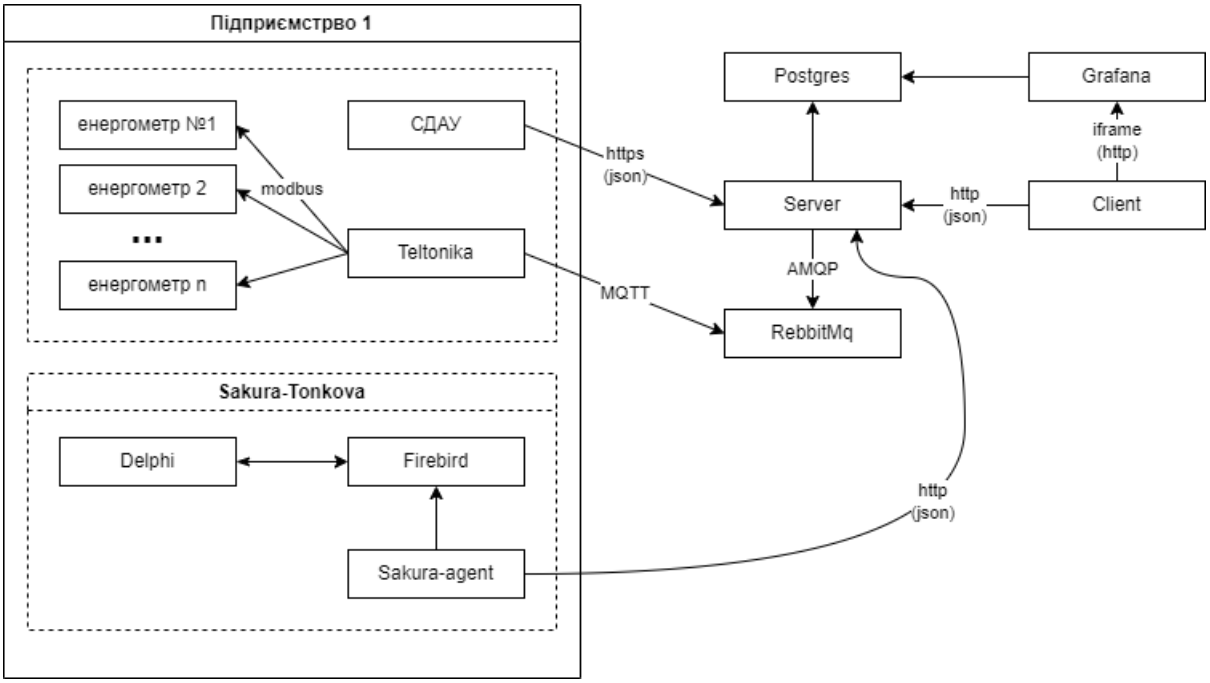

Рисунок А.1 – Структурна схема взаємодії компонентів системи автоматизації виробничих процесів

