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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка мікросервісу авторизації для управління доступом до корпоративних додатків»

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

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

_____ Дмитро ОЛАБА
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Дмитро ОЛАБА

Керівник: Богдан ХУДІК
доктор філософії(PhD)

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Олабі Дмитру Романовичу

1. Тема кваліфікаційної роботи: «Розробка мікросервісу авторизації для управління доступом до корпоративних додатків»
керівник кваліфікаційної роботи доктор філософії (PhD) Богдан ХУДІК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

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

3. Вихідні дані до кваліфікаційної роботи: аналіз існуючих підходів до управління доступом, опис вимог до мікросервісної системи авторизації, документація з технологій автентифікації та авторизації, технічна документація фреймворків та бібліотек для розробки мікросервісі

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз предметної області та постановка задачі розробки мікросервісу авторизації для корпоративних додатків.
2. Проектування архітектури мікросервісу Access Guard та системи авторизації в мікросервісному середовищі.
3. Реалізація та тестування системи автентифікації та авторизації з підтримкою різних механізмів верифікації користувачів.

4. Розгортання, інтеграція та моніторинг роботи системи в корпоративному середовищі.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення
3. Програмні засоби реалізації
4. Архітектура та взаємодія мікросервісів Access Guard
5. Механізми автентифікації Access Guard.
6. Діаграма класів ключових таблиць сервісу Access-guard.
7. Діаграма послідовності автентифікації.
8. Екранні форми
9. Апробація результатів дослідження

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Визначення вимог до мікросервісу авторизації	14.03-20.03.2025	
4	Проектування архітектури системи Access Guard	21.03-30.03.2025	
5	Проектування інформаційної моделі та схеми бази даних	31.03-07.04.2025	
6	Розробка системи автентифікації з підтримкою JWT та refresh токенів	08.04-14.04.2025	
7	Реалізація двофакторної автентифікації через Mattermost	15.04-18.04.2025	
8	Розробка системи авторизації та управління ролями	19.04-23.04.2025	
9	Створення адміністративного інтерфейсу	24.04-28.04.2025	
10	Тестування та інтеграція з корпоративними сервісами	29.04-03.05.2025	

11	Налаштування моніторингу та системи логування	04.05-08.05.2025	
12	Оформлення роботи: вступ, висновки, реферат	09.05-11.05.2025	
13	Розробка демонстраційних матеріалів	12.05-18.05.2025	
14	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Дмитро ОЛАБА
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан ХУДІК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 98 стор., 26 табл., 68 рис., 20 джерел.

Мета роботи — поліпшення уніфікованого керування доступом до корпоративних застосунків у мікросервісній архітектурі шляхом впровадження мікросервісу Access Guard, що підвищує рівень безпеки інформаційних ресурсів, спрощує адміністрування прав та забезпечує єдину точку контролю активності користувачів.

Об'єкт дослідження — процес автентифікації та авторизації користувачів у децентралізованому середовищі мікросервісної архітектури корпоративних додатків.

Предмет дослідження — технології побудови мікросервісу Access Guard для централізованої автентифікації, авторизації та аудиту доступу в розподіленій мікросервісній архітектурі.

Короткий зміст роботи: У роботі проаналізовано архітектурні підходи до реалізації систем управління доступом у мікросервісному середовищі. Досліджено існуючі рішення (Keycloak, Auth0, IdentityServer) та виявлено їхні обмеження для корпоративного використання.

Спроектовано та реалізовано мікросервіс Access Guard на базі реактивного стеку технологій (Spring WebFlux, Project Reactor) з підтримкою різних механізмів автентифікації (JWT, двофакторна через Mattermost, пристрої, API-ключі).

Реалізовано систему авторизації з гнучкою ієрархією ролей, каскадним управлінням доступом до додатків та детальним аудитом змін.

Розроблено адміністративний інтерфейс для керування користувачами та їх правами, систему моніторингу з використанням Prometheus та Grafana, а також впроваджено механізми авторизації для розподілених запитів.

Система контейнеризована з використанням Docker та інтегрована з корпоративною інфраструктурою через Discovery Server.

Сферою використання застосунку є корпоративні інформаційні системи з мікросервісною архітектурою, де необхідне централізоване управління автентифікацією та авторизацією користувачів. Мікросервіс Access Guard успішно впроваджено в компанії TSA для забезпечення безпеки доступу до розподілених корпоративних ресурсів.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІС, АВТЕНТИФІКАЦІЯ, АВТОРИЗАЦІЯ, JWT, ДВОФАКТОРНА АВТЕНТИФІКАЦІЯ, SPRING WEBFLUX, РЕАКТИВНЕ ПРОГРАМУВАННЯ, КОНТЕЙНЕРИЗАЦІЯ, DOCKER, МОНІТОРИНГ

ЗМІСТ

ВСТУП

Актуальність: У сучасному корпоративному середовищі цифрова трансформація стала невід'ємною частиною розвитку бізнесу. Спостерігається еволюція архітектури програмного забезпечення від монолітних систем до мікросервісних екосистем. Ця трансформація породжує низку специфічних проблем, серед яких особливо актуальною є забезпечення надійних та ефективних механізмів керування доступом користувачів до розподілених корпоративних ресурсів.

Мікросервісна архітектура, попри численні переваги, створює комплексне середовище автентифікаційних та авторизаційних викликів. Окремі мікросервіси, функціонуючи як автономні одиниці, потребують механізмів верифікації ідентичності користувачів та визначення їхніх прав доступу. Реалізація цих механізмів у кожному сервісі окремо призводить до дублювання коду, ускладнення підтримки та формування потенційних вразливостей системи безпеки.

За таких умов виникає необхідність створення централізованої системи контролю доступу, яка функціонуватиме як єдине джерело інформації щодо автентифікації та авторизації користувачів у розподіленому середовищі. Така система повинна забезпечувати уніфіковану автентифікацію, централізоване управління правами, гнучку інтеграцію, масштабованість та високий рівень безпеки.

Об'єктом дослідження є процес автентифікації та авторизації користувачів у розподіленому середовищі мікросервісної архітектури [1] корпоративних додатків.

Предметом дослідження є технології побудови мікросервісу Access Guard для централізованої автентифікації, авторизації та аудиту доступу в розподіленій мікросервісній архітектурі.

Метою роботи є поліпшення уніфікованого керування доступом до корпоративних застосунків у мікросервісній архітектурі шляхом впровадження мікросервісу Access Guard, що підвищує рівень безпеки інформаційних ресурсів, спрощує адміністрування прав та забезпечує єдину точку контролю активності користувачів.

Методи дослідження: у процесі роботи було проаналізовано існуючі підходи до управління доступом у мікросервісних архітектурах, досліджено провідні рішення на ринку (Keycloak, Auth0, IdentityServer) та виявлено їхні обмеження для корпоративного використання. На основі цього аналізу було спроектовано архітектуру мікросервісу Access Guard з використанням реактивного стеку технологій. Дослідження процесу інтеграції різних механізмів автентифікації проводилося під час безпосередньої розробки системи, включаючи JWT-токени, двофакторну автентифікацію через Mattermost, автентифікацію пристроїв та API-ключі для міжсервісної взаємодії.

Наукова новизна роботи полягає у розробці комплексного підходу до централізованого управління доступом у мікросервісному середовищі з використанням реактивної архітектури, що забезпечує високу продуктивність та ефективне використання ресурсів. Запропоновано оригінальний механізм двофакторної автентифікації через корпоративний месенджер Mattermost, реалізовано систему каскадного управління доступом до додатків та їх залежностей, а також впроваджено багаторівневу систему аудиту з детальним відстеженням усіх змін у ключових таблицях бази даних.

Практична значущість результатів полягає у можливості використання розробленого мікросервісу для централізованого управління доступом у корпоративних системах з мікросервісною архітектурою, що дозволяє значно спростити адміністрування прав доступу, підвищити безпеку всієї системи та забезпечити ефективний моніторинг і аудит активності користувачів. Мікросервіс Access Guard успішно впроваджено в компанії TSA для забезпечення безпеки доступу до розподілених корпоративних ресурсів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис задачі та предметної галузі

У сучасному корпоративному середовищі цифрова трансформація стала невід'ємною частиною розвитку бізнесу. Спостерігається еволюція архітектури програмного забезпечення від монолітних систем до мікросервісних екосистем[2]. Ця трансформація породжує низку специфічних проблем, серед яких особливо актуальною є забезпечення надійних та ефективних механізмів керування доступом користувачів до розподілених корпоративних ресурсів.

Мікросервісна архітектура, попри численні переваги, створює комплексне середовище автентифікаційних та авторизаційних викликів. Окремі мікросервіси, функціонуючи як автономні одиниці, потребують механізмів верифікації ідентичності користувачів та визначення їхніх прав доступу. Реалізація цих механізмів у кожному сервісі окремо призводить до дублювання коду, ускладнення підтримки та формування потенційних вразливостей системи безпеки.

Аналіз традиційних підходів до управління доступом демонструє їхню недостатню ефективність у розподілених системах через такі причини:

1. Фрагментація автентифікаційних даних — відомості про користувачів розподіляються між різними компонентами системи;
2. Асинхронізація станів — зміна прав доступу в одному сегменті системи не гарантує оновлення в інших сегментах;
3. Відсутність централізованого контролю — ускладнюється реалізація процедур аудиту та моніторингу;
4. Різноманітність інтерфейсів — різні сервіси впроваджують несумісні механізми автентифікації.

За таких умов виникає необхідність створення централізованої системи контролю доступу, яка функціонуватиме як єдине джерело інформації щодо

автентифікації та авторизації користувачів у розподіленому середовищі. Така система повинна забезпечувати:

1. Уніфіковану автентифікацію — єдиний механізм верифікації ідентичності користувачів;
2. Централізоване управління правами — консолідація логіки авторизації в єдиному модулі;
3. Гнучку інтеграцію — можливість впровадження різних методів автентифікації;
4. Масштабованість — ефективне функціонування при зростанні кількості запитів;
5. Безпеку — стійкість до поширених видів кібератак.

У цьому контексті розробка мікросервісу Access Guard набуває значної наукової та практичної цінності як системне вирішення проблематики управління доступом у сучасних корпоративних інформаційних системах.

1.2 Аналіз існуючих рішень для управління доступом

На сучасному ринку існує декілька поширених систем управління автентифікацією та авторизацією, кожна з яких має свої особливості, переваги та обмеження. Розглянемо найпопулярніші рішення та порівняємо їх із пропонованим мікросервісом Access Guard (таблиця 1.1).

1.2.1 Keycloak

Keycloak — відкрита платформа управління ідентифікацією та доступом, розроблена Red Hat (рисунок 1.1). Система надає функції єдиного входу, забезпечує інтеграцію з різними протоколами автентифікації та пропонує широкі можливості для управління користувачами.

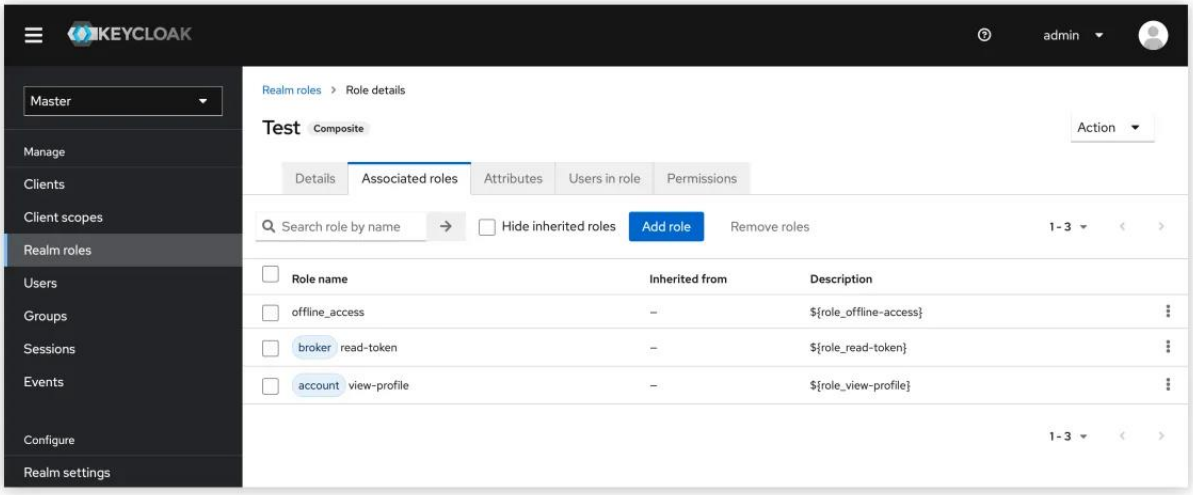


Рис. 1.1 Загальна архітектура системи Keycloak

Переваги:

1. Потужна готова інфраструктура SSO;
2. Підтримка стандартів OAuth 2.0, OpenID Connect, SAML 2.0 [3];
3. Вбудований адміністративний інтерфейс;
4. Розширена підтримка соціальних ідентифікаторів.

Недоліки:

1. Значні ресурсні вимоги до інфраструктури;
2. Складність конфігурації та адаптації під специфічні вимоги;
3. Обмежені можливості кастомізації бізнес-логіки;
4. Надмірність функціоналу для багатьох корпоративних сценаріїв.

1.2.2 Auth0

Auth0 — комерційна платформа управління ідентифікацією як послуга (IDaaS), що пропонує безпечні механізми автентифікації та авторизації (рисунок 1.2).

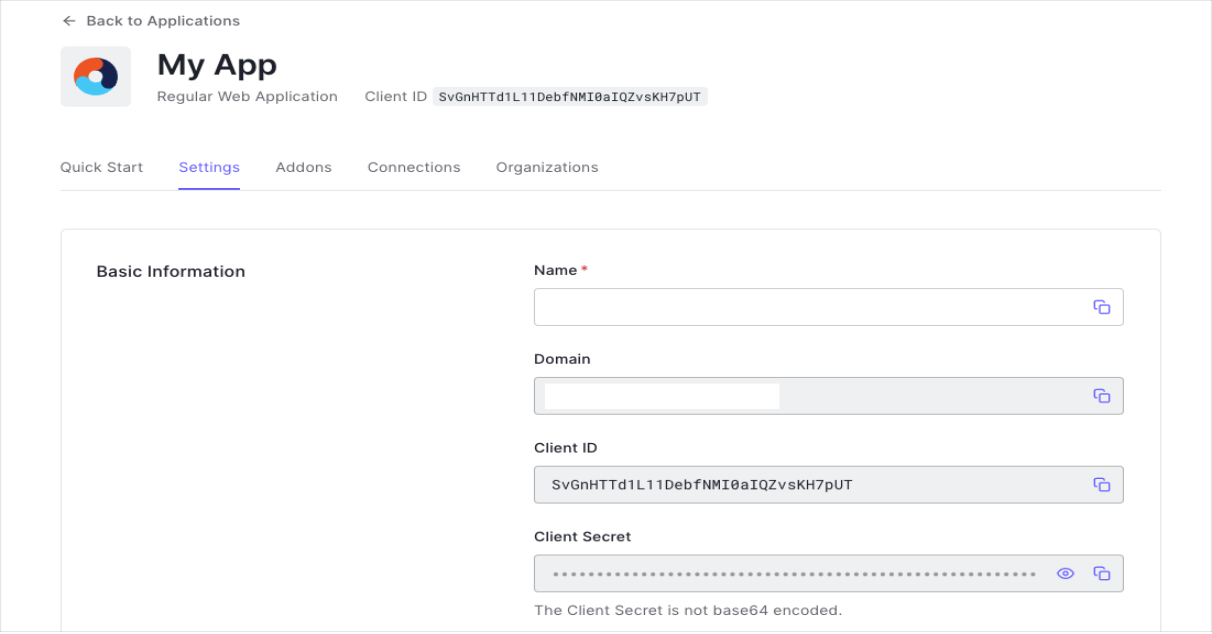


Рис. 1.2 Інтерфейс управління користувачами Auth0

Переваги:

1. Швидке розгортання та готові інтеграції;
2. Різноманітні соціальні конектори;
3. Регулярні оновлення та підтримка.

Недоліки:

1. Висока вартість при масштабуванні;
2. Обмежений контроль над інфраструктурою даних;
3. Залежність від зовнішнього провайдера;
4. Обмеження кастомізації для складних корпоративних сценаріїв.

1.2.3 IdentityServer

IdentityServer — відкрита платформа для .NET, що реалізує OpenID Connect та OAuth 2.0 для автентифікації та авторизації (рисунок 1.3).

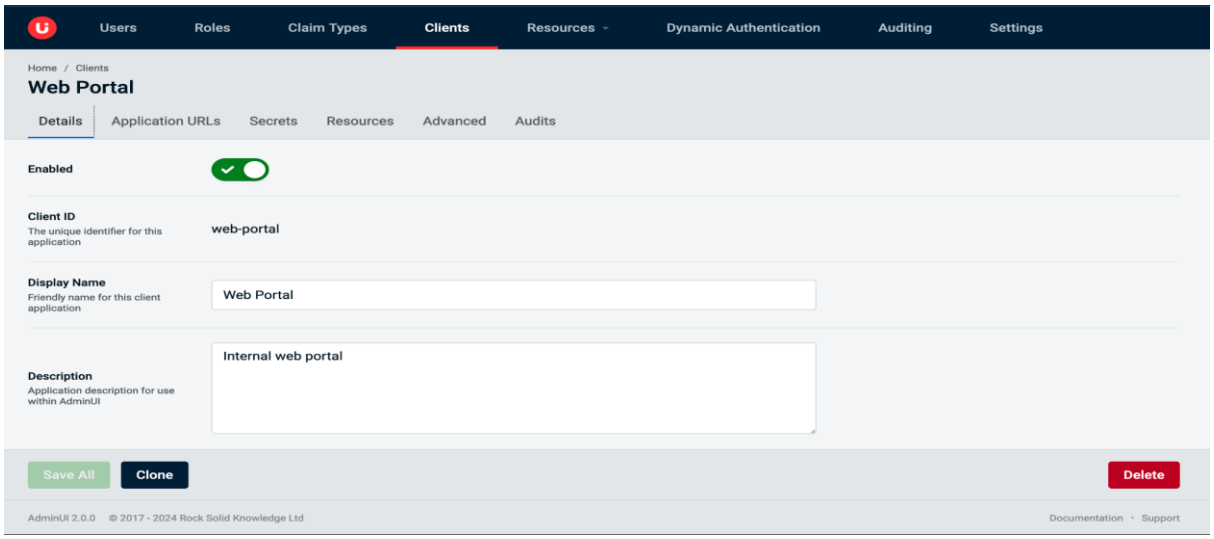


Рис. 1.3 Структура роботи IdentityServer

Переваги:

1. Добре інтегрується з екосистемою .NET;
2. Підтримка стандартних протоколів;
3. Гнучкість налаштування.

Недоліки:

1. Обмежена інтеграція з не-.NET системами;
2. Складність розгортання в гетерогенному середовищі;
3. Відносно обмежена спільнота порівняно з іншими рішеннями.

1.2.4 Порівняльна таблиця рішень

Таблиця 1.1

Порівняльна таблиця рішень для управління доступом

Характеристика	**Access Guard**	**Keycloak**	**Auth0**	**IdentityServer**
Цільова екосистема	Мікросервісна Java/Spring	Універсальна	Універсальна	.NET

Продовження таблиці 1.1

Порівняльна таблиця рішень для управління доступом

Характеристика	**Access Guard**	**Keycloak**	**Auth0**	**IdentityServer**
Використання ресурсів	Низьке	Високе	Незастосовно (SaaS)	Середнє
Інтеграція з Mattermost	Так	Ні	Через додаткові конфігурації	Ні
Реактивний підхід	Так	Ні	Ні	Ні
Зберігання історії входів	Розширене	Базове	Розширене	Базове
Кастомізація авторизації	Висока	Середня	Низька	Середня
Управління пристроями	Так	Частково	Частково	Ні
Інтеграція з корпоративними системами	Нативна/Через API	Через конфігурації	Через API	Через конфігурації
Готовий адмін-інтерфейс	Так	Так	Так	Частково
Вартість	Внутрішня розробка	Безкоштовно (self-hosted)	Висока	Безкоштовно (self-hosted)

1.2.5 Обґрунтування актуальності розробки Access Guard

Здійснивши аналіз наявних технологічних розв'язків, вдалося викристалізувати низку визначальних чинників, котрі незаперечно підтверджують нагальність та доречність імплементації мікросервісу Access Guard:

1. Вузькопрофільне спрямування на мікросервісну екосистему: на противагу всеосяжним універсальним платформам, Access Guard конструюється виключно для середовища мікросервісів, враховуючи найдрібніші нюанси та особливості їхньої взаємодії;

2. Ощадливість ресурсоспоживання: архітектурна концепція, що ґрунтується на реактивній парадигмі, уможливлює високу продуктивність за низького споживання обчислювальних потужностей.

3. Безшовне злиття з корпоративною інфраструктурою: природжена сумісність із корпоративними засобами комунікації (зокрема, Mattermost) та різноманітними бізнес-системами;

4. Безпрецедентна пластичність та всеосяжний контроль: цілковите панування над кодовим підґрунтям, змога видозмінювати функціонал відповідно до будь-яких організаційних запитів без жодних залежностей від сторонніх постачальників послуг;

5. Витончена аналітика доступу: прискіпливе документування хронології входів та всіх спроб отримання доступу задля посилення безпекового аудиту;

6. Консолідований осередок моніторингу: зосередження керування доступом до цілої низки корпоративних застосунків через єдиний уніфікований інтерфейс;

7. Довершена взаємодія з користувачем: спеціально розроблений інтерфейсний компонент для ефективного та інтуїтивно зрозумілого адміністрування доступу;

8. Самобутні механізми верифікації особи: симбіоз різноманітних методик автентифікації, включно з інтеграцією корпоративних месенджерів для двофакторного підтвердження.

Отже, розробка мікросервісу Access Guard постає як своєчасна й обміркована відповідь на виняткові запити корпоративних екосистем із мікросервісною архітектурою, пропонуючи витончене та всебічне рішення для централізованого адміністрування доступу в комплексних інформаційних ландшафтах сучасних підприємств.

1.3 Опис IT-засобів для розробки Access Guard

1.3.1 Бекенд-технології

Java та Spring Framework

Рис. 1.4 Логотип Java

Рис. 1.5 Логотип Spring Framework

Основою для розробки бекенду Access Guard обрано мову програмування Java (рисунок 1.4) та фреймворк Spring Boot [4] (рисунок 1.5), що надають такі переваги:

- стабільність та надійність у корпоративному середовищі;
- багата екосистема компонентів та інтеграцій;
- широкі можливості для конфігурації та кастомізації;
- потужна підтримка спільноти та документації.

Spring WebFlux та реактивне програмування

Рис. 1.6 Логотип Spring WebFlux

На відміну від традиційного Spring MVC, для Access Guard обрано реактивний підхід із використанням Spring WebFlux (рисунок 1.6):

— неблокуюча обробка запитів для вищої пропускної здатності;
— ефективне використання ресурсів сервера;
— асинхронна обробка для покращеної відгукоздатності;
— підтримка реактивних потоків (Mono, Flux) для обробки даних.

Spring Security

Рис. 1.7 Логотип Spring Security

Для базових механізмів безпеки кастомними розширеннями [5] (рисунок 1.7):

— фреймворк для автентифікації та авторизації;
— інтеграція з JWT для безсесійної автентифікації;
— захист від поширених веб-вразливостей;
— гнучка конфігурація правил безпеки.

Spring Cloud

Рис. 1.8 Логотип Spring Cloud з мікросервісами

Компоненти Spring Cloud забезпечують інтеграцію з мікросервісною екосистемою (рисунок 1.8):

— eureka для реєстрації та виявлення сервісів
— gateway для маршрутизації запитів
— config для централізованої конфігурації
— circuit Breaker для підвищення відмовостійкості

1.3.2 Бази даних та зберігання

PostgreSQL

Рис. 1.9 Логотип PostgreSQL

Як основне сховище даних обрано PostgreSQL (рисунок 1.9):

— надійна реляційна база даних з відкритим кодом;
— потужна підтримка транзакцій та обмежень цілісності;
— широкі можливості для індексації та оптимізації запитів;
— розширений функціонал для зберігання та пошуку JSON-даних.

Spring Data JPA

Рис. 1.10 Логотип Spring Data JPA

Для роботи з базою даних використовуються:

— spring Data JPA (рисунок 1.10)— для декларативного доступу до даних;

— hibernate — як реалізація JPA для об'єктно-реляційного відображення.

Hibernate Envers

Рис. 1.11 Логотип роботи Hibernate Envers

Для забезпечення аудиту змін використовується Hibernate Envers:

— автоматичне відстеження змін у ключових таблицях;

— зберігання історії модифікацій з часовими мітками;

— можливість відновлення попередніх станів об'єктів.

1.3.3 Комунікація та інтеграція

RabbitMQ

Рис. 1.12 Логотип RabbitMQ

Для асинхронної комунікації між мікросервісами використовується RabbitMQ (рисунок 1.12):

— брокер повідомлень для надійної асинхронної комунікації;

— підтримка різних моделей обміну повідомленнями;

— гарантоване доставлення повідомлень;

— висока продуктивність та масштабованість.

WebClient

Рис. 1.13 Логотип роботи WebClient

Для синхронної взаємодії з іншими сервісами використовується реактивний WebClient (рисунок 1.13):

— неблокуюча HTTP-комунікація;

— підтримка різних форматів даних (JSON, XML);

— гнучкі можливості налаштування запитів та обробки відповідей.

JSON Web Token (JWT)

JWT

JSON Web Tokens

Рис. 1.14 JWT токен

Для реалізації безсесійної автентифікації обрано стандарт JWT [6]:

— компактний та самодостатній формат для передачі інформації між сторонами;

— можливість включення метаданих про користувача та його права;

— криптографічний захист від підробки;

— широка підтримка в різних мовах та фреймворках.

1.3.4 Фронтенд-технології

React та TypeScript

Рис. 1.15 Логотип React

Рис. 1.16 Логотип TypeScript

Для розробки користувацького інтерфейсу використовуються:

— react — бібліотека для побудови інтерактивних UI-компонентів;

— typeScript — типізована надмножина JavaScript для підвищення якості коду.

Material UI

Рис. 1.17 Логотип Material UI

Як UI-фреймворк обрано Material UI (рисунок 1.17):

— готові компоненти з дизайном у стилі Material Design

— широкі можливості для кастомізації

— підтримка темної теми та адаптивного дизайну

React Query та Redux

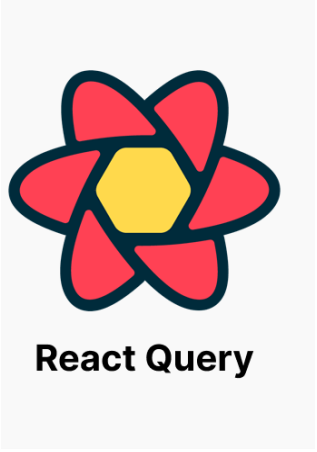


Рис. 1.18 Логотип React Query

Рис. 1.19 Логотип Redux

Для управління станом застосунку використовуються:

— redux — для глобального стану додатку (рисунок 1.19);

— react Query — для управління станом серверних даних та кешування.

AG Grid

Рис. 1.20 Логотип AG Grid

Для відображення табличних даних використовується AG Grid:

— високопродуктивна таблична бібліотека;

— підтримка сортування, фільтрації, пагінації;

— можливість кастомізації відображення комірок.

1.3.5 Інструменти розробки та розгортання

Docker та Docker Compose

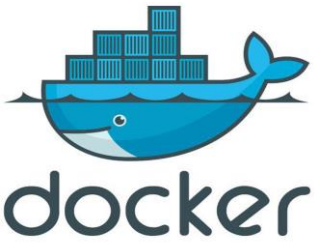


Рис. 1.21 Логотип Docker

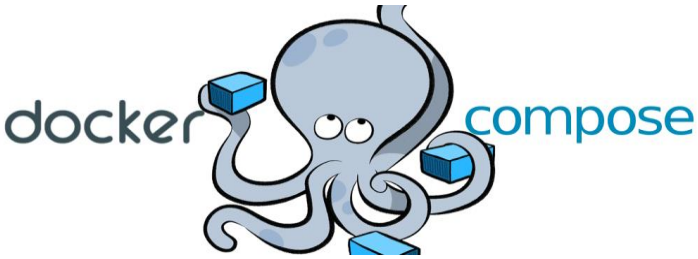


Рис. 1.22 Логотип Docker Compose

Для контейнеризації та розгортання використовуються:

— docker — для створення ізольованих контейнерів із додатками;

— docker Compose — для оркестрації мультиконтейнерних додатків.

Maven

Рис. 1.23 Логотип Maven

Як інструмент збірки для Java-проектів використовується Maven:

— управління залежностями;

— автоматизація процесу збірки;

— стандартизований життєвий цикл проекту.

Swagger/OpenAPI

Рис. 1.24 Логотип Swagger/OpenAPI

Для документування API використовується Swagger/OpenAPI (рисунок 1.24):

— автоматична генерація документації API;

— інтерактивний інтерфейс для тестування ендпоінтів;

— підтримка різних форматів документації.

Grafana та Prometheus

Рис. 1.25 Логотип Grafana та Prometheus

Для моніторингу та збору метрик використовуються:

— prometheus — для збору та зберігання метрик;

— grafana — для візуалізації метрик через інтерактивні дашборди;

— loki — для централізованого збору логів;

— zipkin — для трасування розподілених запитів.

Цей технологічний стек забезпечує всі необхідні інструменти для розробки, розгортання та підтримки мікросервісу Access Guard, надаючи оптимальний баланс між продуктивністю, надійністю, безпекою та зручністю розробки.

1.4 Визначення об'єкту, предмету, мети та задач роботи

1.4.1 Об'єкт дослідження

Об'єктом наукового розвідування є багатовимірний процес верифікації особистості та легітимації доступу користувачів у децентралізованому просторі мікросервісної будови корпоративних застосунків, що охоплює розмаїття механізмів валідації тотожності, регулювання привілеїв доступу та забезпечення недоторканності інформаційних набутків.

1.4.2 Предмет дослідження

Предметною сферою пізнавальних зусиль виступають методологічні підходи, інструментальні засоби та технологічні парадигми втілення централізованого мікросервісу дозвільного адміністрування з використанням реактивної архітектури, котрий забезпечуватиме єдиний вузол контролювання автентифікаційних та авторизаційних процесів для цілісної сукупності складників розосередженої системи.

1.4.3 Мета роботи

Метою кваліфікаційної роботи є створення високопотужного, пластично-розширюваного та надійно-захищеного мікросервісу Access Guard для унітарного керування доступами до корпоративних програмних комплексів у середовищі мікросервісної архітектури, що уможливить спрощення процедур

адміністративного налаштування прав і запровадження єдиної контрольної точки для ревізії та спостереження за користувацькою активністю.

1.4.4 Завдання Кваліфікаційної Роботи

1. Аналіз існуючих рішень для управління доступом та визначення функціональних вимог до системи;
2. Проектування архітектури мікросервісу та створення інформаційної моделі бази даних;
3. Реалізація JWT автентифікації з HTTP-only cookies та механізмом оновлення токенів;
4. Розробка REST API сервісу та адміністративного веб-інтерфейсу;
5. Інтеграція з Mattermost для двофакторної автентифікації та автентифікації пристроїв;
6. Контроль доступу на основі ролей та управління сесіями [10];
7. Створення комплексної системи тестування та контейнеризація через Docker;
8. Налаштування моніторингу (Prometheus, Grafana) та документування API.

1.5 Функціональні та нефункціональні вимоги до програмного забезпечення

1.5.1 Функціональні вимоги

1. Створення, редагування та видалення користувачів;
2. Контроль доступу на основі ієрархії ролей користувачів;
3. Блокування/розблокування облікових записів;
4. Автентифікація через логін/пароль з JWT-токенами та HTTP-only cookies;
5. Двофакторна автентифікація через Mattermost;
6. Автентифікація пристроїв користувачів;

7. Генерація, оновлення та управління API-ключами для міжсервісної автентифікації;
8. Оновлення токенів через refresh-токени;
9. Обмеження доступу до окремих додатків;
10. Реєстрація та управління додатками в системі;
11. Підтримка різних типів додатків (INTERNAL, EXTERNAL);
12. Здійснення аудиту змін у ключових таблицях бази даних;
13. Обмеження кількості одночасних сесій користувача;
14. Автоматична реєстрація нових сервісів через Discovery Server;
15. Асинхронна комунікація через RabbitMQ;
16. Веб-інтерфейс для управління користувачами та їх правами.

1.5.2 Нефункціональні вимоги

1. Криптографічне шифрування паролів за допомогою BCrypt;
2. Використання HTTPS для всіх комунікацій;
3. Захист від CSRF, XSS та інших типів атак;
4. Автоматичне знищення токенів після закінчення сесії;
5. Маскування паролів та чутливих даних у логах системи;
6. Обробка запитів автентифікації з латентністю не більше 200 мс;
7. Підтримка до 1000 одночасних активних сесій;
8. Горизонтальне масштабування через автоматичне створення додаткових Docker-інстансів при збільшенні навантаження;
9. Доступність на рівні 99%;
10. Автоматичне відновлення після збоїв;
11. Механізми повторних спроб при тимчасових проблемах зв'язку;
12. Детальна документація API через Swagger/OpenAPI;
13. Багаторівневе логування з градацією важливості подій;
14. Підтримка моніторингових інструментів (Prometheus, Grafana, Loki);
15. Трасування розподілених запитів через Zipkin;
16. Підтримка української та англійської мов інтерфейсу.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ МІКРОСЕРВІСУ ACCESS GUARD

2.1 Огляд архітектурних підходів до розробки мікросервісних систем

2.1.1 Принципи мікросервісної архітектури

Сучасні корпоративні інформаційні системи дедалі частіше базуються на мікросервісній архітектурі, що передбачає декомпозицію монолітних додатків на незалежні, вузькоспеціалізовані компоненти. Цей підхід характеризується низкою принципових особливостей, які суттєво відрізняють його від традиційних архітектурних рішень [7] (таблиця 2.1).

Ключові принципи мікросервісної архітектури:

1. *Функціональна декомпозиція* забезпечує високу когезію всередині сервісу та слабке зчеплення між окремими сервісами, що, своєю чергою, полегшує розробку, тестування та підтримку кожного елемента системи.

2. *Ізольованість розгортання* мікросервісів значно прискорює цикл розробки програмного забезпечення. У контексті системи автентифікації та авторизації це дозволяє оперативно впроваджувати оновлення безпекових механізмів без необхідності перезапуску всього комплексу додатків.

3. *Технологічна гетерогенність* надає можливість використовувати оптимальні технології для вирішення конкретних завдань. Втім, у корпоративному середовищі при розробці взаємопов'язаних сервісів, доцільно дотримуватися певної технологічної уніфікації, що спрощує підтримку та забезпечує єдину модель безпеки.

4. *Організаційна автономність* мікросервісних команд відображає конвергенцію архітектурних та управлінських практик. Такий підхід підвищує мотивацію та відповідальність, але, водночас, потребує ефективної координації між командами, особливо у сфері безпеки та міжсервісної взаємодії.

Таблиця 2.1

Порівняння монолітної та мікросервісної архітектури

Характеристика	**Моноліт**	**Мікросервіси**
Структура	Єдина цілісна система	Розподілена система з незалежних компонентів
Масштабування	Вертикальне (збільшення потужності)	Горизонтальне (збільшення кількості екземплярів)
Стійкість	Відмова одного компонента зупиняє всю систему	Ізольовані відмови з мінімальним впливом
Розробка	Єдина кодова база	Розподілена розробка незалежними командами
Технології	Єдиний стек технологій	Можливість використання різних технологій

У контексті системи управління доступом, мікросервісна архітектура висуває специфічні вимоги до проектування механізмів автентифікації та авторизації, що повинні забезпечувати:

1. Централізоване управління ідентифікацією користувачів;
2. Уніфікований механізм верифікації доступу до розподілених ресурсів;
3. Гомогенне застосування політик безпеки;
4. Ефективне масштабування відповідно до зростання навантаження.

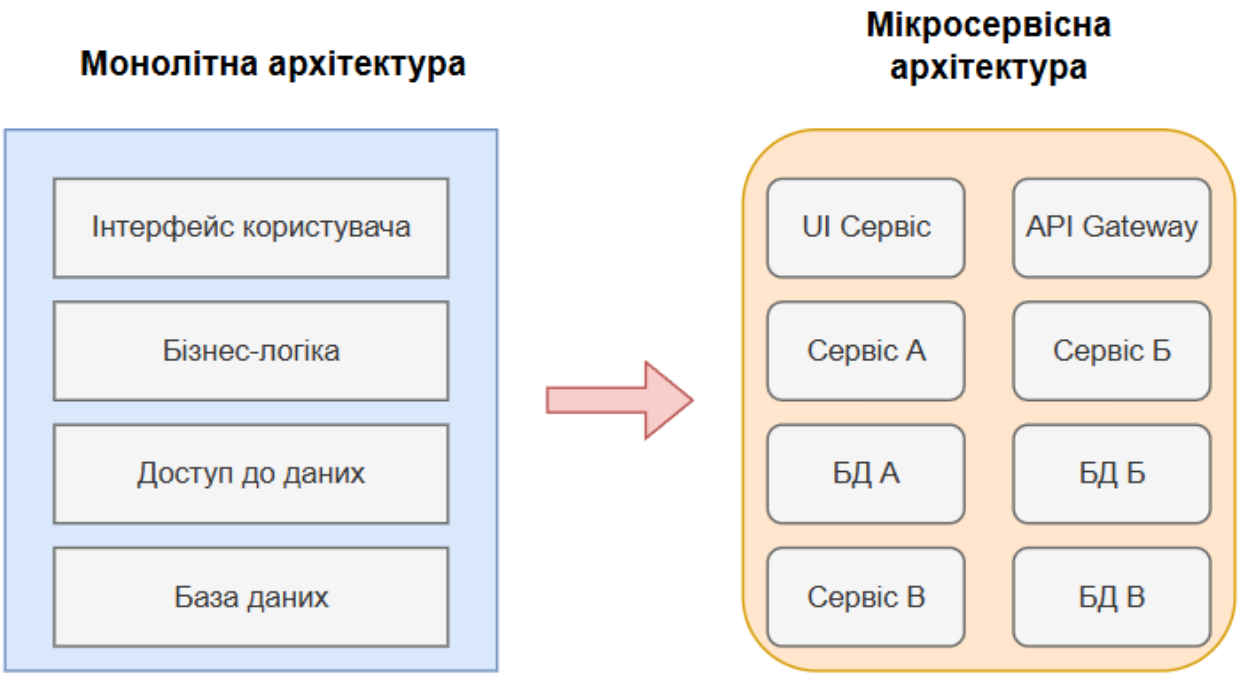


Рис. 2.1 Порівняння монолітної та мікросервісної архітектури

2.1.2 Особливості проектування систем безпеки в мікросервісному середовищі

Проектування систем безпеки [8] для мікросервісної архітектури вимагає принципово іншого підходу порівняно з монолітними системами. Традиційні моделі безпеки, що ґрунтуються на периметровому захисті, виявляються недостатньо ефективними в умовах розподілених систем, де кожен сервіс потенційно може стати вразливою точкою входу.

Ключові виклики безпеки в мікросервісному середовищі:

1. Розподілена автентифікація — необхідність верифікації користувачів у багатьох точках системи;
2. Управління ідентичністю — консистентне поширення інформації про користувачів між сервісами;
3. Міжсервісна комунікація — забезпечення безпеки взаємодії між сервісами;
4. Захист персональних даних — контроль за розподіленим зберіганням чутливої інформації;

5. Моніторинг та аудит — централізований збір та аналіз подій безпеки.

Особливої актуальності набуває парадигма "нульової довіри" (*Zero Trust*), згідно до якої кожен запит, незалежно від джерела походження, підлягає верифікації. Система Access Guard проектується з урахуванням цієї парадигми, забезпечуючи механізми для автентифікації та авторизації кожного запиту на доступ до захищених ресурсів.

Децентралізована природа мікросервісів породжує комплексні виклики у сфері управління ідентифікацією користувачів (рис 2.2). Ключовою проблемою є необхідність консистентного поширення інформації про користувачів, їхні ролі та права доступу між множиною взаємодіючих сервісів. Системи розповсюдження стану користувача повинні забезпечувати баланс між швидкістю оновлення інформації та навантаженням на інфраструктуру.

Архітектурні рішення для безпеки мікросервісів:

1. Токен-орієнтована автентифікація — використання JWT для інкапсуляції інформації про користувача та його права;
2. Централізований сервіс автентифікації — єдина точка управління доступом;
3. API-ключі для міжсервісної комунікації — специфічні механізми авторизації сервісів;
4. Шифрування транспортного рівня — захист комунікацій через HTTPS;
5. Централізований аудит — єдина система збору та аналізу подій безпеки.

Автентифікація у мікросервісному середовищі вимагає використання безсесійних механізмів, оскільки необхідність верифікації сеансових даних на багатьох серверах призводить до значних накладних витрат. Архітектурно обґрунтованим рішенням є впровадження токен-орієнтованих підходів, зокрема

із застосуванням JWT (JSON Web Token), що дозволяє інкапсулювати інформацію про користувача та його повноваження безпосередньо у токені.

Авторизація запитів у розподіленій системі вимагає не лише перевірки прав доступу користувача, але й верифікації автентичності міжсервісних комунікацій. Комплексний підхід передбачає диференційовані механізми авторизації для різних типів суб'єктів – кінцевих користувачів, сервісів та зовнішніх систем.

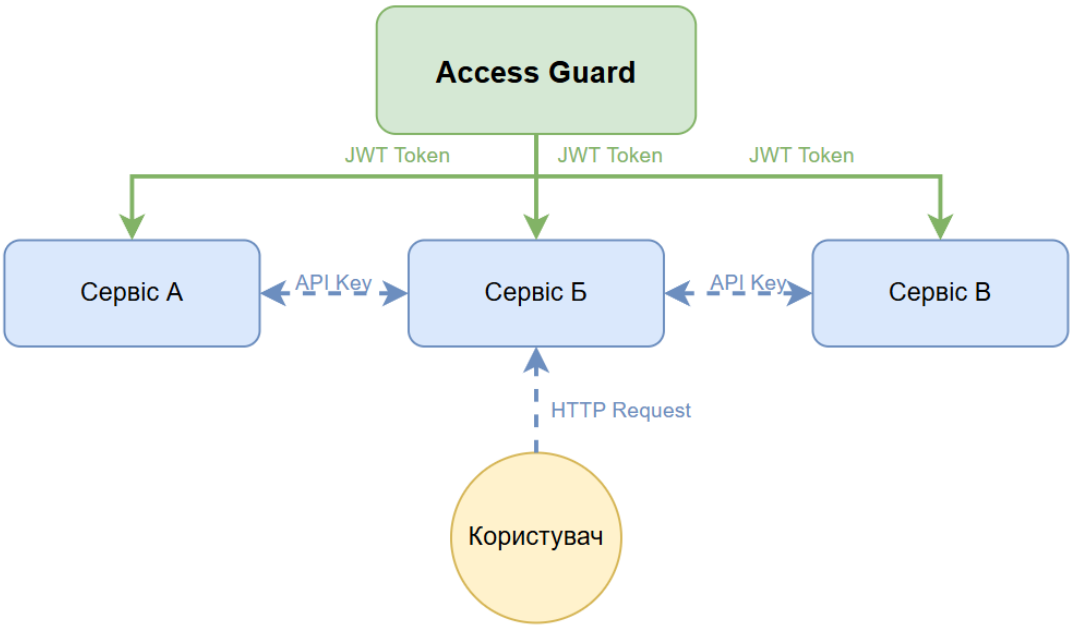


Рис. 2.2 Схема безпеки мікросервісів

2.1.3 Обґрунтування вибору реактивної архітектури

Реактивна архітектура являє собою парадигматичний підхід до розробки систем, орієнтований на забезпечення чотирьох фундаментальних властивостей: відгукоздатності, стійкості, еластичності та асинхронності повідомлень. У контексті проектування системи управління доступом для мікросервісного середовища, реактивний підхід демонструє значні переваги, особливо з урахуванням специфіки безпекових сервісів.

Ключові властивості реактивної архітектури:

1. *Відгукоздатність* системи автентифікації та авторизації має критичне значення для користувацького досвіду, оскільки процедури верифікації

інтегровані в кожну взаємодію з системою. Традиційні блокуючі підходи з синхронною обробкою запитів створюють ризик каскадних затримок при зростанні навантаження (таблиця 2.2). Реактивна архітектура, заснована на неблокуючій обробці подій, дозволяє підтримувати стабільні показники латентності автентифікації навіть за умов пікового навантаження (рисунок 2.3);

2. *Стійкість до відмов* є імперативною вимогою для безпекових систем, адже недоступність сервісу автентифікації призводить до паралізації всієї інформаційної інфраструктури. Реактивний підхід передбачає ізоляцію відмов та впровадження механізмів відновлення, що забезпечують безперервність функціонування критичних компонентів;

3. *Еластичність* — здатність системи адаптуватися до змінного навантаження — набуває особливої ваги для сервісів автентифікації, характерною особливістю яких є нерівномірний розподіл запитів протягом робочого дня. Реактивна архітектура з її асинхронною обробкою подій та неблокуючим введенням/виведенням забезпечує ефективне використання обчислювальних ресурсів при змінних навантаженнях;

4. *Асинхронний обмін повідомленнями* є структурним фундаментом реактивної архітектури, що забезпечує слабке зчеплення компонентів системи. Для мікросервісу автентифікації це особливо актуально в контексті оповіщення інших сервісів про зміни в правах доступу користувачів або блокування облікових записів.

Таблиця 2.2

Порівняння реактивної та блокуючої архітектури

Характеристика	**Блокуюча архітектура**	**Реактивна архітектура**
Обробка запитів	Синхронна, блокуюча	Асинхронна, неблокуюча
Модель потоків	1 потік на запит	Кілька запитів на потік
Ефективність використання ресурсів	Низька при I/O-операціях	Висока

Продовження таблиця 2.2

Порівняння реактивної та блокуючої архітектури

Характеристика	Блокуюча архітектура	Реактивна архітектура
Масштабованість	Обмежена кількістю потоків	Висока, обмежена лише ресурсами
Складність реалізації	Низька	Вища, потребує спеціальних підходів
Обробка помилок	Проста (try-catch)	Складніша (монади)

Вибір реактивного підходу [9] для реалізації Access Guard обґрунтовується також технічними особливостями проекту. Застосування Spring WebFlux як реактивного фреймворку дозволяє досягти значно вищих показників пропускної здатності порівняно з традиційним Spring MVC. Цей аспект критично важливий для сервісу автентифікації, який обробляє більшість запитів у системі.

Необхідно відзначити, що реактивна парадигма накладає певні обмеження на розробку та підвищує когнітивну складність програмування. Однак, для критично важливих компонентів інфраструктури, таких як система управління доступом, переваги у продуктивності та стійкості перевищують витрати на додаткову складність реалізації.

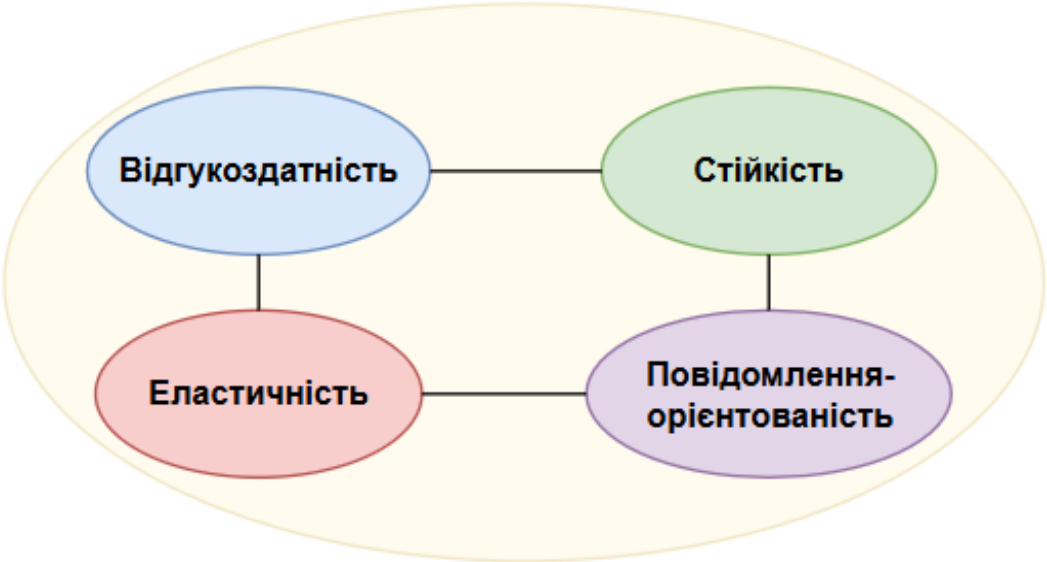


Рис. 2.3 Модель реактивної архітектури

Переваги в мікросервісній архітектурі:

- висока пропускна здатність при менших ресурсах;
- стабільний час відгуку навіть при пікових навантаженнях;
- ефективна обробка міжсервісних комунікацій;
- природна інтеграція з асинхронними повідомленнями;
- кращі механізми відмовостійкості.