Схема логічного поділу ролей на типи

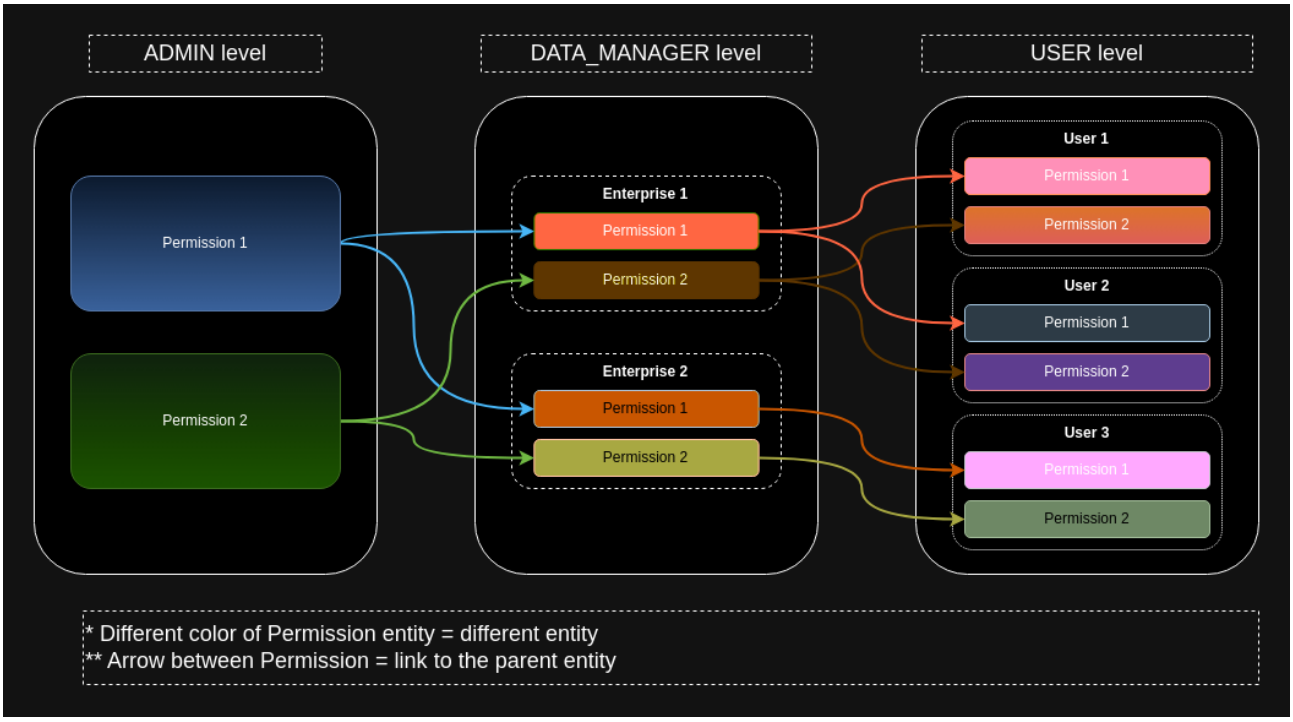

Рисунок А.2 – Схема логічного поділу ролей на типи

Схема композиції сутностей керування доступом

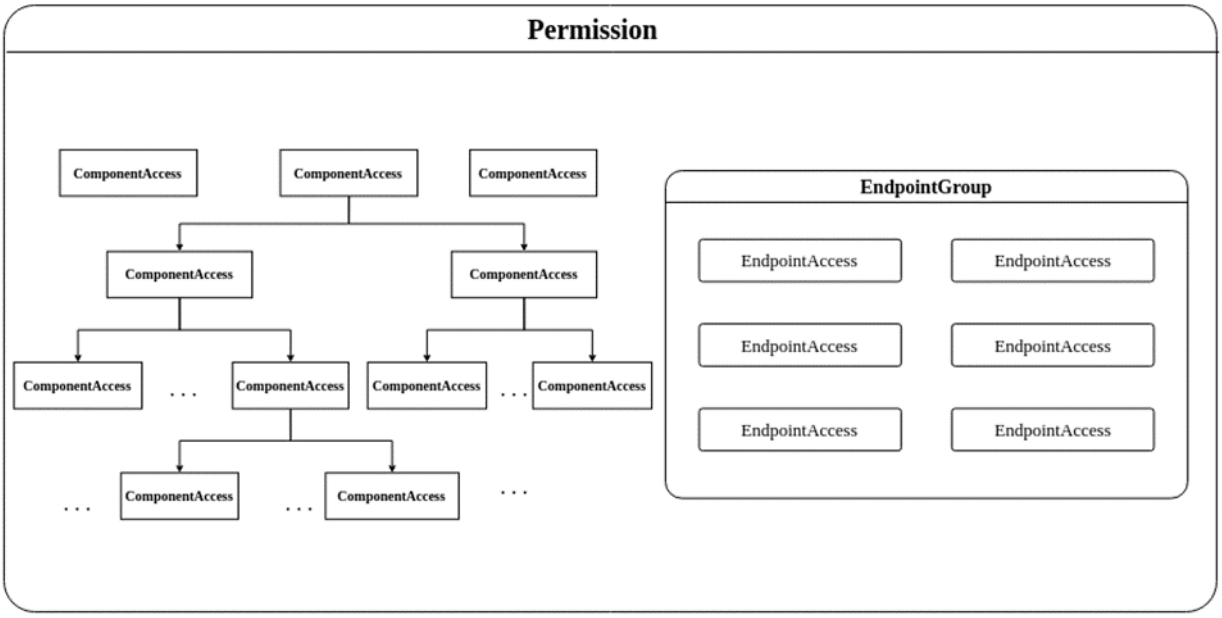

Рисунок А.3 – Схема композиції сутностей керування доступом

UML-діаграма взаємодії шарів застосунку на прикладі Permission

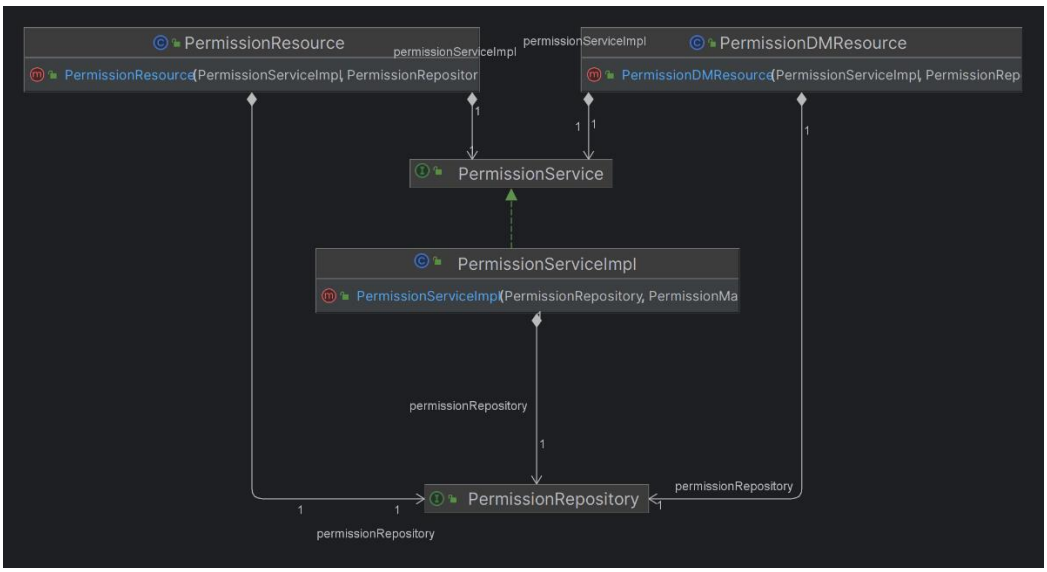

Рисунок А.4 – UML-діаграма взаємодії шарів застосунку на прикладі Permission

Додаток Г
(довідниковий)
Інструкція користувача

Для використання розробленого програмного модулю керування дозволами та доступами, потрібно:

1. Здійснити HTTP POST-запит на ендпойнт *http://localhost:8080/api/authenticate* контролера, що виконує автентифікацію на основі ім’я існуючого користувача та паролю, що містяться в тілі запиту у форматі JSON. Запит на отримання особистого токена представлено на рисунку 4.1. У результаті запиту користувач отримує власний JWT токен. Тепер користувач може використовувати токен для доступу до ресурсів веб-додатку, додаючи до кожного HTTP-запиту заголовок «Authorization» зі значенням «Bearer <особистий токен>». Якщо програмний модуль розгорнуто на віддалено сервері, то замість "*localhost:8080"* потрібно вказати відповідні IP-адресу з портом або доменне ім’я сервера.

2. Отримати шаблони Permission системного рівня за допомогою GET-запиту на шлях *http://localhost:8080/api/permissions/system-template* або створити власні шаблони з потрібними дозволами за допомогою GET-запиту на *http://localhost:8080/api/permissions*.