2.2 Концептуальна архітектура системи Access Guard

2.2.1 Загальна структура мікросервісу

Мікросервіс Access Guard, що виконує роль централізованого шлюзу безпеки, характеризується багаторівневою архітектурою, структурованою згідно з принципами чіткого розмежування відповідальності. Кожен архітектурний шар забезпечує специфічну функціональність та взаємодіє з іншими шарами через чітко визначені інтерфейси.

Таблиця 2.3

Архітектурний шар та його компонент

Архітектурний шар	**Компоненти**	**Призначення та функціональність**
Презентаційний шар	AuthController	Керує автентифікацією користувачів, включаючи різні механізми входу, управління токенами та реєстрацію
	UserController	Забезпечує управління користувачами, включаючи створення, редагування та видалення профілів
	ApplicationController	Керує застосунками в системі, включаючи реєстрацію, API-ключі та залежності

Продовження таблиці 2.3

Архітектурний шар та його компонент

Архітектурний шар	**Компоненти**	**Призначення та функціональність**
	DeviceController	Забезпечує реєстрацію та управління пристроями користувачів
	SettingController	Керує системними налаштуваннями безпеки
	UserApplicationController	Забезпечує управління зв'язками між користувачами та застосунками
Сервісний шар	AuthService	Реалізує механізми автентифікації (логін/пароль, Mattermost, пристрої)
	UserService	Забезпечує повний життєвий цикл користувача в системі
	ApplicationService	Керує застосунками, їх реєстрацією, залежностями та доступними функціями
	TokenService	Забезпечує генерацію, валідацію та управління токенами автентифікації
	DeviceService	Керує пристроями користувачів та їх автентифікацією
	MattermostAuthService	Забезпечує інтеграцію з Mattermost для двофакторної автентифікації

Таблиця 2.3

Архітектурний шар та його компонент

Архітектурний шар	**Компоненти**	**Призначення та функціональність**
Доменний шар	User	Представляє користувача з його автентифікаційними параметрами
	Application	Описує застосунок в системі з метаданими та API-ключами
	Token	Інкапсулює інформацію про токен оновлення

Продовження таблиці 2.3

Архітектурний шар та його компонент

Архітектурний шар	**Компоненти**	**Призначення та функціональність**
	Device	Представляє пристрій користувача для автентифікації
	UserApplication	Визначає зв'язок між користувачем і застосунком
	Setting	Містить системні налаштування безпеки
	UserVisitStatistics	Зберігає статистику відвідувань для аудиту
Персистентний шар	UserRepository	Забезпечує доступ до користувачів з підтримкою складних запитів
	ApplicationRepository	Надає функціональність для роботи з застосунками
	TokenRepository	Відповідає за управління токенами
	DeviceRepository	Забезпечує операції з пристроями
Інфраструктурний шар	SecurityConfig	Конфігурація безпеки системи
	AOP-компоненти	Забезпечують наскрізні завдання – наприклад, логування та моніторинг
	ErrorHandlers	Забезпечують уніфіковану обробку виключень
	Utilities	Утиліти для спільних функцій

Архітектурною особливістю системи Access Guard є комбінування блокуючого та неблокуючого стилів програмування. Контролери використовують реактивні типи (Mono<T> та Flux<T>) для забезпечення неблокуючої обробки запитів, тоді як внутрішні сервіси поєднують блокуючі операції для взаємодії з базою даних та неблокуючі патерни для зовнішніх інтеграцій. Така гібридна архітектура забезпечує оптимальний баланс між

продуктивністю та зручністю розробки, дозволяючи системі ефективно обробляти велику кількість запитів автентифікації з мінімальною затримкою.

2.2.2 Взаємодія з іншими компонентами системи (User Hub, Gateway, Discovery Server)

Мікросервіс Access Guard функціонує у складному інтеграційному ландшафті, взаємодіючи з іншими ключовими компонентами мікросервісної архітектури. Ефективна інтеграція між цими сервісами є критично важливою для забезпечення цілісності системи безпеки .

User Hub виступає центральним реєстром користувачів, що зберігає розширену інформацію про них, їхні ролі, організаційну структуру та зв'язки з бізнес-сутностями. Взаємодія між Access Guard та User Hub характеризується двонаправленою інтеграцією:

1. *Синхронізація даних користувачів:* Access Guard періодично отримує оновлену інформацію про користувачів через REST API User Hub, забезпечуючи актуальність автентифікаційних даних;
2. *Оновлення автентифікаційної інформації:* При зміні критичних параметрів користувача в Access Guard (наприклад, блокування облікового запису), ця інформація асинхронно передається до User Hub через RabbitMQ;
3. *Розширення профілю користувача:* При автентифікації Access Guard збагачує базову інформацію про користувача додатковими даними з User Hub для формування повного контексту авторизації.

Архітектурно значущим рішенням є використання різних механізмів комунікації: синхронних REST-запитів для критичних операцій, що вимагають миттєвої консистентності, та асинхронних повідомлень через брокер для оповіщення про зміни, що не потребують негайної реакції.

Gateway виступає єдиною точкою входу для клієнтських запитів до всієї мікросервісної архітектури. Взаємодія між Gateway та Access Guard реалізує централізовану модель автентифікації та авторизації:

1. *Верифікація токенів:* Gateway перенаправляє запити на автентифікацію до Access Guard через спеціалізований ендпоінт, який верифікує токени та повертає інформацію про користувача;

2. *Перевірка авторизації:* Для кожного запиту до захищеного ресурсу Gateway консультується з Access Guard для підтвердження прав доступу користувача до конкретного ресурсу;

3. *Оновлення токенів:* Gateway взаємодіє з Access Guard для оновлення токенів доступу, коли їх термін дії закінчується, забезпечуючи безперервний користувацький досвід.

Технічно взаємодія реалізується через реактивний WebClient, що дозволяє обробляти велику кількість паралельних запитів без блокування потоків виконання. Для підвищення відмовостійкості реалізовано механізм повторних спроб з експоненціальною затримкою при тимчасових проблемах зв'язку.

Discovery Server (Eureka) забезпечує динамічне виявлення сервісів у мікросервісній архітектурі. Взаємодія Access Guard з Discovery Server має критичне значення для побудови надійної та масштабованої системи (рис 2.4):

1. *Реєстрація сервісу:* Access Guard реєструється в Discovery Server при запуску, надаючи інформацію про свою адресу, порт та метадані;

2. *Виявлення інших сервісів:* Access Guard використовує Discovery Server для знаходження інших сервісів, з якими він взаємодіє, зокрема User Hub;

3. *Автоматична реєстрація нових сервісів:* Discovery Server повідомляє Access Guard про появу нових сервісів, що дозволяє автоматично надавати їм необхідні API-ключі для автентифікації

Особливо цікавим архітектурним рішенням є механізм автоматичної інтеграції нових сервісів з системою безпеки. Коли новий мікросервіс реєструється в Discovery Server, ця подія перехоплюється спеціальним слухачем, який надсилає інформацію про новий сервіс до Access Guard через RabbitMQ. Access Guard автоматично реєструє новий сервіс у своїй системі безпеки, генерує

для нього API-ключ та налаштовує базові права доступу.

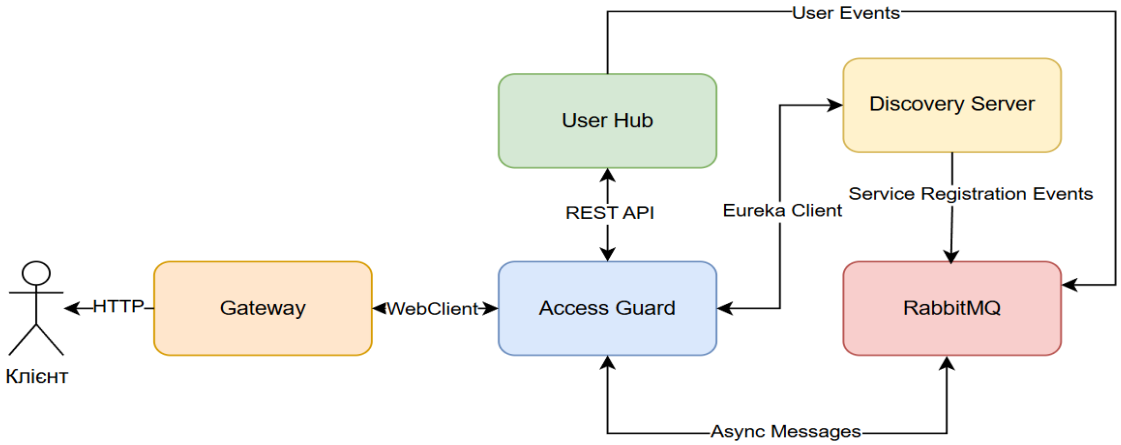


Рис. 2.4 Діаграма взаємодії компонентів системи

2.2.3 Проектування механізмів асинхронної взаємодії через RabbitMQ

Асинхронна взаємодія є критично важливим аспектом архітектури мікросервісів, що забезпечує слабке зчеплення компонентів та підвищує стійкість системи до відмов. У контексті системи управління доступом, де зміни стану користувачів та їх прав повинні ефективно поширюватися між різними сервісами, асинхронний обмін повідомленнями набуває особливої значущості. RabbitMQ обрано як брокер повідомлень для реалізації асинхронної комунікації між мікросервісами. Це рішення обґрунтовується наступними перевагами:

1. Підтримка різноманітних шаблонів обміну повідомленнями (publish-subscribe, request-reply, work queues);
2. Гарантована доставка повідомлень та можливість налаштування підтверджень;
3. Вбудовані механізми балансування навантаження між споживачами повідомлень;
4. Висока пропускна здатність та низька латентність обробки повідомлень;
5. Інтеграція з екосистемою Spring через Spring AMQP.

Архітектура асинхронної взаємодії базується на концепції топіків (exchanges) та черг. Для кожного типу подій в системі визначено окремий топік, що забезпечує тематичну сегрегацію повідомлень та можливість селективної підписки на події.

Основні топіки, використані у системі:

- *user-exchange:* для повідомлень, пов'язаних зі змінами стану користувачів;
- *application-exchange:* для повідомлень про зміни в додатках та правах доступу.

Кожен мікросервіс підписується на відповідні черги, сформовані за шаблоном *{service-name}.{entity-type}*, наприклад:

- *access-guard.user:* черга для повідомлень про користувачів, адресованих Access Guard;
- *user-hub.application:* черга для повідомлень про додатки, адресованих User Hub.

Повідомлення структуруються згідно з заздалегідь визначеними DTO-класами, що забезпечують типобезпечність та структурну валідацію:

- *UserBrokerDTO:* для передачі інформації про зміни в користувачах;
- *ApplicationBrokerDTO:* для передачі інформації про зміни в додатках.

Кожне повідомлення супроводжується метаданими, що визначають тип дії (*RabbitMQAction*):

- *SAVE:* створення нової сутності;
- *UPDATE:* оновлення існуючої сутності;
- *DELETE:* видалення сутності.

Конфігурація RabbitMQ у Access Guard реалізована через клас *RabbitMQConfig*, який визначає:

- топіки для публікації повідомлень;
- черги для прийому повідомлень;
- налаштування зв'язків між топіками та чергами;
- конфігурацію серіалізації/десеріалізації повідомлень.

Для обробки вхідних повідомлень реалізовано спеціалізовані слухачі:

- *UserMessageListener:* обробляє повідомлення про зміни користувачів;
- *ApplicationMessageListener:* обробляє повідомлення про зміни в додатках.

Аспект надійності забезпечується через:

- налаштування підтверджень доставки повідомлень;
- механізми повторної обробки при збоях;
- збереження непроцесених повідомлень у спеціальні черги для подальшого аналізу.

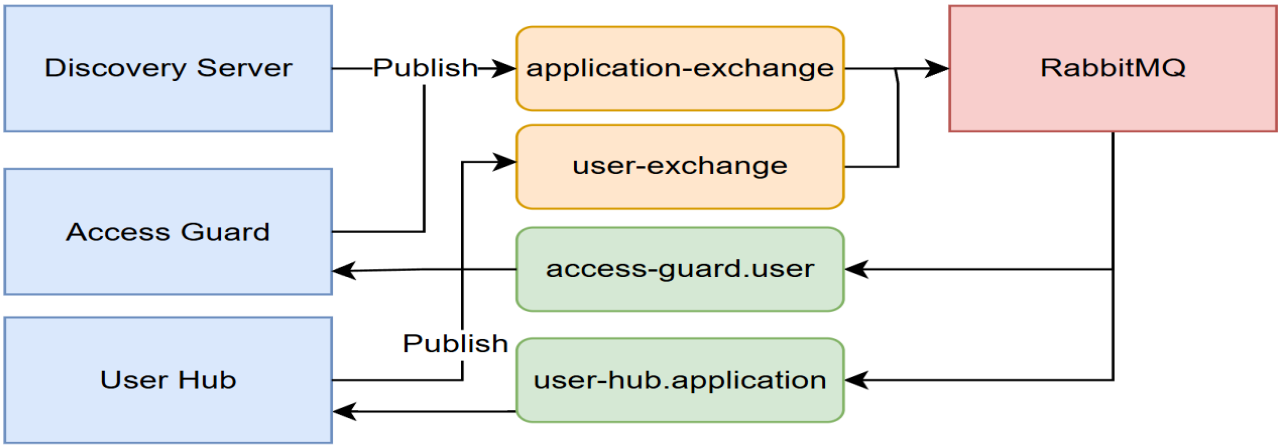


Рис. 2.5 Діаграма асинхронної взаємодії через RabbitMQ

2.3 Проектування інформаційної моделі

2.3.1 Аналіз доменних сутностей та реляційна модель даних

Інформаційна модель системи Access Guard формується на основі ретельного аналізу доменної області управління доступом. Процес моделювання ґрунтується на виявленні ключових бізнес-сутностей, їхніх атрибутів та взаємозв'язків, що забезпечують реалізацію вимог до системи безпеки. Для кожної доменної сутності розроблено відповідну структуру таблиці в реляційній базі даних, оптимізовану з урахуванням вимог до продуктивності, цілісності даних та масштабованості системи.

Сутність User (Користувач): центральною сутністю системи, що представляє суб'єкта автентифікації та авторизації. Модель користувача інкапсулює як базові автентифікаційні атрибути (ім'я користувача, пароль), так і контекстну інформацію, необхідну для різноманітних механізмів автентифікації та контролю доступу. У контексті системи безпеки модель користувача розроблена з урахуванням потреби в багаторівневій автентифікації та підтримці різних методів верифікації ідентичності (таблиця 2.4).

Таблиця 2.4

Структура таблиці user_auth

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор користувача
username	VARCHAR(255)	NOT NULL, UNIQUE	Унікальне ім'я користувача для входу
password	VARCHAR(255)		Зашифрований пароль користувача
role	VARCHAR(50)	NOT NULL	Роль користувача в системі
max_sessions	INTEGER		Максимальна кількість одночасних сесій

Продовження таблиці 2.4

Структура таблиці user_auth

Поле	Тип даних	Обмеження	Опис
duration_of_life_access_token	BIGINT		Індивідуальна тривалість життя access токену (хв)
duration_of_life_refresh_token	BIGINT		Індивідуальна тривалість життя refresh токену (хв)
mattermost_id	VARCHAR(255)		Ідентифікатор користувача в Mattermost
confirm_code	VARCHAR(20)		Код підтвердження для двофакторної автентифікації
count_of_total_requests	BIGINT	DEFAULT 0	Загальна кількість запитів користувача
last_active_at	TIMESTAMP		Час останньої активності користувача
device_auth_enabled	BOOLEAN		Індикатор дозволу автентифікації за пристроєм
created_at	TIMESTAMP	CURRENT_TIMESTAMP	Час створення запису

Сутність Application (Додаток): моделює застосунки, які інтегруються з системою безпеки Access Guard. Ця сутність є центральним елементом для реалізації міжсервісної автентифікації та контролю доступу користувачів до різних компонентів системи. Модель додатку підтримує ієрархічні залежності між застосунками, що дозволяє реалізувати каскадне управління доступом в екосистемі мікросервісів (таблиця 2.5).

Таблиця 2.5

Структура таблиці application

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор додатку
name	VARCHAR(255)	NOT NULL	Назва додатку
spring_application_name	VARCHAR(255)	NOT NULL, UNIQUE	Ідентифікатор додатку в Spring
api_key	VARCHAR(255)		API ключ для міжсервісної автентифікації
description	TEXT		Опис додатку
url	VARCHAR(1000)		URL-адреса доступу до додатку
img	BYTEA		Зображення-іконка додатку
type	VARCHAR(50)	NOT NULL	Тип додатку (INTERNAL/EXTERNAL)
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час створення запису
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час останнього оновлення запису

Сутність Token (Токен): представляє refresh-токен, призначений для оновлення сеансів користувачів без повторної автентифікації. Ця сутність підтримує механізм керування сеансами та забезпечує безперервний користувацький досвід (таблиця 2.6).

Таблиця 2.6

Структура таблиці refresh_token

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор токену
token	VARCHAR(1000)	NOT NULL	Значення токену оновлення

Продовження таблиці 2.6

Структура таблиці refresh_token

Поле	Тип даних	Обмеження	Опис
user_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на користувача
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час створення токену

Сутність Device (Пристрій): моделює фізичні пристрої, з яких користувачі здійснюють доступ до системи. Ця сутність забезпечує додатковий рівень безпеки через прив'язку автентифікації до конкретних пристроїв.

Таблиця 2.7

Структура таблиці device

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор пристрою
device_id	VARCHAR (255)	NOT NULL, UNIQUE	Унікальний ідентифікатор пристрою в системі
name	VARCHAR (255)		Назва або опис пристрою
user_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на користувача
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час реєстрації пристрою
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час останнього оновлення інформації

Сутність UserApplication (Зв'язок Користувача з Додатком): реалізує відношення багато-до-багатьох між користувачами та додатками (таблиця 2.8).

Таблиця 2.8

Структура таблиці user_application

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор зв'язку
user_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на користувача
application_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на додаток
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час створення зв'язку
		UNIQUE(user_id, application_id)	Унікальність пари користувач-додаток

Таблиця 2.9

Структура таблиці application_dependency

Поле	Тип даних	Обмеження	Опис
application_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на головний додаток
dependent_application_id	BIGINT	NOT NULL, FOREIGN KEY	Посилання на залежний додаток
		PRIMARY KEY(application_id, dependent_application_id)	Композитний первинний ключ

Ця таблиця забезпечує зберігання ієрархічних залежностей між додатками, дозволяючи реалізувати каскадне управління доступом (таблиця 2.9).

Сутність Setting (*Налаштування*): містить глобальні параметри конфігурації системи безпеки (таблиця 2.10)..

Таблиця 2.10

Структура таблиці setting_access_guard

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор налаштування
device_auth_enabled	BOOLEAN	DEFAULT FALSE	Глобальне увімкнення автентифікації за пристроєм
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час створення запису
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час останнього оновлення

Сутність UserVisitStatistics (Статистика Відвідувань): s є сутністю для аудиту доступу, що зберігає деталізовану інформацію про спроби автентифікації. Ця модель забезпечує комплексний моніторинг та аудит активності користувачів (таблиця 2.11).

Таблиця 2.11

Структура таблиці user_visit_statistics

Поле	Тип даних	Обмеження	Опис
id	BIGINT	PRIMARY KEY	Унікальний ідентифікатор запису
username	VARCHAR(255)	NOT NULL	Ім'я користувача, що ініціював запит
visit_date	BIGINT	NOT NULL	Часова мітка відвідування (Unix timestamp)
ip_address	VARCHAR(100)		IP-адреса джерела запиту
user_agent	VARCHAR(1000)		Інформація про клієнтське програмне забезпечення
request_url	VARCHAR(1000)		URL-адреса запиту

Продовження таблиці 2.11

Структура таблиці user_visit_statistics

Поле	Тип даних	Обмеження	Опис
http_method	VARCHAR(20)		HTTP метод запиту
authentication_method	VARCHAR(50)		Метод автентифікації
is_successful	BOOLEAN	DEFAULT FALSE	Індикатор успішності автентифікації
error_type	VARCHAR(255)		Тип помилки (якщо є)
error_message	TEXT		Деталізований опис помилки
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Час створення запису

Особливості реляційної моделі:

1. Застосування індексів для оптимізації продуктивності запитів:

1.1. Індекси на полях *username* та *mattermost_id* в таблиці *user_auth*;

1.2. Індекс на полі *spring_application_name* в таблиці *application*;

1.3. Індекс на полі *device_id* в таблиці *device*;

1.4. Композитний індекс на полях *user_id* та *application_id* в таблиці *user_application*;

2. Обмеження цілісності для забезпечення консистентності даних:

2.1. Унікальні обмеження на ключові поля (*username*, *spring_application_name*, *device_id*);

2.2. Зовнішні ключі для забезпечення референційної цілісності;

2.3. Композитний первинний ключ у таблиці *application_dependency*;

3. Часові мітки для відстеження життєвого циклу сутностей:

3.1. Поля *created_at* та *updated_at* для відстеження створення та

оновлення записів;

3.2. Поле *last_active_at* для моніторингу активності користувачів;

4. Оптимізація зберігання даних:

4.1. Використання *BYTEA* для зберігання бінарних даних зображень;

4.2. Застосування *TEXT* для довгих текстових полів;

4.3. Вибір відповідних типів даних для оптимального використання простору.

Особливістю розробленої доменної моделі є її орієнтація на підтримку множинних механізмів автентифікації та авторизації, що відображає ключові вимоги до системи безпеки в мікросервісному середовищі. Модель забезпечує гнучкість налаштування параметрів безпеки на рівні окремих користувачів, додатків та їх взаємозв'язків.

Реляційна схема бази даних представлена на діаграмі (рисунок 2.6):

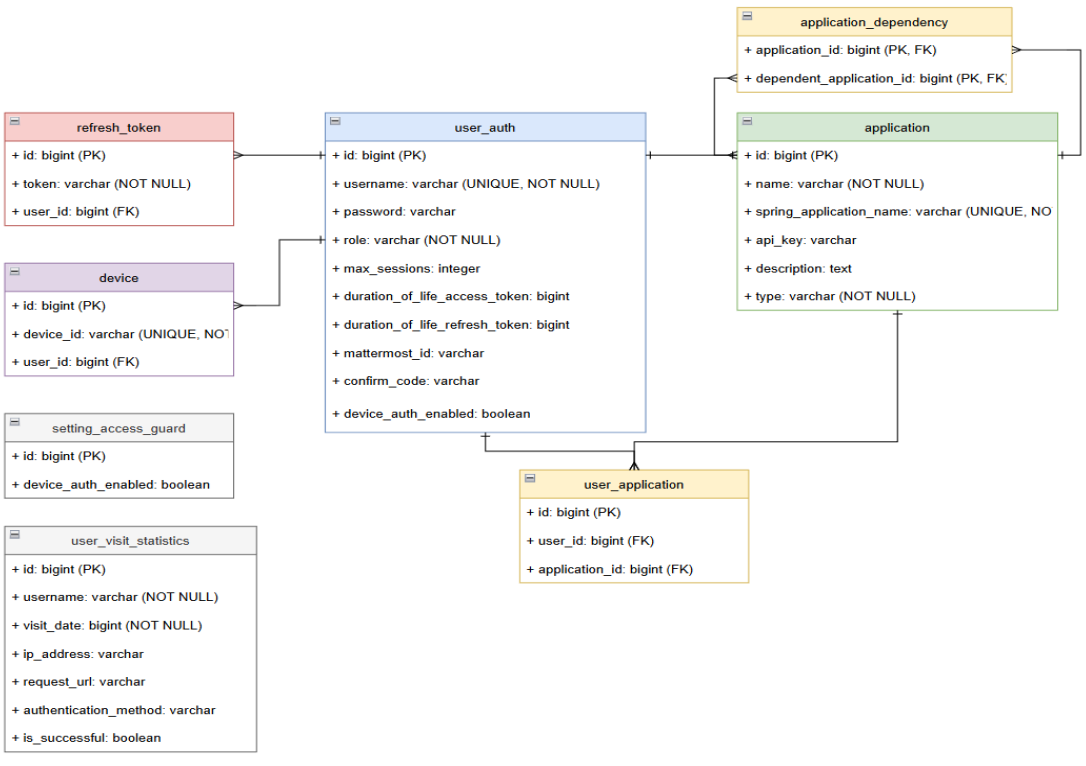


Рис. 2.6 Реляційна схема бази даних Access Guard

2.3.2 Проектування механізмів аудиту та історії змін

Забезпечення повного аудиту та збереження історії змін є імперативною вимогою для систем безпеки, особливо тих, що управляють доступом до корпоративних ресурсів. У контексті Access Guard ця проблематика особливо актуальна, оскільки будь-які несанкціоновані зміни в параметрах користувачів (або їхніх правах доступу) можуть призвести до серйозних безпекових інцидентів.

Розроблена система аудиту в Access Guard реалізує комплексний підхід до відстеження змін у критичних сутностях (рисунок 2.7) через поєднання наступних механізмів.

Для забезпечення автоматичного відстеження змін у ключових таблицях використовується бібліотека Hibernate Envers. Цей підхід забезпечує збереження повної історії змін критичних сутностей без необхідності додаткового програмування логіки відстеження змін.

Анотація @Audited активує автоматичне відстеження змін для всіх полів сутності. Для кожної аудійованої таблиці Hibernate Envers створює відповідну таблицю аудиту з суфіксом _AUD (таблиця 2.12):

access-guard > src > main > java > ua > com > tsa > accessguard > domain > model > user >

```
1   @Getter
2   @Setter
3   @Entity
4   @Audited
5   @NoArgsConstructor
6   @AllArgsConstructor
7   @Table(name = "user_auth")
8   @Accessors(chain = true)
9   @EntityListeners(AuditingEntityListener.class)
10  public class User  implements UserDetails {
11
12      @Id
13      @Column(name = "id")
14      private Long id;
15      // Entity fields ...
```

Рис. 2.7 Приклад налаштування аудиту за допомогою Hibernate Envers

Таблиця 2.12

Структура таблиці user_auth_AUD для аудиту

Поле	**Тип даних**	**Обмеження**	**Опис**
id	BIGINT	NOT NULL	Ідентифікатор користувача (від основної таблиці)
REV	INTEGER	NOT NULL	Ідентифікатор ревізії
REVTYPE	SMALLINT	NOT NULL	Тип ревізії (0 - створення, 1 - оновлення, 2 - видалення)
username	VARCHAR (255)		Ім'я користувача на момент ревізії
password	VARCHAR (255)		Пароль на момент ревізії
role	VARCHAR (50)		Роль на момент ревізії
max_sessions	INTEGER		Кількість сесій на момент ревізії
duration_of_life_access_token	BIGINT		Тривалість access токену на момент ревізії
duration_of_life_refresh_token	BIGINT		Тривалість refresh токену на момент ревізії
mattermost_id	VARCHAR (255)		Ідентифікатор Mattermost на момент ревізії
			Статус автентифікації пристрою на момент ревізії

Продовження таблиці 2.12

Структура таблиці user_auth_AUD для аудиту

device_auth_enabled	BOOLEAN		Статус автентифікації пристрою на момент ревізії
		PRIMARY KEY (id, REV)	Композитний первинний ключ
		FOREIGN KEY (REV) REFERENCES REVINFO(REV)	Зв'язок з таблицею ревізій

Таблиця 2.13

Структура таблиці REVINFO для зберігання ревізій

Поле	**Тип даних**	**Обмеження**	**Опис**
REV	INTEGER	GENERATED BY DEFAULT AS IDENTITY, PRIMARY KEY	Унікальний ідентифікатор ревізії
REVTSTMP	BIGINT	NOT NULL	Часова мітка ревізії (Unix timestamp)
username	VARCHAR(255)		Ім'я користувача, який здійснив зміну
ip_address	VARCHAR(100)		IP-адреса, з якої було здійснено зміну

Таблиця REVINFO зберігає інформацію про ревізії (версії) змін, включаючи часову мітку та, за наявності розширеної конфігурації, ідентифікатор користувача, який здійснив зміну (таблиця 2.13).

Для розширення стандартної функціональності Hibernate Envers, в системі впроваджено кастомний слухач ревізій, який збагачує записи аудиту контекстною інформацією (рисунок 2.8):

```java
@Component
public class AuditRevisionListener implements RevisionListener {

    @Autowired
    private SecurityContextHolder securityContextHolder;

    @Override
    public void newRevision(Object revisionEntity) {
        CustomRevisionEntity revision = (CustomRevisionEntity) revisionEntity;

        // Встановлення ідентифікатора користувача, який здійснив зміну
        if (securityContextHolder.getContext() != null &&
            securityContextHolder.getContext().getAuthentication() != null) {
            revision.setUsername(securityContextHolder.getContext().getAuthentication().getName());
        } else {
            revision.setUsername("system");
        }

        // Додавання IP-адреси та інших даних про контекст
        HttpServletRequest request =
            ((ServletRequestAttributes) RequestContextHolder.currentRequestAttributes()).getRequest();
        revision.setIpAddress(getClientIpAddress(request));
    }

    // Допоміжні методи
}
```

Рис. 2.8 Конфігурація користувацького слухача ревізій

Для ефективного використання даних аудиту система включає спеціалізовані репозиторії, що забезпечують доступ до історичних змін:

```java
public interface AuditRepository {

    <T> List<Revision<Integer, T>> findRevisions(Class<T> entityClass, Long entityId);

    <T> T findRevision(Class<T> entityClass, Long entityId, Integer revisionNumber);

    <T> Map<RevisionType, List<T>> compareRevisions(Class<T> entityClass,
                                                    Long entityId,
                                                    Integer oldRevision,
                                                    Integer newRevision);
}
```

Рис. 2.9 Інтерфейс репозиторію для доступу до історії змін

Цей інтерфейс надає методи для отримання всієї історії ревізій сутності, доступу до конкретної ревізії та порівняння різних ревізій для виявлення конкретних змін.

Окрім версіонування даних, система впроваджує механізми для відстеження активності користувачів, зокрема спроб автентифікації та авторизації. Ця інформація зберігається у таблиці user_visit_statistics та надає детальний контекст для аудиту безпеки.

Особливістю цього підходу є впровадження специфічних анотацій для

маркування методів, що підлягають логуванню, та використання анотацій для вказівки чутливих даних (рисунок 2.10), які повинні бути замасковані в логах:

```java
@Target({ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface HideLogSensitiveData {
    String[] visibleArgs() default {};
    boolean hideReturn() default true;
}

@HideLogSensitiveData(visibleArgs = {"applicationName"})
public boolean isApplicationInToken(String token, String applicationName) {
    // Реалізація
}
```

Рис. 2.10 Анотація та аспект для маскування чутливих даних

Розроблена система аудиту та історії змін забезпечує повну прозорість всіх модифікацій у критичних даних системи безпеки, що є фундаментальною вимогою для систем управління доступом у корпоративному середовищі.

2.3.3 Інтеграція з системою моніторингу та логування

Для забезпечення ефективного аналізу даних аудиту та операційного стану системи, Access Guard інтегрується з комплексом інструментів моніторингу та логування. Ця інтеграція забезпечує централізований збір, аналіз та візуалізацію як технічних метрик, так і бізнес-подій, пов'язаних із безпекою.

Access Guard експонує ключові метрики продуктивності та безпеки через ендпоінт */actuator/prometheus*, що дозволяє системі Prometheus збирати та зберігати часові ряди даних. Серед важливих метрик, що відстежуються:

- кількість успішних та невдалих спроб автентифікації;
- час відгуку на запити автентифікації;
- кількість активних сесій;
- частота оновлення токенів;
- кількість заблокованих користувачів.

Grafana використовується для створення інтерактивних дашбордів, що

візуалізують зібрані метрики та дозволяють швидко виявляти аномалії або проблеми безпеки. Спеціалізовані дашборди включають:

— "Authentication Monitoring" — відстеження спроб автентифікації;

— "Token Management" — аналіз життєвого циклу токенів;

— "User Activity" — моніторинг активності користувачів;

— "Security Alerts" — відображення потенційних загроз безпеці.

Система використовує Loki для централізованого збору та аналізу логів з усіх компонентів Access Guard. Логи структуруються за допомогою MDC (Mapped Diagnostic Context), що збагачує кожен запис логу контекстною інформацією:

— ідентифікатор запиту для відстеження повного шляху обробки;

— ім'я користувача, який ініціював операцію;

— IP-адреса та User-Agent клієнта;

— результат операції (успіх/невдача);

— деталі помилки (якщо є).

Особливістю системи логування є автоматичне маскування чутливих даних (паролів, токенів) через використання кастомних анотацій та аспектів:

```
@Aspect
@Component
@RequiredArgsConstructor
public class LoggingAspect {

    @Around("(@within(ua.com.tsa.accessguard.aop.annotation.LogController) ||
    public Object logMethod(ProceedingJoinPoint joinPoint) throws Throwable {
        // Логіка маскування чутливих даних у логах
        // ...
    }
}
```

Рис. 2.11 Аспект логування

Для відстеження повного шляху обробки запитів через різні компоненти мікросервісної архітектури використовується Zipkin. Кожен запит отримує

унікальний ідентифікатор трасування (traceId) та ідентифікатори окремих сегментів (spanId), що дозволяє візуалізувати повний шлях обробки запиту від клієнта через Gateway, Access Guard та інші сервіси.

Tracing налаштований таким чином, щоб автоматично поширювати контекст трасування через усі сервіси, забезпечуючи повну видимість обробки запитів автентифікації та авторизації. Система збирає детальну інформацію про час виконання кожного етапу обробки запиту, включаючи час очікування між сервісами, тривалість роботи з базою даних та взаємодію з зовнішніми API.

Zipkin інтегрований з усіма компонентами системи через Spring Cloud Sleuth, що забезпечує автоматичне додавання заголовків трасування до HTTP-запитів та повідомлень RabbitMQ. Це дозволяє відстежувати не тільки синхронні виклики між сервісами, але й асинхронні операції, такі як обробка повідомлень про зміни користувачів або оновлення даних додатків.

Особливо корисною є можливість аналізу проблемних запитів автентифікації, коли можна точно визначити, на якому етапі відбувається затримка (таблиця 3.4): при перевірці токена в Access Guard, отриманні даних користувача з User Hub, або взаємодії з Mattermost API. Zipkin також надає статистику про найповільніші операції та допомагає виявляти вузькі місця в архітектурі системи, що є критично важливим для оптимізації продуктивності сервісу автентифікації.

Діаграма потоку даних аудиту представлена нижче (рисунку 2.12):

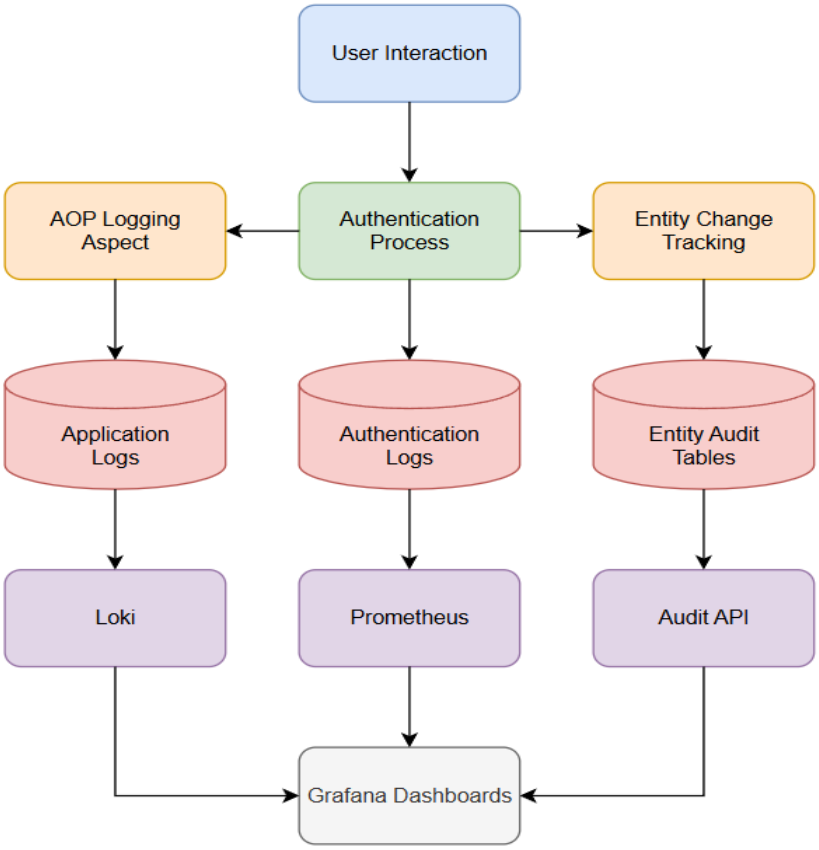


Рис. 2.12 Діаграма потоку даних аудиту

Розроблена система аудиту та історії змін забезпечує повну прозорість всіх модифікацій у критичних даних системи безпеки, що є фундаментальною вимогою для систем управління доступом у корпоративному середовищі.

2.4 Проектування системи автентифікації та авторизації

2.4.1 Архітектура JWT-автентифікації з механізмом оновлення токенів

Архітектура системи автентифікації є фундаментальним компонентом Access Guard, що забезпечує надійну верифікацію ідентичності користувачів. Центральним елементом розробленої архітектури є механізм автентифікації на основі JSON Web Token (JWT) з підтримкою оновлення токенів, що забезпечує баланс між безпекою та зручністю користування.

JWT-автентифікація базується на створенні підписаних токенів, що містять інформацію про користувача та його повноваження. Цей підхід надає значні переваги для мікросервісної архітектури (таблиця 2.14).

Таблиця 2.14

Переваги для мікросервісної архітектури

Перевага	**Опис**
Безсесійність	Сервер не зберігає стан сесії, що суттєво спрощує горизонтальне масштабування системи та зменшує навантаження на сервер
Самодостатність токенів	Уся необхідна інформація для авторизації міститься безпосередньо в токені, що зменшує потребу в додаткових запитах до бази даних
Підтримка мікросервісної архітектури	Токени можуть використовуватися різними сервісами без додаткових запитів до сервера автентифікації, що сприяє автономності мікросервісів

Токени в системі Access Guard мають чітко визначену структуру з трьох компонентів, які показані в таблиці 2.15.

Таблиця 2.15

Компоненти JWT токена

Компонент	**Зміст**
Header	Містить метадані про тип токена та використаний алгоритм підпису (HS256)
Payload	Містить корисні дані: - *sub*: Ім'я користувача; - *userId*: Ідентифікатор користувача; - *role*: Роль користувача в системі; - *tokenType*: Тип токену (ACCESS або REFRESH); - *applications*: Список доступних користувачу додатків; - *exp*: Час закінчення дії токена.
Signature	Цифровий підпис, що забезпечує цілісність та автентичність токена

Архітектура автентифікації реалізує дворівневу систему токенів з чітким розподілом обов'язків (таблиця 2.16).

Таблиця 2.16

Типи токенів та їх характеристики

Тип токена	**Характеристики**	**Призначення**
Access Token	- Обмежений термін дії (за замовчуванням 15 хвилин); - містить інформацію про доступні застосунки; - легкий та компактний.	Використовується для авторизації запитів до захищених ресурсів
Refresh Token	- Триваліший термін дії (за замовчуванням 7 днів); - зберігається в базі даних для можливості відкликання; - унікальний ідентифікатор.	Використовується виключно для отримання нового access token після закінчення терміну його дії

Така дворівнева архітектура забезпечує оптимальний компроміс між безпекою (короткий термін дії access token) та зручністю користування (довготривалий refresh token для безперебійної роботи). Процес автентифікації користувача складається з трьох ключових етапів (рисунок 2.13).

Початкова автентифікація:

1. Користувач надає облікові дані (логін/пароль або інший метод автентифікації);
2. Система перевіряє коректність даних;
3. Генеруються access і refresh токени;
4. Токени передаються клієнту (через HTTP-only cookies або у відповіді API).

Використання access token:

1. Клієнт додає access token до кожного запиту до захищених ресурсів;

2. Сервер перевіряє підпис та термін дії токена;
3. При успішній валідації запит обробляється.

Оновлення токенів:

1. Коли термін дії access token закінчується, клієнт використовує refresh token для отримання нового;
2. Сервер перевіряє валідність refresh token у базі даних;
3. Генерується новий access token (та опціонально новий refresh token).

Ключовим елементом архітектури є механізм оновлення токенів (рисунок 2.14), що забезпечує безперервність користувацького досвіду без необхідності повторної автентифікації:

```
@Transactional
@HideLogSensitiveData
public TokenResponse refreshToken(@NonNull TokenResponse token) {
    Token refreshToken = tokenService.getByToken(token.getRefreshToken());
    User user = refreshToken.getUser();

    if (tokenService.isTokenExpired(refreshToken.getToken())) {
        tokenService.delete(refreshToken);
        throw new RefreshTokenExpiredException();
    }

    String newAccessToken = tokenService.createAccessToken(user);
    return new TokenResponse(newAccessToken, refreshToken.getToken());
}
```

Рис. 2.13 Процес генерації та валідації JWT токенів

Процес оновлення токенів охоплює наступні етапи:

1. Клієнт надсилає refresh token для оновлення;
2. Сервер перевіряє наявність та дійсність refresh token у базі даних;
3. Валідується термін дії refresh token;
4. Генерується новий access token;
5. Опціонально, може оновлюватися і сам refresh token.

В архітектурі Gateway використовується розширений механізм автоматичного оновлення токенів:

```
@Override
public Mono<SecurityContext> load(ServerWebExchange exchange) {
    // Спроба автентифікації через cookies
    return Mono.justOrEmpty(exchange.getRequest().getCookies().getFirst("accessToken"))
        .flatMap(accessTokenCookie -> {
            return authenticationManager.authenticateWithToken(accessTokenCookie.getValue())
                .map(authentication -> new SecurityContextImpl(authentication))
                .cast(SecurityContext.class);
        })
        .switchIfEmpty(
            // Інші методи автентифікації
        )
        .switchIfEmpty(Mono.defer(() -> {
            // Перевірка наявності refresh token, якщо немає access token
            HttpCookie refreshTokenCookie =
                exchange.getRequest().getCookies().getFirst("refreshToken");
            if (refreshTokenCookie != null) {
                return authenticationManager.authenticateByRefreshToken(refreshTokenCookie)
                    .flatMap(authentication -> {
                        return cookieService.setAuthenticationCookies(exchange, authentication)
                            .thenReturn(new SecurityContextImpl(authentication));
                    })
                    .cast(SecurityContext.class);
            }
            return Mono.empty();
        }))
        .onErrorResume(e -> {
            // Спроба відновлення через refresh token при помилці
            HttpCookie refreshTokenCookie =
                exchange.getRequest().getCookies().getFirst("refreshToken");
            if (refreshTokenCookie != null) {
                return authenticationManager.authenticateByRefreshToken(refreshTokenCookie)
                    .flatMap(authentication -> {
                        return cookieService.setAuthenticationCookies(exchange, authentication)
                            .thenReturn(new SecurityContextImpl(authentication));
                    })
                    .cast(SecurityContext.class);
            }
            return Mono.error(e);
        });
}
```

Рис. 2.14 Механізм оновлення токенів

Такий підхід забезпечує безперебійну роботу системи без необхідності для клієнта явно обробляти процес оновлення токенів.

Особлива увага приділяється безпечній передачі токенів між сервером та клієнтом. Система використовує HTTP-only cookies для зберігання токенів:

```java
private ResponseCookie createCookie(String name, String value, long duration) {
    return ResponseCookie.from(name, value)
        .path("/")
        .httpOnly(true)
        .secure(true)
        .sameSite("none")
        .maxAge(Duration.ofMinutes(duration))
        .build();
}

public void setAuthCookies(ServerWebExchange exchange, TokenResponse tokenResponse) {
    ResponseCookie accessTokenCookie = createCookie(
        "accessToken",
        tokenResponse.getAccessToken(),
        defaultAccessTokenDuration
    );
    ResponseCookie refreshTokenCookie = createCookie(
        "refreshToken",
        tokenResponse.getRefreshToken(),
        defaultRefreshTokenDuration
    );

    exchange.getResponse().addCookie(accessTokenCookie);
    exchange.getResponse().addCookie(refreshTokenCookie);
}
```

Рис. 2.15 Конфігурація безпечних cookies

Використання HTTP-only cookies забезпечує додатковий рівень захисту від XSS-атак, оскільки JavaScript не має доступу до таких cookies (рисунок 2.15).

Розроблена архітектура JWT-автентифікації з механізмом оновлення токенів забезпечує оптимальний баланс між безпекою, продуктивністю та зручністю використання системи, що є критично важливим для центрального сервісу автентифікації в мікросервісній архітектурі (рисунок 2.16).

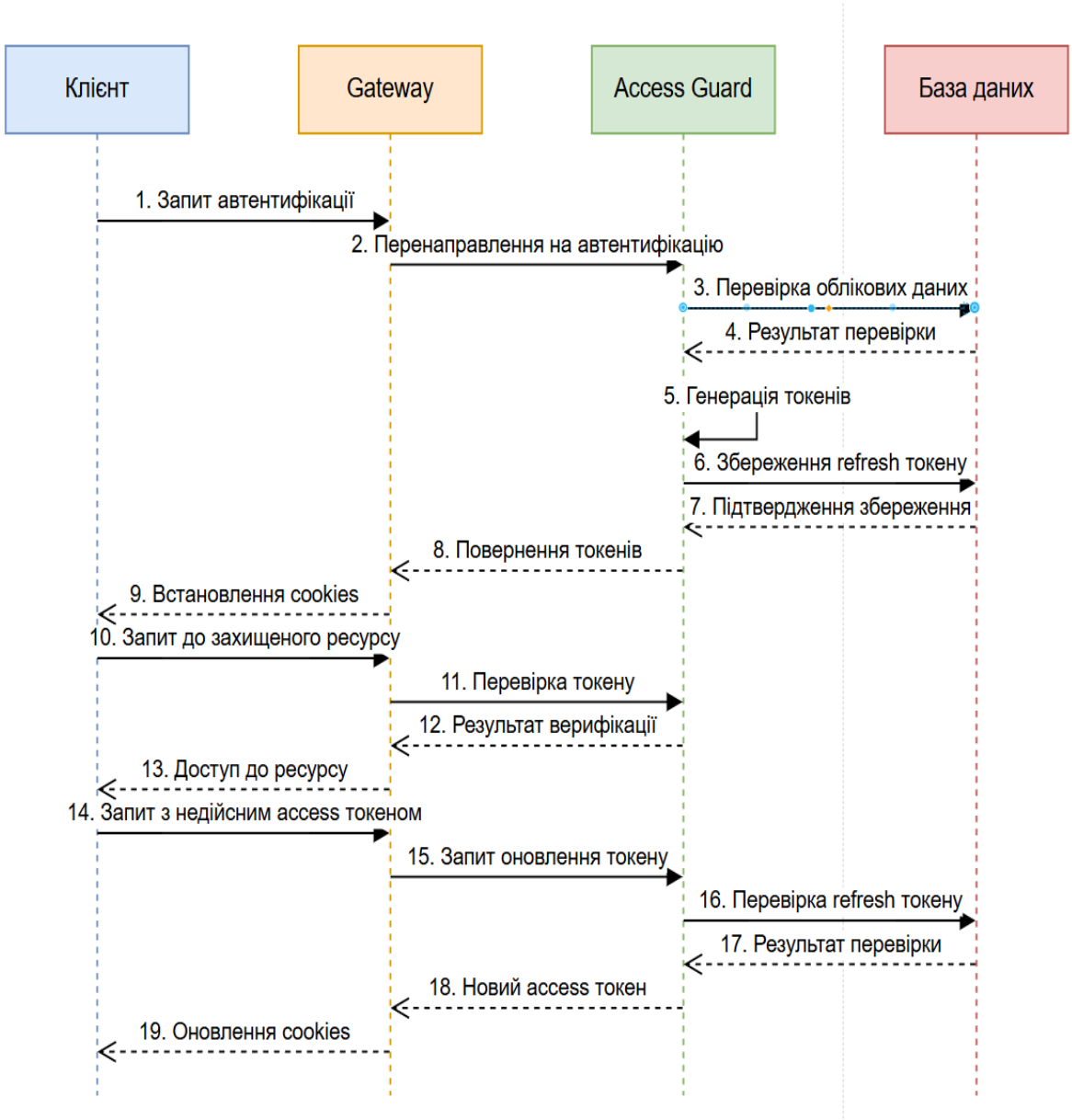


Рис. 2.16 Діаграма послідовності автентифікації

2.4.2 Проектування системи управління ролями користувачів

Система управління ролями користувачів є фундаментальним компонентом архітектури безпеки Access Guard, що забезпечує гнучке та ієрархічне розмежування доступу до ресурсів. Реалізована модель ґрунтується на принципах найменших привілеїв та чіткого розподілу відповідальності.