3. Присвоїти відповідні шаблони ролей підприємству. Для цього потрібно здійснити запит на адресу *http://localhost:8080/api/permissions/enterprise-template/{{enterpriseId}}*, вказавши при цьому змінною URL шляху унікальний ідентифікато підприємства та ідентифікатор дозволу як параметр запиту. В результаті створюється копія ролі з прив’язкою з підприємства, що є дочірньою по відношенню до шаблону.

4. Здійснити POST-запит за адресою *http://localhost:8080/api/permissions/user-template/{{userId}}*, передавши унікальний ідентифікатор користувача змінною шляху та ідентифікатор ролі-шаблону 2 типу як параметр запиту.

5. Отримати роль з набором дозволів користувача, виконавши GET-запит за адресою *http://localhost:8080/api/permissions/user-template/{{userId}}*, вказавши ідентифікатор користувача змінною шляху.

Додаток Д (довідниковий)

ІННОВІННПРОМ
Україна 21036
Вінниця / вул. Шевченка 5А
Тел: (0432) 50-20-05
info@innovinnprom.com

Вих. № 04/06/25
Від «04» червня 2025 р.

Довідка

Видана Поліщуку Володимиру Леонідовичу в тому, що результати, одержані ним у процесі виконання бакалаврської кваліфікаційної роботи на тему «Програмний модуль керування дозволами та доступами для систем автоматизації виробничих процесів», а саме розроблені алгоритми та програмна реалізація, планується використати в розробках ТОВ «ІННОВІННПРОМ».

Довідка видана для пред'явлення за місцем вимоги.

Директор
ТОВ «ІННОВІННПРОМ» **Марина СКИДАН**