Архітектура системи передбачає ієрархічну модель ролей з чітко визначеними рівнями доступу (таблиця 2.17).

Таблиця 2.17

Ієрархія ролей користувачів

Роль	Призначення	Обсяг повноважень
ROLE_ROOT	Найвищий рівень доступу	Повний доступ до всіх функцій системи, включно з управлінням ролями, створенням адміністраторів та конфігурацією системних налаштувань
ROLE_ADMIN	Адміністративний доступ	Доступ до адміністративних функцій, включно з управлінням користувачами, додатками та налаштуваннями; обмежений доступ до критичних системних налаштувань
ROLE_SERVICE	Міжсервісна взаємодія	Спеціалізований доступ до API для інтеграції, обмежені адміністративні функції
ROLE_USER	Базовий рівень користувача	Доступ лише до дозволених додатків та власного профілю
ROLE_BLOCKED	Заблокований користувач	Повне обмеження доступу до всіх ресурсів системи

Ключовою особливістю ієрархічної моделі є наслідування прав: кожна вища роль автоматично має всі права нижчих ролей у ієрархії, що забезпечує логічну послідовність процесів авторизації.

Перевірка прав доступу на основі ролей реалізується на різних рівнях архітектури, що забезпечує багаторівневий захист системи (рисунок 2.17):

```
@RestController
@RequestMapping("/api/v1/settings")
public class SettingController {
    @PutMapping
    @ResponseStatus(HttpStatus.OK)
    @PreAuthorize("hasRole('ROLE_ROOT')")
    public Mono<ResponseDTO<Setting>> update(@RequestBody Setting setting) {
        return Mono.fromCallable(() -> new ResponseDTO<>(settingService.updateSetting(setting)));
    }
}
```

Рис. 2.17 Перевірка ролей на рівні контролерів

```java
@Transactional
public void deleteUser(Long userId) {
    User currentUser = securityUtil.getCurrentUser();
    User userToDelete = getById(userId);

    if (currentUser.getRole() != Role.ROLE_ROOT &&
        userToDelete.getRole() == Role.ROLE_ROOT) {
        throw new AccessDeniedException("Only ROOT users can delete ROOT users");
    }

    if (currentUser.getRole().ordinal() > userToDelete.getRole().ordinal()) {
        throw new AccessDeniedException("Cannot delete user with higher role");
    }

    userRepository.delete(userToDelete);
}
```

Рис. 2.18 Програмна перевірка ролей на рівні сервісів

Окрім системних ролей, архітектура передбачає можливість налаштування доступу на рівні окремих додатків через відповідну доменну модель. Система підтримує деталізовані ролі для кожного додатку (таблиця 2.18):

Таблиця 2.18

Ролі додатка та їх призначення

Роль додатка	**Призначення**
APP_ROLE_ADMIN	Адміністратор додатку з повними правами управління
APP_ROLE_EDITOR	Редактор з можливістю модифікації даних
APP_ROLE_VIEWER	Оглядач з правами лише на перегляд

Цей підхід дозволяє незалежно налаштовувати рівень доступу користувача до кожного конкретного додатку, незалежно від його системної ролі. Наприклад, користувач із роллю ROLE_USER може мати роль APP_ROLE_ADMIN в одному додатку та APP_ROLE_VIEWER в іншому.

Додатково система підтримує тонке налаштування доступу до окремих таблиць бази даних, що дозволяє обмежити видимість певних даних навіть у межах одного додатку.

Архітектура системи передбачає спеціалізований адміністративний

інтерфейс для управління ролями користувачів через API (рисунок 2.19):

```
@PutMapping("/{id}/role")
@ResponseStatus(HttpStatus.OK)
@PreAuthorize("hasRole('ROLE_ROOT')")
public Mono<ResponseDTO<User>> updateRole(@PathVariable Long id, @RequestBody RoleUpdateRequest request) {
    return Mono.fromCallable(() -> {
        User user = userService.getById(id);
        user.setRole(request.getRole());
        return new ResponseDTO<>(userService.update(id, user));
    });
}
```

Рис. 2.19 API для управління ролями користувачів

Система підтримує ряд обмежень для запобігання несанкціонованого підвищення привілеїв:

1. Лише користувачі з роллю ROLE_ROOT можуть призначати роль ROLE_ROOT;
2. Користувач не може підвищити роль іншого користувача вище своєї власної;
3. Користувачі з вищим рівнем ролі не можуть бути видалені користувачами з нижчим рівнем ролі.

Діаграма процесу авторизації на основі ролей (рисунок 2.20):

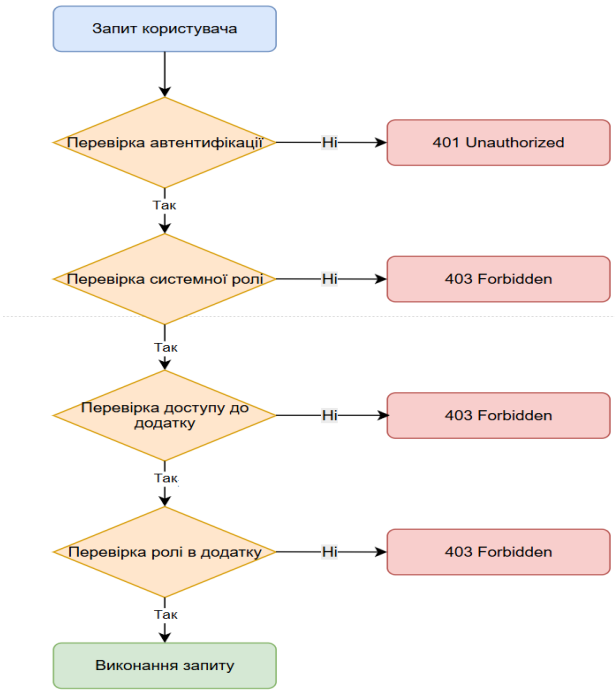


Рис. 2.20 Діаграма процесу авторизації на основі ролей

2.4.3 Архітектура двофакторної автентифікації через Mattermost

У контексті корпоративної системи автентифікації, додатковий рівень безпеки стає критично важливим для запобігання несанкціонованого доступу до конфіденційної інформації. Розроблена архітектура двофакторної автентифікації (2FA) через корпоративний месенджер Mattermost є інноваційним підходом, що дозволяє підвищити безпеку без зниження зручності користування системою.

Архітектура 2FA через Mattermost базується на принципі "щось, що ви знаєте" (логін/пароль) та "щось, що ви отримуєте" (код підтвердження у корпоративному месенджері). Такий підхід має ряд переваг у корпоративному середовищі:

1. Інтеграція з існуючою інфраструктурою — використання наявних каналів комунікації;
2. Мінімальний вплив на користувацький досвід — звична для корпоративних користувачів взаємодія;
3. Відсутність залежності від особистих пристроїв — немає необхідності в смартфоні чи додаткових додатках;
4. Відслідковуваність — всі повідомлення з кодами зберігаються в корпоративній системі.

Процес двофакторної автентифікації через Mattermost містить наступні етапи (рисунок 2.21):

1. Введення облікових даних користувача:
    a. Користувач вводить своє ім'я користувача в системі;
    b. Система ідентифікує відповідний Mattermost ID користувача;
2. Генерація та відправка коду підтвердження:
    a. Система генерує випадковий числовий код (зазвичай 4-6 цифр);
    b. Код зберігається у профілі користувача в зашифрованому вигляді;
    c. Код надсилається користувачу через Mattermost API;
3. Перевірка коду:
    a. Користувач отримує код у Mattermost та вводить його в систему;

b. Система порівнює введений код із збереженим;

c. При успішній перевірці система генерує токени доступу;

d. Код підтвердження видаляється з профілю користувача.

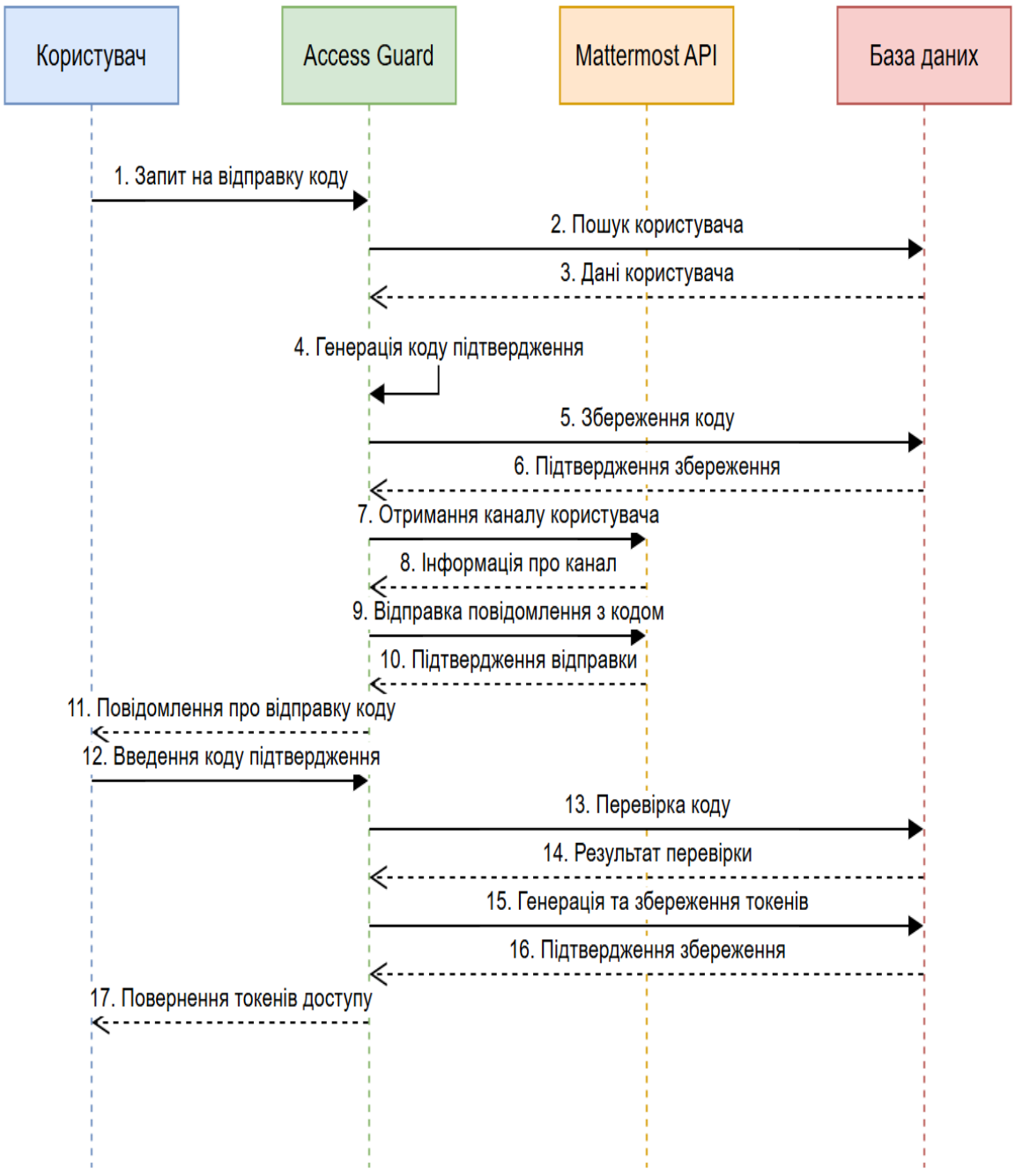


Рис. 2.21 Діаграма послідовності двофакторної автентифікації

Архітектурну реалізацію двофакторної автентифікації забезпечує спеціалізований сервісний компонент, що відповідає за інтеграцію з Mattermost (рисунок 2.22). Даний компонент використовує реактивний підхід до взаємодії з

API Mattermost, що забезпечує асинхронність і високу продуктивність обробки запитів. Безпечна передача даних забезпечується використанням механізму Bearer-токенів для авторизації з API Mattermost.

```java
@Service
@RequiredArgsConstructor
public class MattermostIntegrationService {

    @Value("${url.mattermost.base}")
    private String mattermostUrl;

    @Value("${path.mattermost.channel}")
    private String pathChannel;

    @Value("${path.mattermost.create-post}")
    private String pathCreatePost;

    @Value("${api-key.mattermost}")
    private String apiKey;

    @Value("${bot.mattermost.auth}")
    private String botAuth;

    private final WebClientService webClientService;

    public void sendConfirmationCode(String mattermostId, String confirmCode) {
        Map<String, String> headers = new HashMap<>() {{
            put(HttpHeaders.AUTHORIZATION, "Bearer " + apiKey);
        }};

        // Отримання ID каналу
        webClientService.post(
            mattermostUrl,
            pathChannel,
            Arrays.asList(botAuth, mattermostId),
            new ParameterizedTypeReference<Map<String, String>>() {},
            headers
        )
        .subscribe(response -> {
            String channelId = response.get("id");

            // Відправка коду підтвердження в канал
            sendMessageToChannel(channelId, confirmCode);
        });
    }

    private void sendMessageToChannel(String channelId, String confirmCode) {
        Map<String, Object> requestBody = new HashMap<>() {{
            put("channel_id", channelId);
            put("message", formatSecurityMessage(confirmCode));
        }};

        Map<String, String> headers = new HashMap<>() {{
            put(HttpHeaders.AUTHORIZATION, "Bearer " + apiKey);
        }};

        webClientService.post(
            mattermostUrl,
            pathCreatePost,
            requestBody,
            new ParameterizedTypeReference<Map<String, Object>>() {},
            headers
        )
        .subscribe(
            response -> log.debug("Message with confirmation code sent successfully"),
            error -> log.error("Failed to send message with confirmation code", error)
        );
    }
```

Рис. 2.22 Компонент інтеграції з Mattermost

Для генерації кодів підтвердження (рисунок 2.23) застосовано надійний алгоритм з такими характеристиками:

1. Випадковість — використання криптографічно стійкого генератора псевдовипадкових чисел;
2. Простота використання — коди складаються лише з цифр для зручності введення;
3. Достатня ентропія — 10^6 можливих комбінацій, що є достатнім для тимчасового коду.

```
public class PasswordService {

    private static final int CONFIRMATION_CODE_LENGTH = 6;
    private static final String DIGITS = "0123456789";

    public static String generateRandomConfirmationPassword() {
        SecureRandom random = new SecureRandom();
        StringBuilder sb = new StringBuilder(CONFIRMATION_CODE_LENGTH);

        for (int i = 0; i < CONFIRMATION_CODE_LENGTH; i++) {
            int randomIndex = random.nextInt(DIGITS.length());
            sb.append(DIGITS.charAt(randomIndex));
        }

        return sb.toString();
    }
}
```

Рис. 2.23 Алгоритм генерації кодів підтвердження

Архітектура передбачає двоетапний процес автентифікації. Перший етап відповідає за генерацію та надсилання коду підтвердження. Під час цього етапу система перевіряє наявність зв'язку користувача з Mattermost, генерує унікальний код підтвердження, зберігає його в профілі користувача та надсилає через API Mattermost.

Другий етап — це верифікація введеного користувачем коду. Система порівнює введений код із збереженим, і в разі успішної перевірки генерує токени доступу. Важливим аспектом є негайне видалення коду підтвердження з профілю користувача після успішної автентифікації, що запобігає повторному використанню коду (рисунок 2.24).

```java
@Transactional
public void generateAndSendCode(String username) {
    User byId = userService.getByUsername(username);

    if (byId.getMattermostId() == null || byId.getMattermostId().isEmpty()) {
        throw new UserDontHaveMattermostException();
    }

    String confirmCode = PasswordService.generateRandomConfirmationPassword();
    byId.setConfirmCode(confirmCode);
    userService.update(byId.getId(), byId);

    // Передача даних для відправки в Mattermost
    integrationService.sendConfirmationCode(byId.getMattermostId(), confirmCode);
}

@Transactional
@HideLogSensitiveData
public TokenResponse confirmCode(@NonNull SignInRequest signInRequest) {
    User byId = userService.getByUsername(signInRequest.getUsername());

    if (byId.getMattermostId() == null || byId.getMattermostId().isEmpty()) {
        throw new UserDontHaveMattermostException();
    }

    if (byId.getConfirmCode() == null ||
        !byId.getConfirmCode().equals(signInRequest.getCode())) {
        throw new InvalidOrEmptyVerificationDataException("confirmation code");
    }

    // Очищення коду підтвердження
    byId.setConfirmCode(null);
    userService.update(byId.getId(), byId);

    // Керування сесіями
    List<Token> userTokens = tokenService.findAllValidTokenByUser(byId.getId());
    if (byId.getMaxSessions() == null) {
        byId.setMaxSessions(defaultMaxSessions);
    }
    if (userTokens.size() >= byId.getMaxSessions()) {
        tokenService.delete(userTokens.get(0)); // Видаляємо найстарішу сесію
    }

    // Генерація токенів
    String accessToken = tokenService.createAccessToken(byId);
    String refreshToken = tokenService.createRefreshToken(byId).getToken();

    return new TokenResponse(accessToken, refreshToken);
}
```

Рис. 2.24 Процес верифікації коду автентифікації

Для забезпечення зручності використання системи двофакторної автентифікації розроблено інтуїтивний користувацький інтерфейс, що проводить користувача через всі етапи автентифікації. Інтерфейс дозволяє обрати метод автентифікації (звичайний, через Mattermost або через пристрій), ввести ім'я користувача, запросити код підтвердження та ввести отриманий код.

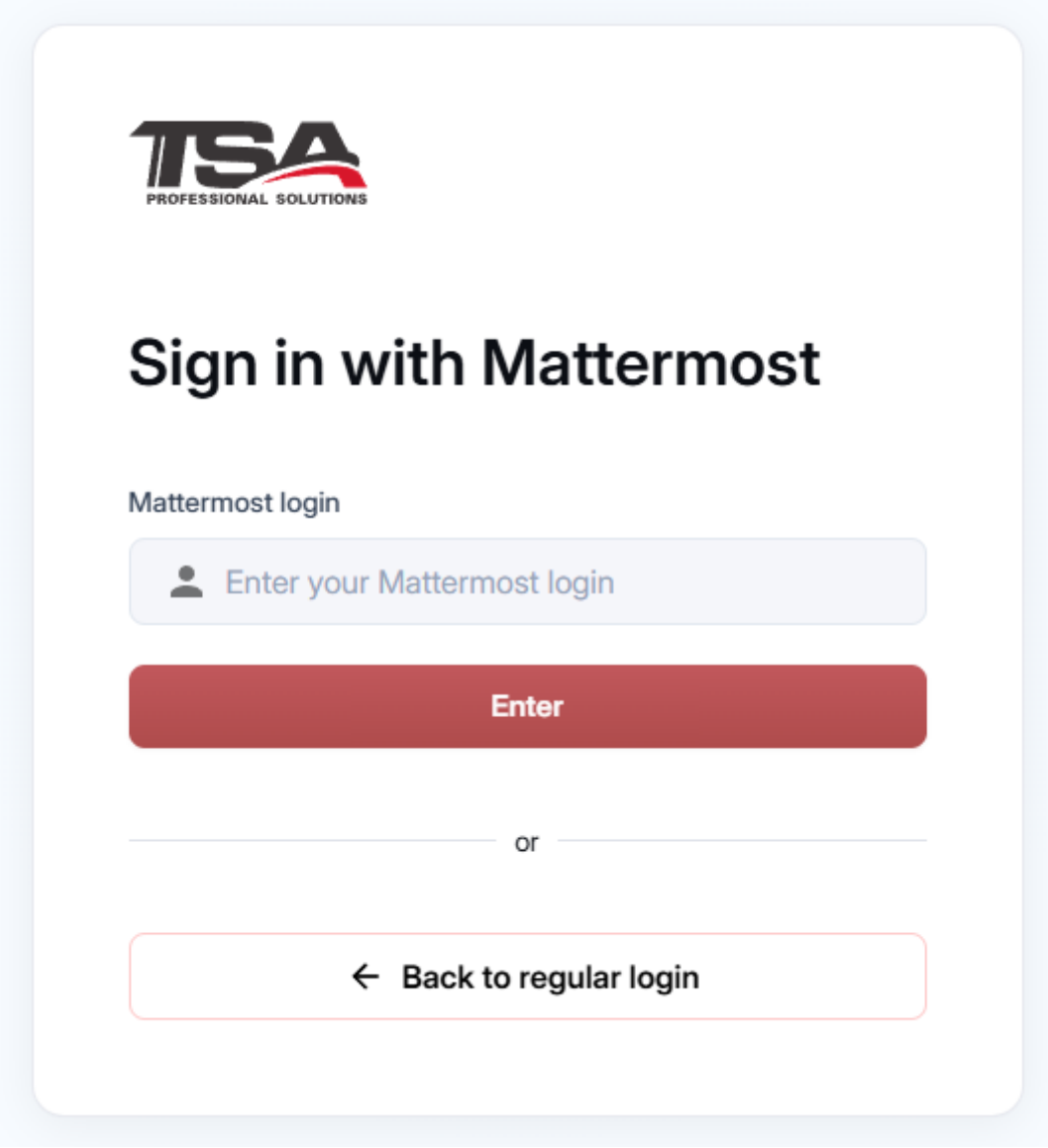


Рис. 2.25 Інтерфейс двофакторної автентифікації

Система містить низку захисних механізмів для забезпечення безпеки процесу автентифікації:

1. Механізм обмеження кількості спроб — після визначеної кількості невдалих спроб введення коду система тимчасово блокує можливість

автентифікації для захисту від брутфорс-атак;

2. Часове обмеження дії коду — кожен код має обмежений термін дії (зазвичай 5 хвилин), після чого автоматично анулюється. Для реалізації цього механізму використовується планувальник завдань, що періодично перевіряє та видаляє прострочені коди;

3. Унікальність кодів — кожен новий запит генерує новий код, анулюючи попередній, що унеможливлює використання старих кодів;

4. Автоматичне очищення коду після використання — після успішної автентифікації код видаляється з профілю користувача, запобігаючи повторному використанню.

Важливою особливістю архітектури є її розширюваність та адаптивність. Систему двофакторної автентифікації через Mattermost можна розширити для підтримки автентифікації мобільних та інших пристроїв. При такому сценарії користувач може автентифікуватися через свій зареєстрований пристрій, а система надсилатиме код підтвердження через Mattermost, забезпечуючи таким чином додатковий рівень безпеки.

Архітектура передбачає гнучкі можливості налаштування системи двофакторної автентифікації як на рівні всієї системи, так і на рівні окремих користувачів. Адміністратори можуть вмикати або вимикати двофакторну автентифікацію через Mattermost глобально.

2.5 Проектування адміністративного інтерфейсу

2.5.1 Архітектура frontend-компонентів

Адміністративний інтерфейс Access Guard представляє собою комплексну React-аплікацію, розроблену за принципами компонентно-орієнтованої архітектури. Використання TypeScript як основної мови програмування забезпечує статичну типізацію та підвищує надійність коду.

Архітектура інтерфейсу ґрунтується на чотирьох основних логічних

шарах, кожен з яких має чітко визначену відповідальність:

- *презентаційний шар* відповідає за відображення даних та взаємодію з користувачем. Він включає компоненти різного рівня абстракції – від базових елементів інтерфейсу до комплексних складених компонентів, таких як таблиці користувачів чи форми автентифікації. На скриншотах видно реалізацію інтуїтивно зрозумілих таблиць для управління користувачами, налаштування відносин між користувачами та додатками, а також відстеження активності:
- *шар управління станом* забезпечує контроль над даними застосунку. Архітектура використовує комбінований підхід: Redux для глобального стану (автентифікація, навігація), React Query для керування станом серверних даних з автоматичним кешуванням, та локальні React-хуки для стану окремих компонентів;
- *комунікаційний шар* відповідає за взаємодію з бекендом через спеціалізовані API-клієнти. Цей шар абстрагує деталі HTTP-комунікації, обробляє помилки та забезпечує форматування запитів і відповідей. Вся комунікація з сервером відбувається через два основних клієнти: *guardClient* для взаємодії з Access Guard та *userHubClient* для взаємодії з User Hub;
- *утилітарний шар* надає допоміжні функції та сервіси для валідації даних, форматування, локалізації та інших крос-функціональних завдань.

Кодова база клієнтської частини вибудована за функціональним принципом із чітким розподілом компонентів (таблиця 2.19). В основі структури – каталоги для API-клієнтів та інтерфейсів взаємодії з сервером, багаторазових UI-компонентів, організованих за доменними областями, спеціалізованих React-хуків, файлів локалізації з підтримкою української та англійської мов, а також компонентів верхнього рівня, що формують основні сторінки застосунку. Така архітектурна організація забезпечує ефективну навігацію, полегшує розширення функціональності та підтримку проєкту в довгостроковій перспективі.

Таблиця 2.19

Модулі адміністративного інтерфейсу

Модуль	**Призначення**	**Функціональність**
Управління користувачами	Керування обліковими записами користувачів	Створення, редагування, видалення користувачів, призначення ролей та блокування/розблокування
Управління додатками	Керування додатками в системі	Реєстрація додатків, генерація API-ключів, налаштування залежностей
Налаштування відносин	Керування доступом користувачів до додатків	Визначення прав доступу, встановлення ролей в додатках
Моніторинг активності	Відстеження дій користувачів	Перегляд статистики входів, фільтрація за часовими періодами
Системні налаштування	Конфігурація параметрів безпеки	Увімкнення/вимкнення механізмів автентифікації

Особливою перевагою розробленого інтерфейсу є адаптивність та підтримка темної теми через використання Material UI компонентів. Інтернаціоналізація реалізована з використанням бібліотеки i18next, що забезпечує зручне перемикання між українською та англійською мовами інтерфейсу.

2.5.2 Проектування взаємодії між клієнтською та серверною частинами

Ефективна взаємодія між клієнтською та серверною частинами є критично важливим аспектом системи автентифікації та авторизації. В Access Guard ця взаємодія побудована на принципах RESTful архітектури, з використанням уніфікованого формату даних та стандартизацією відповідей.

Основними принципами проектування взаємодії є:

1. Використання стандартних HTTP-методів для CRUD-операцій;
2. Уніфікований формат передачі даних у форматі JSON;

3. Систематичний підхід до обробки помилок;
4. Стандартизована структура відповідей сервера;
5. Токен-базована автентифікація через JWT.

Для забезпечення консистентності всіх відповідей API, розроблено стандартизований формат, в якому кожна відповідь сервера містить об'єкт з полями *data* (корисні дані) та опціональним *error* (інформація про помилку). Завдяки цьому підходу клієнтський код може обробляти відповіді уніфіковано, незалежно від конкретного ендпоінту.

Процес автентифікації побудований на використанні JWT-токенів, що передаються в HTTP-only cookies, що додатково захищає їх від XSS-атак. Механізм автоматичного оновлення токенів реалізований у Gateway та не потребує додаткової логіки на стороні клієнта, що значно спрощує розробку та підвищує безпеку.

Для оптимізації продуктивності та покращення користувацького досвіду впроваджено кілька ключових технік:

1. *Кешування даних* – React Query забезпечує "розумне" кешування з налаштовуваним періодом актуальності, що мінімізує кількість запитів до сервера;
2. *Оптимістичні оновлення* – інтерфейс відображає зміни миттєво, не чекаючи відповіді сервера, з автоматичним відкатом при помилках;
3. *Пагінація та фільтрація* на стороні сервера, що дозволяє ефективно працювати з великими об'ємами даних.

Механізм обробки помилок реалізований на кількох рівнях, щоб забезпечити гнучкість та інформативність:

1. Глобальний обробник для стандартних сценаріїв (неавторизований доступ, заборонений ресурс);
2. Компоненти для відображення стану завантаження та помилок;
3. Контекстно-залежна обробка специфічних помилок для різних

операцій.

Безпека комунікацій забезпечується через:

— захист від CSRF через використання HTTP-only cookies;

— валідацію вхідних даних на клієнті перед відправкою на сервер;

— впровадження Content Security Policy для захисту від XSS-атак;

— використання Secure та HttpOnly атрибутів для cookies.

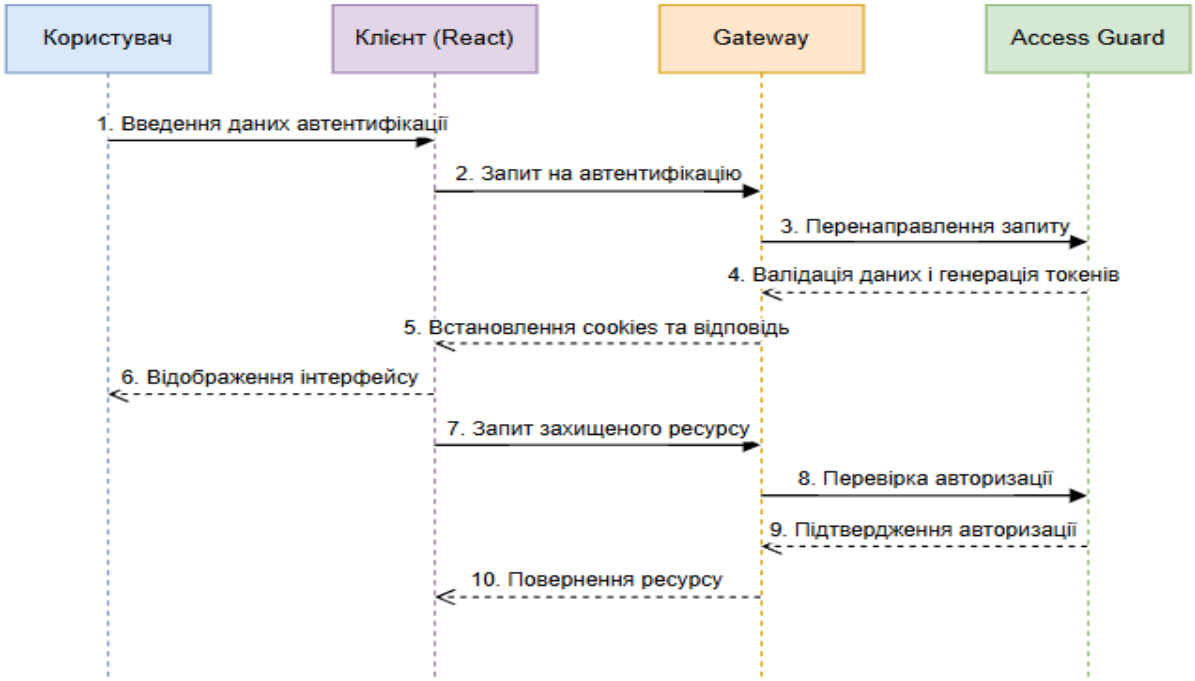


Рис. 2.26 Діаграма взаємодії клієнта та сервера при автентифікації

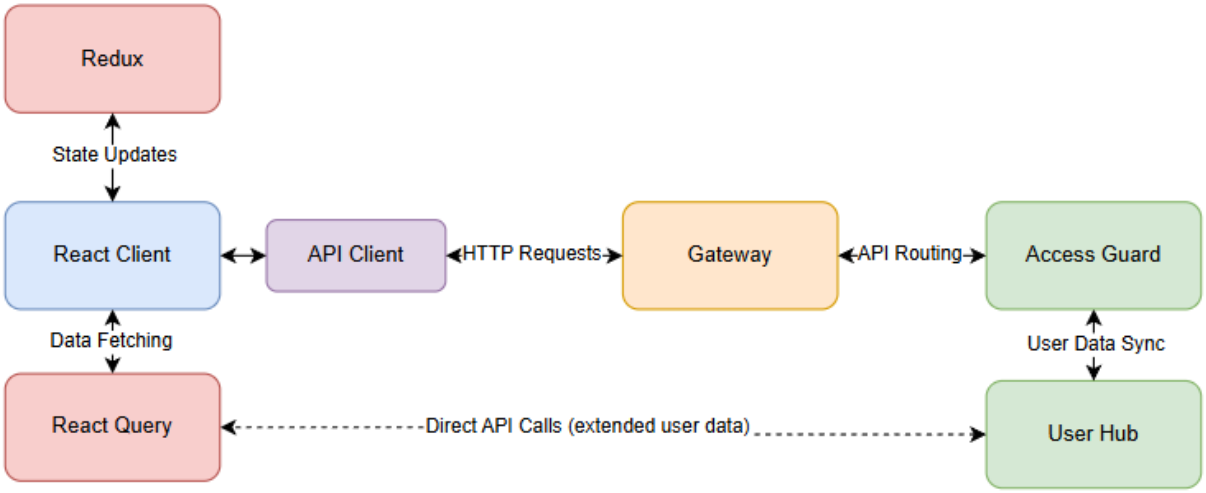


Рис. 2.27 Загальна схема взаємодії компонентів системи

2.5.3 Архітектура системи моніторингу та сповіщень

Система моніторингу Access Guard забезпечує комплексне відстеження активності користувачів, продуктивності роботи сервісу та потенційних безпекових інцидентів. Адміністративний інтерфейс надає можливість відстежувати статистику за обраними часовими періодами з детальною інформацією про активність користувачів у кожному з інтегрованих додатків.

Архітектура системи моніторингу складається з трьох ключових компонентів:

1. *Збір даних про активність* – реалізовано за допомогою аспектно-орієнтованого програмування, що дозволяє неінвазивно фіксувати всі спроби автентифікації та авторизації в системі. Центральною сутністю для збереження даних є клас *UserVisitStatistics*, який фіксує:
   - ідентифікатор користувача;
   - часову мітку події;
   - IP-адресу джерела запиту;
   - інформацію про клієнтське програмне забезпечення;
   - URL запитуваного ресурсу;
   - метод автентифікації;
   - результат операції;
   - детальну інформацію про помилки (за наявності);
2. *Аналіз та обробка статистики* – система зберігає захищені дані про активність у спеціалізованих таблицях з оптимізованими індексами для швидкого пошуку та аналізу. Реалізовано механізми агрегації статистичних даних для швидкого обчислення ключових показників безпеки;
3. *Візуалізація та адміністрування* – адміністративний інтерфейс надає інструменти для:
   - фільтрації даних за часовими періодами;
   - перегляду активності за додатками;
   - аналізу поведінки окремих користувачів;
   - виявлення аномальної активності;

- керування блокуванням підозрілих облікових записів.

Система моніторингу фокусується на виявленні потенційних інцидентів безпеки:

1. Послідовні невдалі спроби автентифікації з однієї IP-адреси;
2. Нетипова активність користувачів у незвичний час;
3. Спроби доступу до ресурсів без належних прав;
4. Підозрілі шаблони використання API.

Для оцінки продуктивності та стабільності роботи сервісу в архітектурі моніторингу передбачено збір технічних метрик:

1. *Метрики віртуальної машини* – показники використання пам'яті, навантаження потоків, ефективності збирання сміття
2. *Метрики API* – статистика запитів за статусами відповіді, часом обробки та типами операцій
3. *Бізнес-метрики* – показники, специфічні для сфери автентифікації:

- співвідношення успішних та невдалих спроб входу;
- інтенсивність оновлення токенів;

3 ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ

3.1. Методологія та стратегія тестування

Тестування мікросервісної архітектури системи Access Guard потребує комплексного підходу через розподілений характер системи. У процесі розробки застосовано різні рівні тестування, що забезпечують якість та надійність компонентів.

Для модульного тестування використано технології JUnit 5 та Mockito. JUnit 5 забезпечує основну структуру для написання тестів, а Mockito дозволяє створювати мок-об'єкти для ізоляції тестованих компонентів від їхніх залежностей. Для тестування реактивних потоків застосовано StepVerifier, що є особливо важливим для системи Access Guard, побудованої на реактивному підході. WebTestClient використовується для тестування веб-контролерів без необхідності запуску повноцінного сервера.

3.1.1. Результати модульного тестування

Таблиця 3.1

Покриття модульними тестами компонентів Access Guard

Компонент	**Кількість тестів**	**Тестове покриття**	**Критичність**
AuthController	18	96.2%	Висока
UserController	12	91.8%	Висока
ApplicationController	14	93.5%	Висока
DeviceController	8	89.7%	Середня
SettingController	4	87.2%	Середня
UserApplicationController	9	92.1%	Висока

Таблиця 3.1

Покриття модульними тестами компонентів Access Guard

Компонент	**Кількість тестів**	**Тестове покриття**	**Критичність**
AuthService	24	95.3%	Критична
UserService	22	93.6%	Висока
ApplicationService	16	90.8%	Висока
TokenService	15	94.7%	Критична
DeviceService	10	88.9%	Середня
MattermostAuthService	14	92.3%	Висока
AccessControlService	18	93.5%	Висока
UserApplicationService	12	91.2%	Висока
Маппери даних	28	96.8%	Середня
Репозиторії	34	88.5%	Середня

У процесі розробки досягнуто високих показників покриття коду тестами, особливо для компонентів із критичною та високою важливістю для системи (таблиця 3.1). Для компонентів, відповідальних за безпеку та автентифікацію, встановлено підвищені вимоги до тестового покриття.

Таблиця 3.2

Порівняння модульного тестування мікросервісів системи

Метрика	**Access Guard**	**User Hub**
Кількість тестів	192	142
Загальне покриття	93.4%	89.6%
Час виконання тестів	4 сек 630 мс	2 сек 681 мс

Результати тестування для User Hub та Access Guard відображено на рисунку 3.1, 3.2:

Рис. 3.1 Результати модульного тестування сервісу User Hub

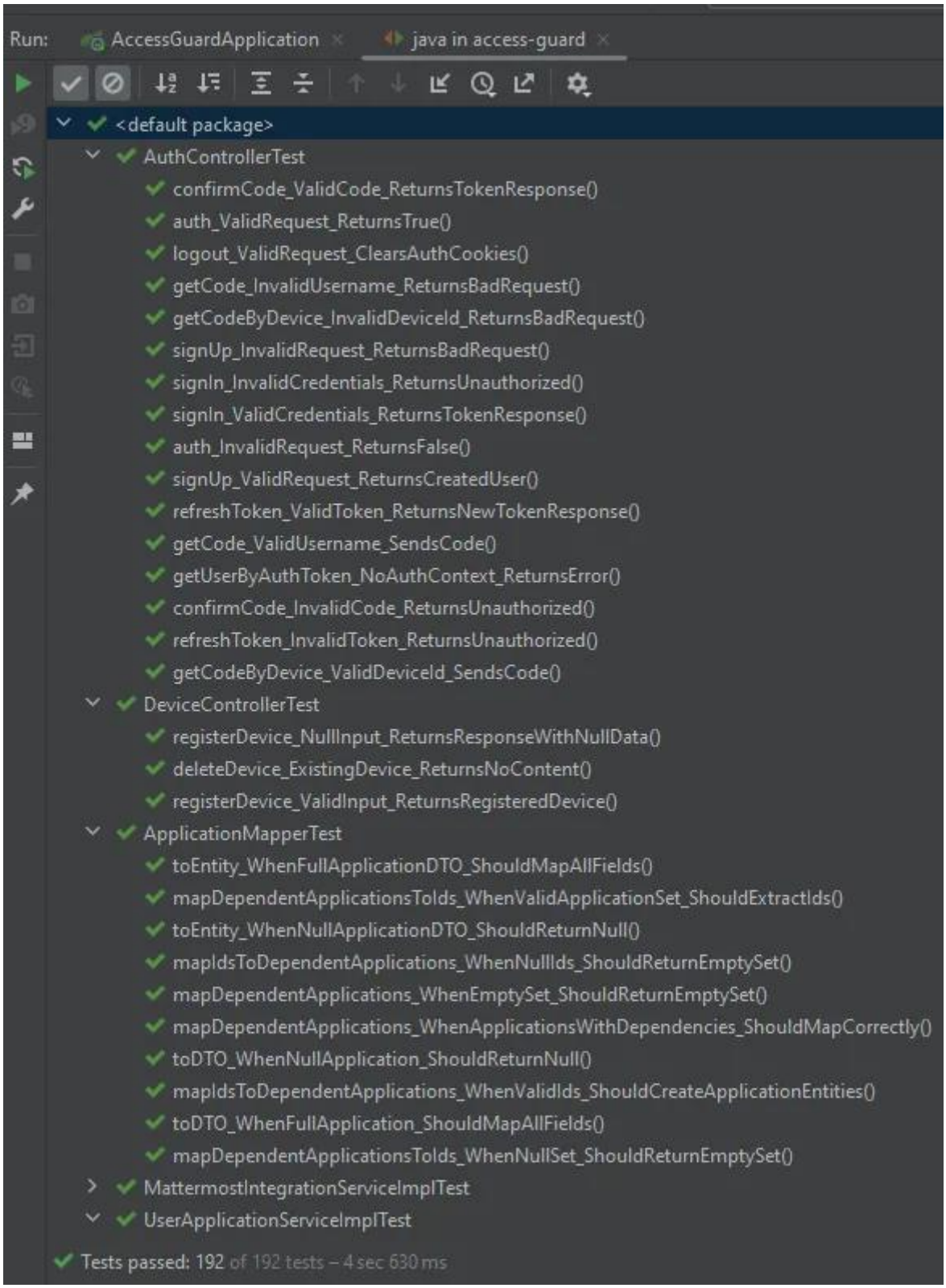


Рис. 3.2 Результати модульного тестування сервісу Access Guard

3.1.2. Інтеграційне тестування

Інтеграційне тестування спрямоване на перевірку взаємодії між різними компонентами системи. Основну увагу приділено перевірці взаємодії між

мікросервісами та інтеграції з зовнішніми системами, зокрема Mattermost.

Для тестування інтеграції з Mattermost розроблено спеціалізовані тести, що перевіряють процес отримання інформації про користувачів та надсилання повідомлень. Тести для інтеграції між Access Guard та User Hub перевіряють обмін даними про користувачів, їхні права доступу та додатки.

Окрему увагу приділено тестуванню асинхронної взаємодії через RabbitMQ, що є ключовим для обміну інформацією між мікросервісами. Розроблено тести для перевірки коректності публікації та споживання повідомлень, а також обробки різних типів подій.

3.2. Розгортання та інтеграція системи

3.2.1. Контейнеризація та оркестрація

Система Access Guard розгортається у контейнерному середовищі з використанням Docker та Docker Compose. Цей підхід забезпечує ізоляцію компонентів, спрощує процес розгортання та підвищує портативність системи.

Для кожного мікросервісу створено Dockerfile, що визначає процес збірки та конфігурацію середовища виконання. Структура Dockerfile побудована за принципом багатоетапної збірки (рисунок 3.3), що дозволяє оптимізувати розмір кінцевого образу:

```
1  # Етап збірки
2  FROM maven:3.8-openjdk-17 AS build
3  WORKDIR /app
4  COPY pom.xml .
5  COPY src ./src
6  RUN mvn clean package -DskipTests
7
8  # Етап запуску
9  FROM openjdk:17
10 WORKDIR /
11 COPY --from=build /app/target/*.jar app.jar
12 RUN useradd -m myuser
13 USER myuser
14 CMD java -jar -Dspring.profiles.active=prod app.jar
```

Рис. 3.3 Структура Dockerfile

Такий підхід забезпечує:

- менший розмір кінцевого образу через видалення непотрібних артефактів збірки;
- підвищення безпеки через запуск від імені непривілейованого користувача;
- стандартизацію процесу збірки та розгортання.

Конфігурація Docker Compose структурована за функціональними групами (рисунок 3.4):

1. Бази даних та інфраструктура:
   a. PostgreSQL – для зберігання даних;
   b. RabbitMQ – для асинхронної комунікації;
2. Мікросервіси:
   a. Discovery Server – для реєстрації та виявлення сервісів;
   b. User Hub – для управління користувачами;
   c. Access Guard – для автентифікації та авторизації;
   d. Gateway – для маршрутизації запитів;
3. Система моніторингу:
   a. Prometheus – для збору метрик;
   b. Grafana – для візуалізації метрик;
   c. Loki – для збору та аналізу логів;
   d. Zipkin – для трасування розподілених запитів.

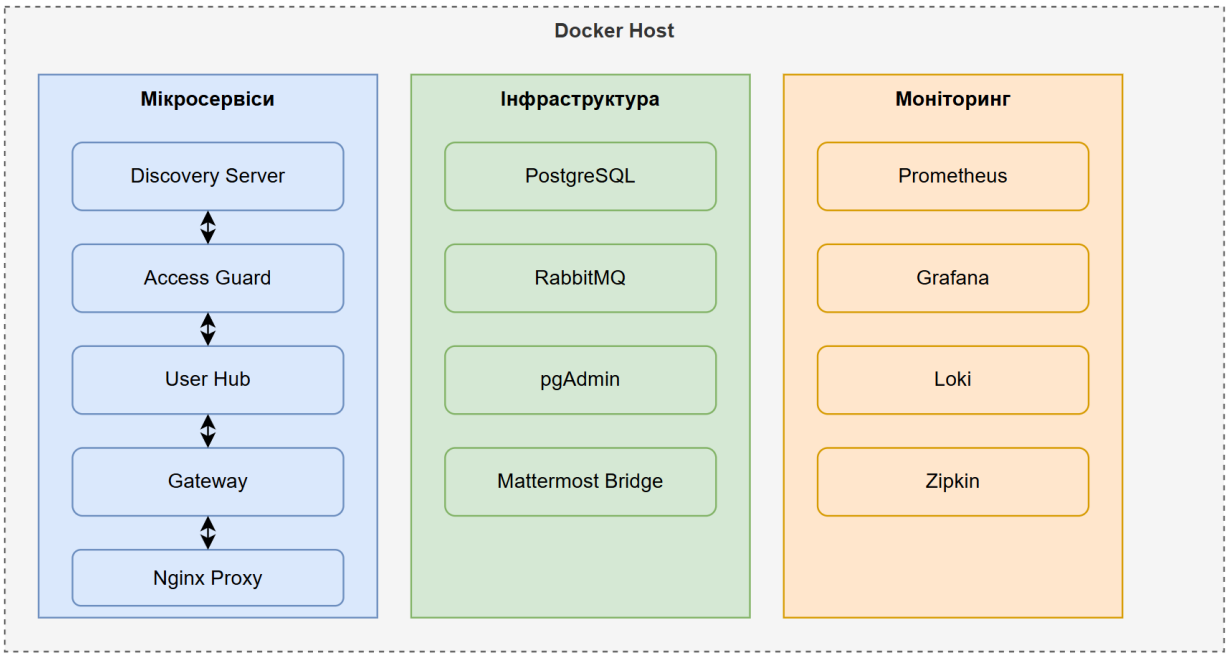


Рис. 3.4 Архітектура розгортання системи

3.2.2. Ініціалізація бази даних

Для забезпечення правильної ініціалізації бази даних при першому запуску системи використовуються спеціалізовані SQL-скрипти. Ці скрипти створюють необхідні таблиці, встановлюють індекси та обмеження цілісності, а також додають початкові дані.

Особливу увагу приділено налаштуванню API-ключів для міжсервісної комунікації (рисунок 3.5):

```
-- Налаштування API-ключів для додатків за їх назвами
UPDATE application
SET api_key = 'ключ для user-hub',
    updated_at = extract(epoch from now()) * 1000
WHERE spring_application_name = 'user-hub';

UPDATE application
SET api_key = 'ключ для prometheus',
    updated_at = extract(epoch from now()) * 1000
WHERE spring_application_name = 'prometheus';
```

Рис. 3.5 Налаштуванню API-ключів при першому запуску додатків

Для управління еволюцією схеми бази даних використовується Liquibase – інструмент для відстеження та застосування змін. Цей підхід забезпечує:

- версіонування схеми бази даних;
- атомарність міграцій;
- автоматичне застосування змін при оновленні системи;
- можливість відстеження історії змін.

3.2.3. Мережева конфігурація та комунікація

Усі сервіси об'єднані в мережу *app-network*, що забезпечує ізоляцію комунікації (таблиця 3.6):

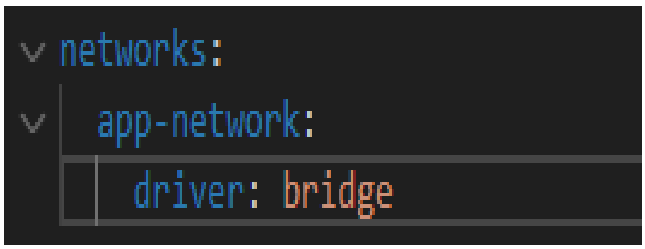


Рис. 3.6 Створення спільної мережі

Таблиця 3.3

Мережеві налаштування компонентів системи

Компонент	Порт	Призначення
Gateway	8080	Єдина точка входу для API-запитів
Access Guard UI	80	Веб-інтерфейс для керування системою
PostgreSQL	5432	Доступ до бази даних
Grafana	3000	Перегляд метрик та логів
Prometheus	9090	Доступ до метрик
RabbitMQ	5672	Основний порт для обміну повідомленнями
RabbitMQ Management	15672	Веб-інтерфейс керування RabbitMQ
Zipkin	9411	Аналіз розподілених трасувань
Discovery Server (Eureka)	9761	Реєстрація та виявлення сервісів

Таблиця 3.4

Принципи архітектури комунікації між компонентами

Принцип	Опис	Технічна реалізація
Взаємодія через Gateway	Всі зовнішні запити проходять через єдину точку входу	Використання Spring Cloud Gateway для маршрутизації запитів
Асинхронний обмін повідомленнями	Використовується для сценаріїв, що не потребують миттєвої відповіді	RabbitMQ як брокер повідомлень для обміну подіями між сервісами
Синхронна взаємодія між сервісами	Використовується для сценаріїв, що потребують миттєвої відповіді	HTTP-запити з використанням WebClient та RestTemplate
Реєстрація в Discovery Server	Автоматичне виявлення сервісів	Реєстрація в Eureka за URL http://eureka:password@discovery-server:9761/eureka

3.3. Моніторинг та документування системи

3.3.1. Система моніторингу

Для забезпечення постійного спостереження за роботою системи Access Guard впроваджено комплексний підхід до моніторингу на базі сучасних інструментів збору та аналізу даних. Архітектура моніторингу побудована за принципом багаторівневого збору інформації з використанням інтегрованого стеку технологій, що дозволяє відстежувати як технічні показники, так і бізнес-метрики системи.

За допомогою впровадженого інструментарію налаштовано збір та візуалізацію наступних категорій метрик:

1. Технічні метрики віртуальної машини Java:
   a. Використання heap та non-heap пам'яті;
   b. Кількість та стан потоків виконання (JVM Live Threads);
   c. Час збирання сміття (GC Pause Time);

d. Загальне навантаження на CPU;
e. Кількість завантажених класів;

2. Метрики веб-запитів:
   a. Кількість HTTP-запитів за ендпоінтами;
   b. Розподіл часу відгуку;
   c. Статистика за HTTP-кодами відповідей;
   d. Розподіл запитів за сервісами;

3. Бізнес-метрики автентифікації та безпеки:
   a. Статистика проходження запитів через ланцюжок фільтрів безпеки;
   b. Кількість успішних та невдалих спроб автентифікації;
   c. Статистика використання різних методів автентифікації.

За допомогою Zipkin реалізовано розподілене трасування запитів, що дозволило відстежувати шлях та час виконання запитів через всі компоненти мікросервісної архітектури (рисунок 3.7).

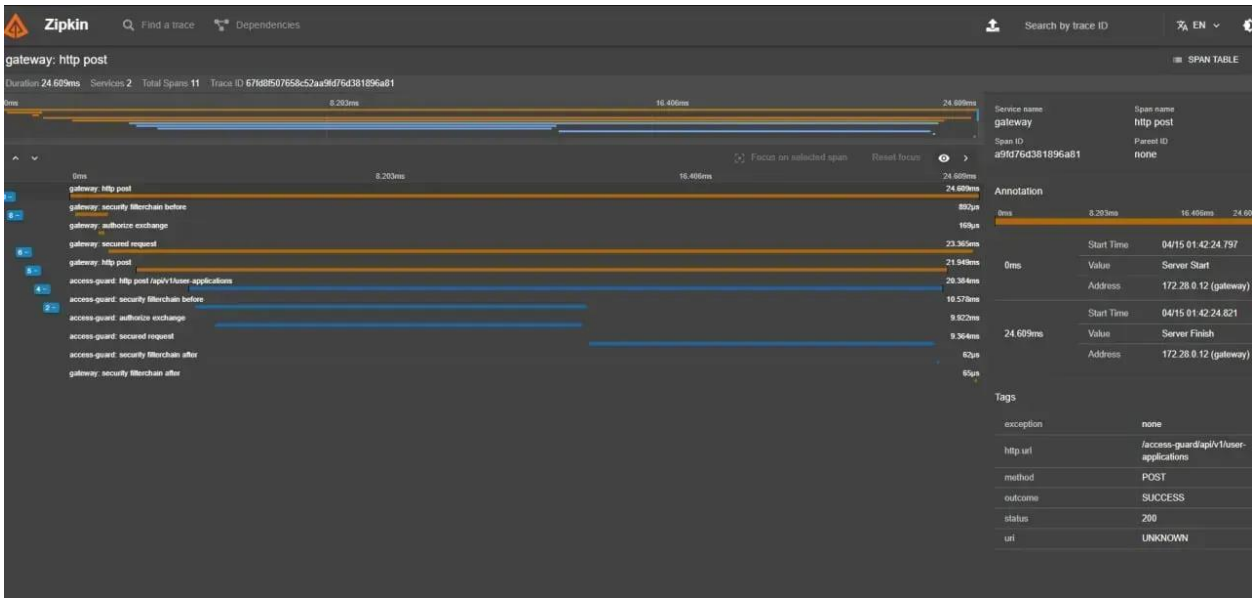


Рис. 3.7 Візуалізація трасування запитів за допомогою Zipkin

Для збору метрик використано Prometheus з подальшою візуалізацією в Grafana, що дозволило створити інформативні дашборди для моніторингу продуктивності системи. На Рис. 3.8 представлено дашборд моніторингу HTTP-запитів (рисунок 3.8), який відображає частоту запитів, час відгуку, кількість

помилок та успішних запитів за різними сервісами.

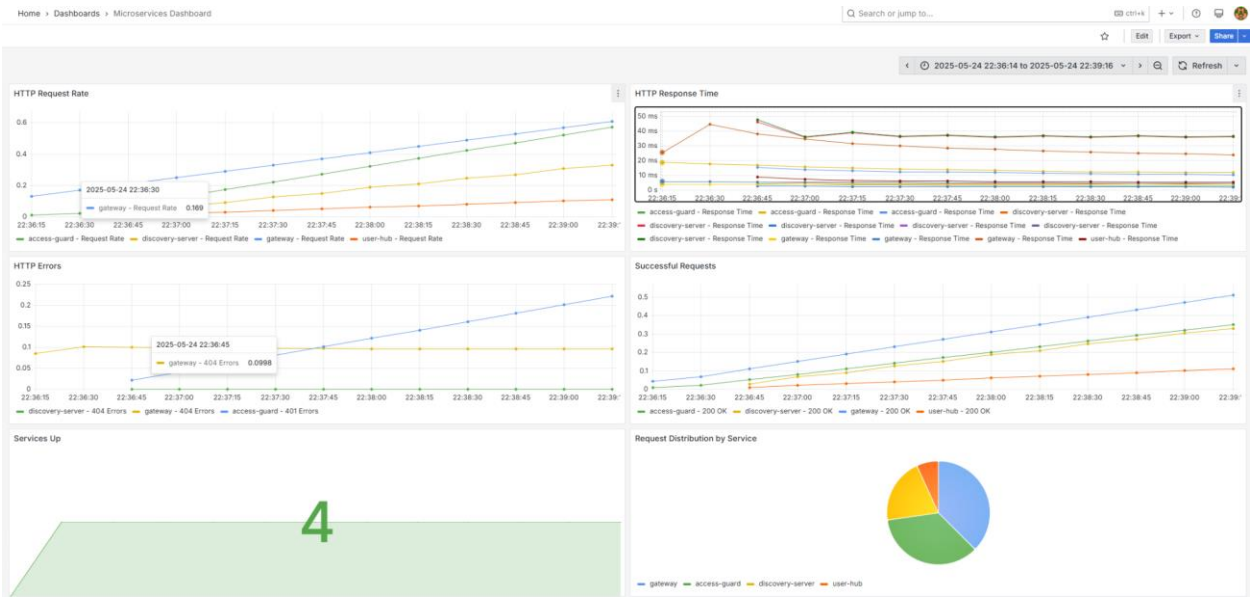


Рис. 3.8 Дашборд моніторингу HTTP-запитів та помилок у Grafana

Для моніторингу технічного стану JVM налаштовано окремий дашборд (рисунок 3.9), який відображає використання пам'яті, активність збирача сміття, навантаження на CPU та інші важливі показники роботи віртуальної машини Java для кожного мікросервісу.

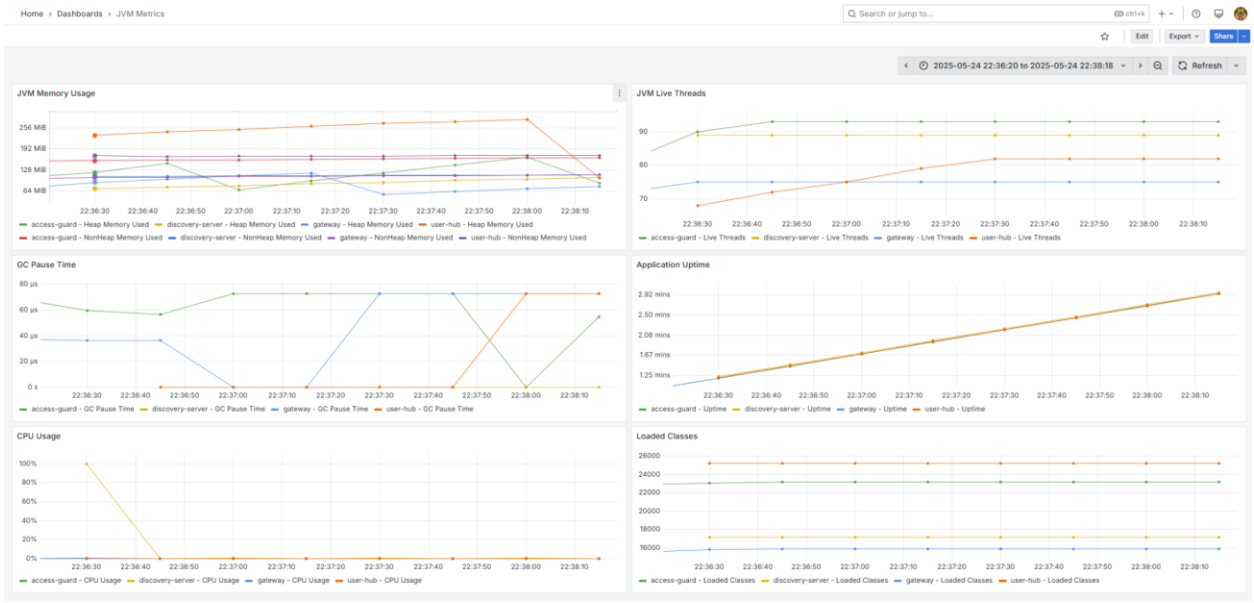


Рис. 3.9. Дашборд моніторингу JVM метрик (пам'ять, потоки, GC, CPU) у Grafana

Система логування реалізована з використанням Loki, що забезпечило

ефективний збір, пошук та аналіз журнальних записів. На (рисунок 3.10) представлено дашборд логування, який відображає статистику журнальних записів з розподілом за рівнями та сервісами, а також дозволяє переглядати конкретні записи логів.

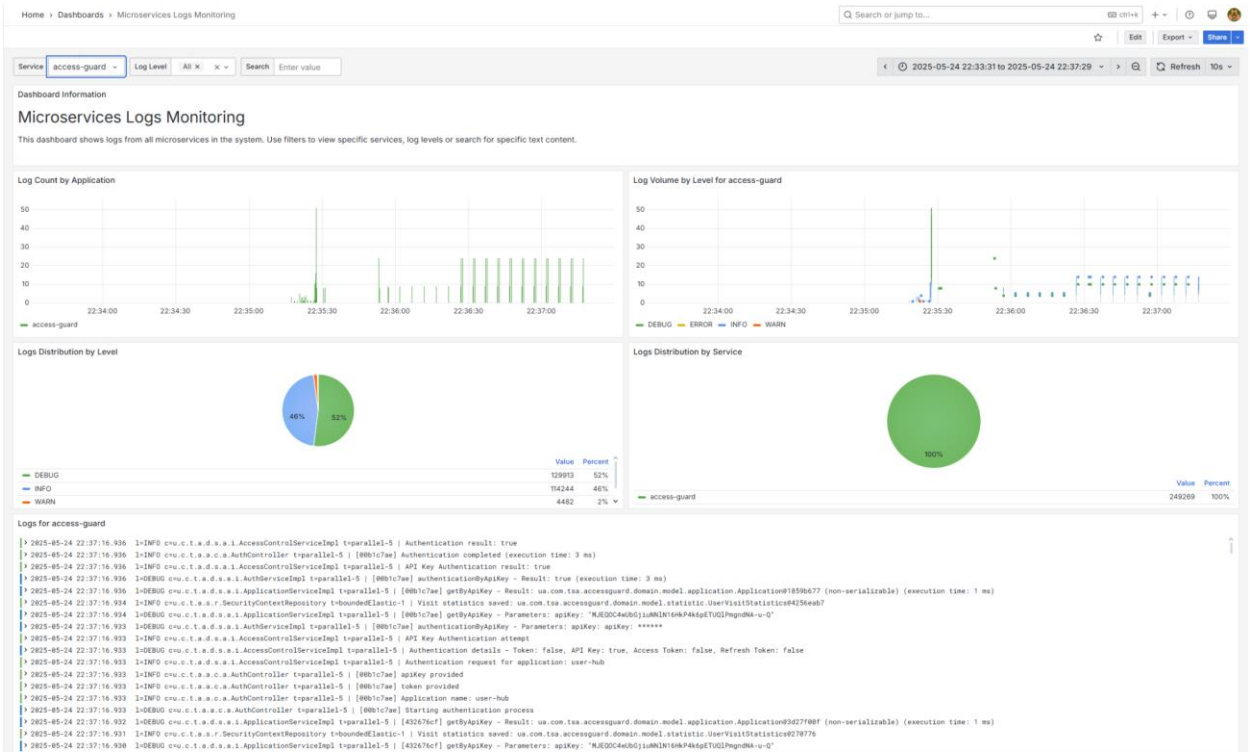


Рис. 3.10. Дашборд аналізу логів access-guard з використанням Loki у Grafana

Впроваджена система моніторингу дозволила значно підвищити спостережуваність системи Access Guard, забезпечивши можливість оперативного виявлення та усунення проблем, аналізу продуктивності окремих компонентів та системи в цілому, а також збору статистичних даних для подальшої оптимізації.

3.3.2. Документування API

Для забезпечення ефективної інтеграції та використання системи реалізовано документування API за допомогою Swagger/OpenAPI. Документація доступна за URL */swagger-ui* для кожного сервісу та агрегується в Gateway для зручного доступу.

Таблиця 3.5

Структура API-документації системи Access Guard

Категорія API	**Основні ендпоінти**	**Призначення**	**Рівень доступу**
Автентифікація	/api/v1/auth/sign-in	Вхід у систему	Публічний
Автентифікація	/api/v1/auth/sign-in/mattermost/code	Запит коду через Mattermost	Публічний
Автентифікація	/api/v1/auth/sign-in/mattermost/confirm	Підтвердження коду	Публічний
Автентифікація	/api/v1/auth/token	Оновлення токену	Публічний
Автентифікація	/api/v1/auth/logout	Вихід із системи	Авторизований
Користувачі	/api/v1/users	Керування користувачами	Адміністративний
Користувачі	/api/v1/users/{id}	Операції з конкретним користувачем	Адміністративний
Користувачі	/api/v1/users/sync	Синхронізація з User Hub	Адміністративний
Додатки	/api/v1/applications	Керування додатками	Адміністративний
Додатки	/api/v1/applications/{id}	Операції з конкретним додатком	Адміністративний
Додатки	/api/v1/applications/api-key/generate/{id}	Генерація API-ключа	Адміністративний
Пристрої	/api/v1/devices/register	Реєстрація пристрою	Авторизований
Пристрої	/api/v1/devices/{deviceId}	Керування пристроями	Авторизований

Нижче наведено скріншоти Swagger-документації для User Hub та Access Guard (рисунок 3.11, 3.12):

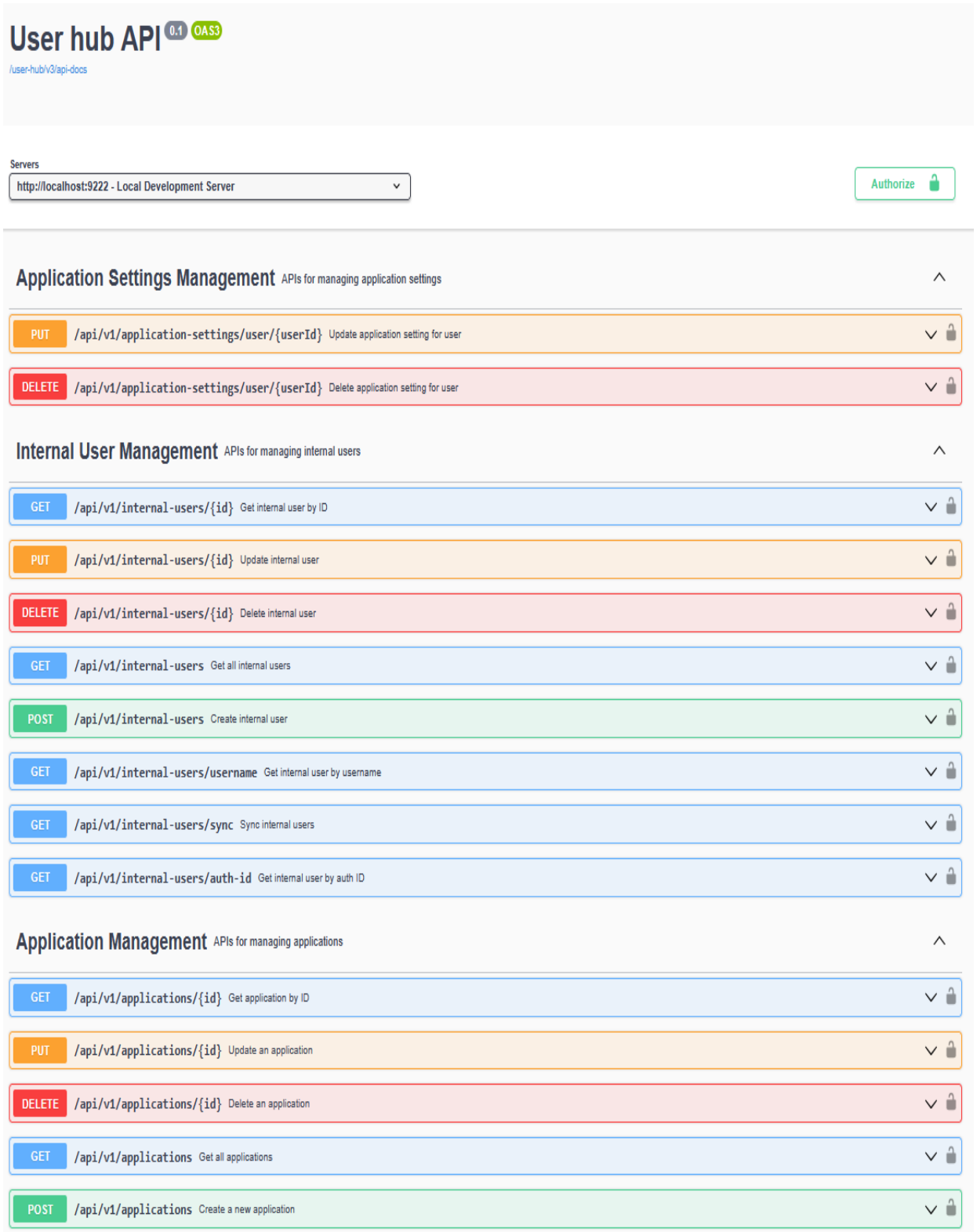


Рис. 3.11 Swagger-документація User Hub API

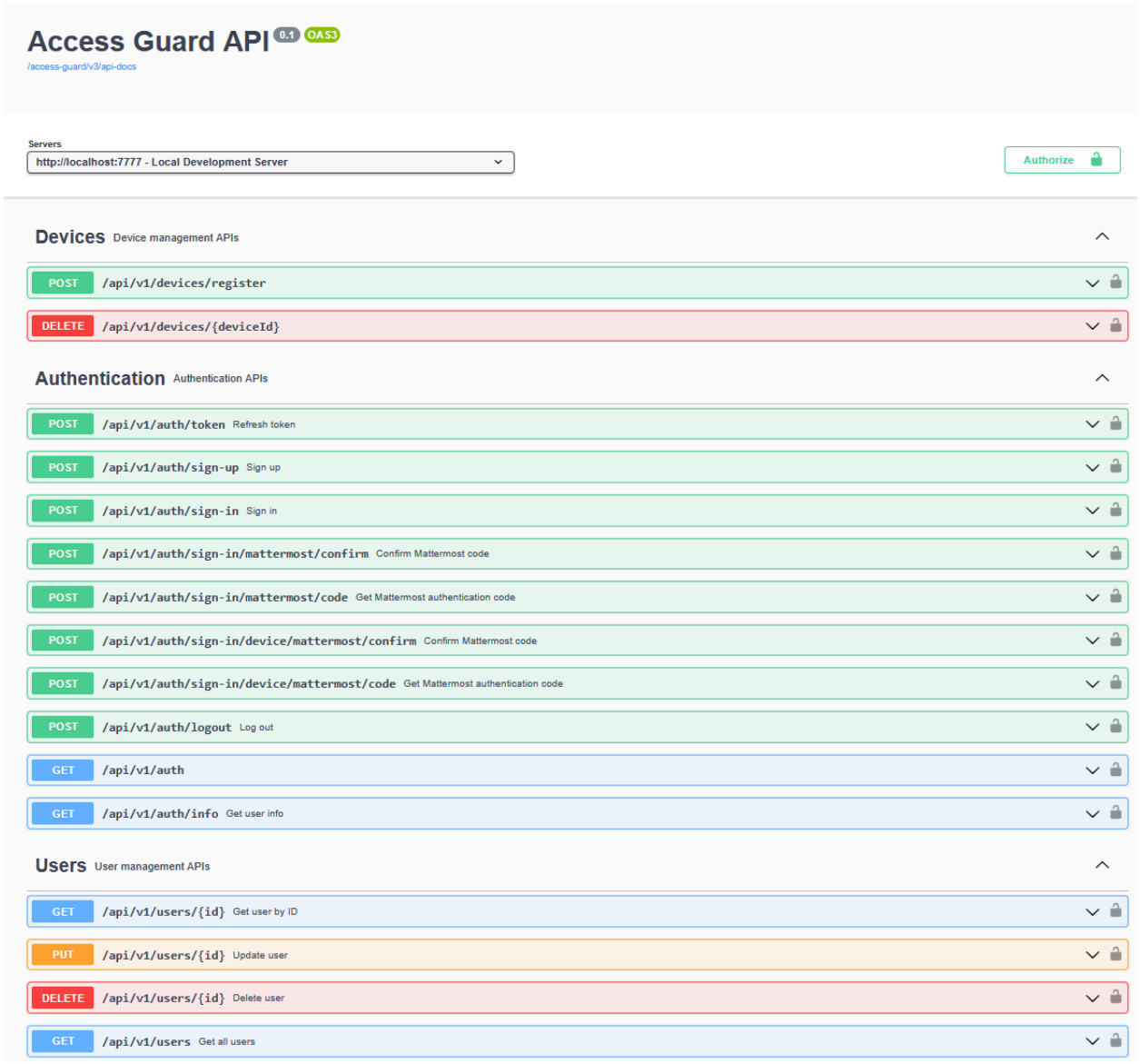


Рис. 3.12 Swagger-документація Access Guard API

Кожен ендпоінт документовано з описом його функціональності, параметрів запитів та відповідей, а також можливих кодів відповідей (таблиця 3.5). Для підтримки актуальності документації впроваджено процес автоматичної перевірки відповідності документації реальному API.

3.3.3. Середовища розгортання

Для забезпечення якості та надійності системи розгортання відбувається в трьох окремих середовищах:

1. Середовище розробки (Development) – призначене для тестування нових функцій під час розробки. У цьому середовищі розробники мають можливість швидко впроваджувати зміни перед інтеграцією з основною кодовою

базою;

2. Тестове середовище (Test) – ізольоване середовище, яке максимально наближене до виробничого. Тут проводиться комплексне тестування після завершення розробки нових функцій;

3. Виробниче середовище (Production) – середовище, де система фактично експлуатується. Оновлення у цьому середовищі відбувається лише після успішного проходження всіх етапів тестування.

Такий підхід до організації розгортання забезпечує поступове підвищення якості програмного забезпечення та мінімізує ризики виникнення проблем у виробничому середовищі.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

4.1. Оцінка відповідності розробленої системи поставленим вимогам

4.1.1. Реалізація функціональних вимог

Відповідно до поставлених завдань було реалізовано наступні функціональні компоненти:

1. Управління користувачами – система забезпечує функціональність створення, редагування та видалення користувачів через компоненти UserController та UserService з відповідними API-ендпоінтами;
2. Ієрархія ролей – впроваджено систему ролей від ROLE_ROOT до ROLE_BLOCKED, що забезпечує управління правами доступу користувачів;
3. Блокування облікових записів – реалізовано функціональність тимчасового та постійного блокування користувачів із логуванням таких дій для подальшого аудиту;
4. JWT-автентифікація – впроваджено механізм автентифікації з використанням JWT-токенів та HTTP-only cookies для захисту від XSS-атак;
5. Двофакторна автентифікація через Mattermost – реалізовано додатковий рівень безпеки з використанням корпоративного месенджера для підтвердження ідентичності;
6. Автентифікація пристроїв – розроблено систему реєстрації та верифікації пристроїв користувачів для підвищення безпеки доступу;
7. Управління API-ключами – забезпечено генерацію, оновлення та контроль API-ключів для міжсервісної автентифікації з компонентом ApplicationController;
8. Механізм оновлення токенів – реалізовано механізм автоматичного оновлення токенів доступу через refresh-токени для підтримки користувацьких сесій;

9. Обмеження доступу до додатків – впроваджено можливість налаштування доступу користувачів до окремих додатків через компонент UserApplicationController;

10. Реєстрація та управління додатками – система підтримує функціональність реєстрації, оновлення та видалення додатків у системі безпеки;

11. Підтримка різних типів додатків – реалізовано диференціацію додатків на INTERNAL та EXTERNAL з відповідними налаштуваннями безпеки;

12. Аудит змін – впроваджено механізм аудиту модифікацій у ключових таблицях бази даних з використанням Hibernate Envers;

13. Управління сесіями – реалізовано обмеження кількості одночасних сесій користувача з автоматичним видаленням найстаріших при перевищенні ліміту;

14. Автоматична реєстрація – забезпечено механізм автоматичної реєстрації нових сервісів через Discovery Server та їх інтеграцію з системою безпеки;

15. Асинхронна комунікація – впроваджено обмін повідомленнями через RabbitMQ для сповіщення про зміни в правах доступу та користувацьких даних;

16. Адміністративний інтерфейс – розроблено веб-інтерфейс для централізованого управління користувачами, додатками та їх правами доступу.

4.1.2. Виконання нефункціональних вимог

Система також охоплює встановлені нефункціональні вимоги:

1. Шифрування паролів – реалізовано криптографічне шифрування паролів за допомогою BCrypt з можливістю налаштування фактора складності;

2. Захищена комунікація – взаємодії відбуваються через HTTPS, що забезпечує конфіденційність та цілісність даних при передачі;

3. Захист від атак – впроваджено механізми захисту від CSRF, XSS та інших типів атак через спеціалізовані фільтри та заголовки безпеки;

4. Обмеження токенів – реалізовано налаштування життєвого циклу токенів з можливістю індивідуальних параметрів для різних типів користувачів;

5. Безпечне зберігання даних – впроваджено механізми маскування чутливих даних у логах та захищене зберігання критичної інформації;

6. Швидкодія – тестування зафіксувало обробку запитів автентифікації з латентністю в межах встановлених вимог (до 200 мс);

7. Масштабованість – система підтримує одночасні активні сесії в межах встановлених вимог з можливістю горизонтального масштабування;

8. Балансування навантаження – реалізовано розподіл запитів між екземплярами сервісів через інтеграцію з Discovery Server;

9. Висока доступність – архітектурні рішення спрямовані на забезпечення встановленого рівня доступності;

10. Відмовостійкість – впроваджено механізми автоматичного відновлення після збоїв;

11. Повторні спроби – реалізовано механізми повторних спроб при тимчасових проблемах зв'язку з експоненційною затримкою;

12. Документація API – забезпечено документацію API через Swagger/OpenAPI з можливістю тестування ендпоінтів;

13. Багаторівневе логування – впроваджено систему логування з градацією важливості подій та маскуванням чутливих даних;

14. Інтеграція з моніторингом – забезпечено підтримку Prometheus, Grafana та інших інструментів для моніторингу продуктивності системи;

15. Трасування запитів – реалізовано трасування розподілених запитів через Zipkin для аналізу взаємодій;

16. Багатомовний інтерфейс – адміністративний інтерфейс підтримує українську та англійську мови з можливістю розширення.

Метрики продуктивності, зібрані під час тестування, відображають відповідність системи встановленим вимогам щодо часу відгуку. Відповідно до

результатів тестування, час обробки запитів автентифікації не перевищує встановлений поріг у 200 мс при навантаженні до 1000 одночасних сесій.

4.2. Порівняльний аналіз із наявними рішеннями

Порівняння розробленого мікросервісу Access Guard із аналогічними рішеннями виявляє ряд відмінностей:

У порівнянні з Keycloak, Access Guard демонструє нижчі показники використання пам'яті (приблизно 450 МБ проти 1,2 ГБ при аналогічному навантаженні) завдяки реактивній архітектурі. Це потенційно дозволяє ефективніше використовувати обчислювальні ресурси інфраструктури.

На відміну від Auth0, Access Guard не залежить від зовнішніх сервісів і дозволяє розгортання в локальній інфраструктурі, що може бути перевагою для організацій із суворими вимогами до контролю даних та відповідності регуляторним нормам.

Порівняно з IdentityServer, який переважно орієнтований на екосистему .NET, Access Guard використовує технології, що забезпечують сумісність із різними технологічними стеками через стандартні протоколи та REST API.[11]

Відмінною характеристикою Access Guard є інтеграція з корпоративними засобами комунікації, зокрема Mattermost, для реалізації двофакторної автентифікації.

4.3. Перспективи розвитку системи

4.3.1. Розширення методів автентифікації

Впровадження додаткових методів автентифікації:

1. Інтеграція з корпоративними каталогами Active Directory за протоколом LDAP, що забезпечить єдину систему управління обліковими записами та спростить адміністрування користувачів у великих організаціях;

2. Впровадження Telegram як альтернативного каналу для двофакторної автентифікації та оперативних сповіщень про підозрілі активності;

3. Біометрична автентифікація – впровадження підтримки розпізнавання відбитків пальців та обличчя для мобільних додатків.

4.3.2. Удосконалення системи авторизації

Розширення моделі авторизації:

1. Атрибутивна модель контролю доступу (ABAC) – додавання можливості визначення правил доступу на основі атрибутів користувача, ресурсу, дії та контексту;

2. Розширений контроль сесій – впровадження детального управління сесіями для кожного користувача та груп користувачів з можливістю встановлення індивідуальних обмежень, політик безпеки та тривалості дії;

3. Механізми адаптивної автентифікації, що регулюватимуть рівень безпеки на основі аналізу контексту запиту (IP-адреса, час доступу, пристрій).

4.3.3. Удосконалення моніторингу та аналітики безпеки

Розширення можливостей моніторингу:

1. Розширений геолокаційний аналіз входів для виявлення підозрілої активності на основі незвичних місць доступу;

2. Застосування технологій машинного навчання для виявлення аномальної поведінки користувачів;

3. Розширена аналітика безпеки – розробка спеціалізованих дашбордів для візуалізації трендів та інцидентів безпеки.

ВИСНОВОК

У рамках виконання кваліфікаційної роботи було розроблено мікросервіс авторизації Access Guard для управління доступом до корпоративних додатків, що функціонує в мікросервісній архітектурі. Система забезпечує централізоване управління автентифікацією та авторизацією користувачів, надаючи єдину точку контролю та моніторингу доступу до розподілених ресурсів.

У процесі дослідження було виконано аналіз предметної області, розглянуто існуючі рішення для управління доступом, оцінено їхні переваги та обмеження. На основі проведеного аналізу було визначено ключові вимоги до розроблюваної системи та обґрунтовано доцільність створення нового рішення, орієнтованого на корпоративне мікросервісне середовище.

Спроєктована архітектура системи базується на принципах розробки програмного забезпечення та враховує специфіку мікросервісного підходу. Архітектурним рішенням стало використання реактивного підходу на базі Spring WebFlux, спрямованого на забезпечення пропускної здатності та використання обчислювальних ресурсів.

У результаті розробки було реалізовано функціональні вимоги, включаючи створення та управління користувачами, впровадження ієрархічної системи ролей, різні методи автентифікації (JWT, Mattermost, пристрої), управління API-ключами, механізми оновлення токенів, налаштування доступу до додатків, аудит змін, обмеження сесій, автоматичну реєстрацію сервісів, асинхронну комунікацію та адміністративний інтерфейс.

Система також включає реалізацію нефункціональних вимог: шифрування паролів, захищена комунікація, захист від атак, оновлення токенів, безпечне зберігання даних, оптимізація часу відгуку, масштабованість, балансування навантаження, доступність, відмовостійкість, механізми повторних спроб,

документація API, логування, інтеграція з моніторингом та підтримка багатомовності.

Проведене тестування дозволило оцінити відповідність розробленої системи поставленим вимогам. Модульні та інтеграційні тести охопили критичні компоненти системи. Результати тестування продуктивності продемонстрували характеристики системи при обробці паралельних запитів автентифікації.

Розроблений мікросервіс Access Guard характеризується меншими вимогами до ресурсів, локальним контролем над інфраструктурою даних, інтеграцією з різними технологічними стеками та можливістю взаємодії з корпоративними засобами комунікації для двофакторної автентифікації.

Визначені перспективи розвитку системи включають: інтеграцію з Active Directory, впровадження Telegram як альтернативного каналу для двофакторної автентифікації, підтримку FIDO2/WebAuthn, механізми адаптивної автентифікації, розширений контроль сесій для груп користувачів, геолокаційний аналіз входів та застосування машинного навчання для аналізу поведінки користувачів.

Розроблений мікросервіс Access Guard представляє рішення для централізованого управління доступом у мікросервісному середовищі відповідно до встановлених вимог щодо безпеки, продуктивності та масштабованості для корпоративних систем.

Робота пройшла апробацію. За її результатами були опубліковані наступні тези доповідей:

1. Олаба Д. Р. Управління ідентифікацією пристроїв як додатковий рівень захисту корпоративних систем. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (ІТ-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 306-308.

2. Олаба Д. Р. Безсесійна автентифікація на основі JWT-токенів у мікросервісній архітектурі. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (ІТ-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 304-306.

3. Олаба Д. Р. Реактивна архітектура мікросервісів на базі Spring WebFlux для високонавантажених систем. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (IT-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 166-169.

4. Олаба Д. Р., Худік Б.О. Проєктування та реалізація централізованої системи управління доступом у розподілених веб-додатках. // VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». – Київ: ДУІКТ, 2025. – С. 49-52.

5. Олаба Д. Р., Худік Б.О. Забезпечення безпеки даних у системі захисту Access Guard. // VI Науково-технічна конференція «Сучасний стан та перспективи розвитку IoT». – Київ: ДУІКТ, 2025. – (подано до друку).

ПЕРЕЛІК ПОСИЛАНЬ

1. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2021. 616 p.
2. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2022. 520 p.
3. OAuth 2.0. OAuth 2.0 Authorization Framework [Електронний ресурс] // OAuth. -- Режим доступу: https://oauth.net/2/ (дата звернення: 08.05.2025).
4. Walls C. Spring in Action, Sixth Edition. Manning Publications, 2022. 520 p.
5. Spilca L. Spring Security in Action. Manning Publications, 2021. 480 p.
6. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519 [Електронний ресурс] // IETF. -- Режим доступу: https://www.rfc-editor.org/rfc/rfc7519 (дата звернення: 08.05.2025).
7. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2022. 560 p.
8. Spring Security. Official Reference Documentation [Електронний ресурс] // Spring. -- Режим доступу: https://docs.spring.io/spring-security/reference/ (дата звернення: 12.05.2025).
9. Derhak M., Kovalchuk A. Reactive Programming in Modern Microservice Architectures. International Journal of Computer Networks & Communications. 2024. Vol. 16, no. 2. P. 45--58.
10. Baeldung. Control the Session with Spring Security [Електронний ресурс] // Baeldung. -- Режим доступу: https://www.baeldung.com/spring-security-session (дата звернення: 11.05.2025).
11. Vega D. How to Secure your REST APIs with Spring Security & JSON Web Tokens (JWTs) [Електронний ресурс] // Dan Vega. -- Режим доступу: https://www.danvega.dev/blog/spring-security-jwt (дата звернення: 12.05.2025).
12. OWASP. JSON Web Token Security Cheat Sheet [Електронний ресурс] // OWASP Foundation. -- Режим доступу: https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html (дата звернення: 23.04.2025).
13. Auth0. JWT Authentication: JSON Web Tokens [Електронний ресурс] // Auth0. -- Режим доступу: https://auth0.com/docs/secure/tokens/json-web-tokens (дата звернення: 23.04.2025).
14. Black Duck. Top 10 Spring Security Best Practices for Java Developers [Електронний ресурс] // Black Duck. -- Режим доступу: https://www.blackduck.com/blog/spring-security-best-practices.html (дата звернення: 10.05.2025).

15. Devops.dev. Spring Boot 3 + Spring Security 6 With JWT Authentication and Authorization [New 2024] [Електронний ресурс] // DevOps.dev. -- Режим доступу: https://blog.devops.dev/spring-boot-3-spring-security-6-with-jwt-authentication-and-authorization-new-2024-989323f29b84 (дата звернення: 12.05.2025).

16. Tutorialspoint. Spring Security Tutorial [Електронний ресурс] // Tutorialspoint. -- Режим доступу: https://www.tutorialspoint.com/spring_security/index.htm (дата звернення: 12.05.2025).

17. Voz.City. Вирішення проблеми з класом WebSecurityConfigurerAdapter в Spring Security [Електронний ресурс] // Voz.City. -- Режим доступу: https://voz.city/qna/vyrishennya-problemy-z-klasom-websecurityconfigureradapter-v-spring-security (дата звернення: 12.05.2025).

18. DOU.ua. Spring Professional Certification: мій досвід підготовки та поради [Електронний ресурс] // DOU.ua. -- Режим доступу: https://dou.ua/forums/topic/43410/ (дата звернення: 12.05.2025).

19. JavaRush. Як зробити авторизацію в Spring Boot та JWT [Електронний ресурс] // JavaRush. -- Режим доступу: https://javarush.com/ua/groups/posts/uk.3238.jak-zrobiti-avtorizacju-v-spring-boot-ta-jwt (дата звернення: 12.05.2025).

20. Spring Security. Spring Security Project Page [Електронний ресурс] // Spring. -- Режим доступу: https://spring.io/projects/spring-security (дата звернення: 12.05.2025).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка мікросервісу авторизації для управління доступом до корпоративних додатків

Виконав студент 4 курсу
групи ПД-44
Олаба Дмитро Романович
Керівник роботи
доктор філософії (PhD), доцент кафедри ІПЗ Худік Богдан Олександрович

Київ – 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Поліпшення уніфікованого керування доступом до корпоративних застосунків у мікросервісній архітектурі шляхом впровадження мікросервісу Access Guard, що підвищує рівень безпеки інформаційних ресурсів, спрощує адміністрування прав та забезпечує єдину точку контролю активності користувачів.

Об'єкт дослідження: Процес автентифікації та авторизації користувачів у децентралізованому середовищі мікросервісної архітектури корпоративних застосунків

Предмет дослідження: Технології побудови мікросервісу Access Guard для централізованої автентифікації, авторизації та аудиту доступу в розподіленій мікросервісній архітектурі.

ЗАВДАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Аналіз існуючих рішень для управління доступом та визначення функціональних вимог до системи
2. Проектування архітектури мікросервісу та створення інформаційної моделі бази даних
3. Реалізація JWT автентифікації з HTTP-only cookies та механізмом оновлення токенів
4. Розробка REST API сервісу та адміністративного веб-інтерфейсу
   Інтеграція з Mattermost для двофакторної автентифікації та автентифікації пристроїв
5. Контроль доступу на основі ролей та управління сесіями
6. Створення комплексної системи тестування та контейнеризація через Docker
7. Налаштування моніторингу (Prometheus, Grafana) та документування API

3

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Створення, редагування та видалення користувачів
2. Контроль доступу на основі ієрархії ролей користувачів
3. Блокування/розблокування облікових записів
4. Автентифікація через логін/пароль з JWT-токенами та HTTP-only cookies
5. Двофакторна автентифікація через Mattermost
6. Автентифікація пристроїв користувачів
7. Генерація, оновлення та управління API-ключами для міжсервісної автентифікації
8. Оновлення токенів через refresh-токени
9. Обмеження доступу до окремих додатків
10. Реєстрація та управління додатками в системі
11. Підтримка різних типів додатків (INTERNAL, EXTERNAL)
12. Здійснення аудиту змін у ключових таблицях бази даних
13. Обмеження кількості одночасних сесій користувача
14. Автоматична реєстрація нових сервісів через Discovery Server
15. Асинхронна комунікація через RabbitMQ
16. Веб-інтерфейс для управління користувачами та їх правами

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

1. Криптографічне шифрування паролів за допомогою BCrypt
2. Використання HTTPS для всіх комунікацій
3. Захист від CSRF, XSS та інших типів атак
4. Автоматичне знищення токенів після закінчення сесії
5. Маскування паролів та чутливих даних у логах системи
6. Обробка запитів автентифікації з латентністю не більше 200 мс
7. Підтримка до 1000 одночасних активних сесій
8. Горизонтальне масштабування через автоматичне створення додаткових Docker-інстансів при збільшенні навантаження
9. Доступність на рівні 99%
10. Автоматичне відновлення після збоїв
11. Механізми повторних спроб при тимчасових проблемах зв'язку
12. Детальна документація API через Swagger/OpenAPI
13. Багаторівневе логування з градацією важливості подій
14. Підтримка моніторингових інструментів (Prometheus, Grafana, Loki)
15. Трасування розподілених запитів через Zipkin
16. Підтримка української та англійської мов інтерфейсу

5

АНАЛІЗ АНАЛОГІВ

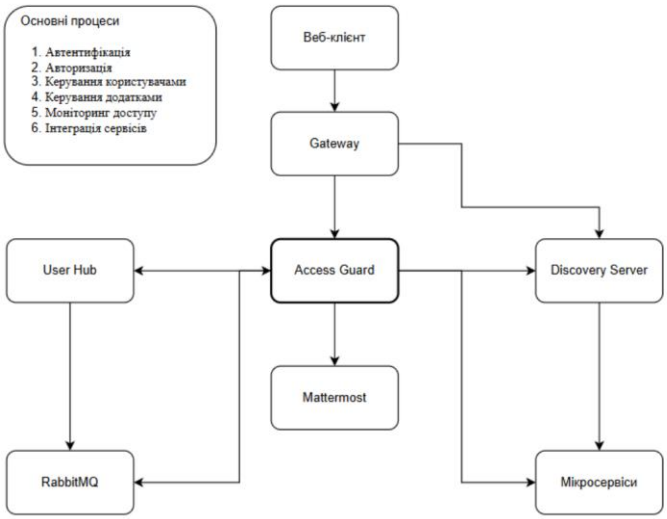
Характеристика	Keycloak	Auth0	IdentityServer	Access Guard
Цільова екосистема	Універсальна	Універсальна	.NET	Мікросервіси на Java/Spring
Використання ресурсів	Високе	Не застосовно	Середнє	Низьке
Інтеграція з Mattermost	Ні	Через додаткові конфігурації	Ні	Так
Реактивний підхід	Ні	Ні	Ні	Так
Зберігання історії входів	Базове	Розширене	Базове	Розширене
Кастомізація авторизації	Середня	Низька	Середня	Висока
Управління пристроями	Частково	Частково	Ні	Так
Інтеграція з корпоративними системами	Через конфігурації	Через API	Через конфігурації	Нативна/Через API
Готовий адмін-інтерфейс	Так	Так	Частково	Так
Вартість	Безкоштовно (self-hosted)	Висока	Безкоштовно (self-hosted)	Внутрішня розробка

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

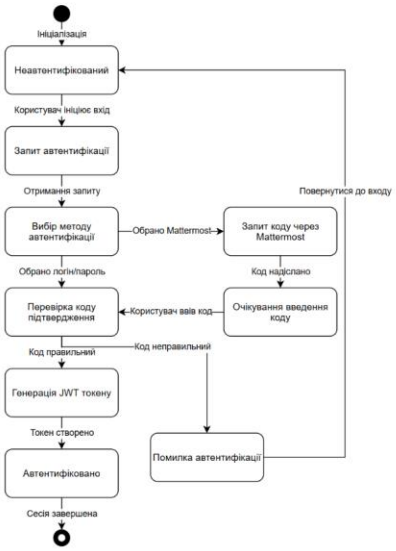
7

Архітектура та взаємодія мікросервісів Access Guard



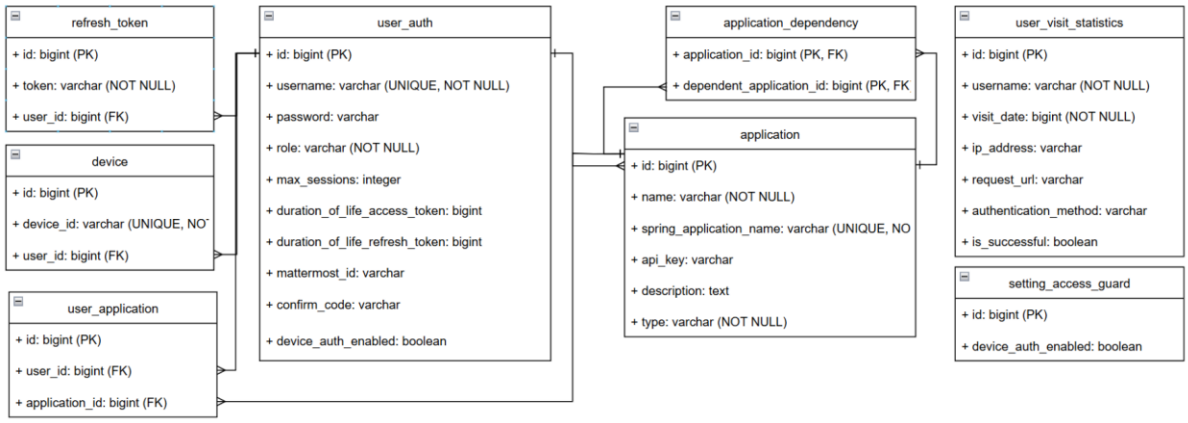
8

Механізми аутентифікації Access Guard



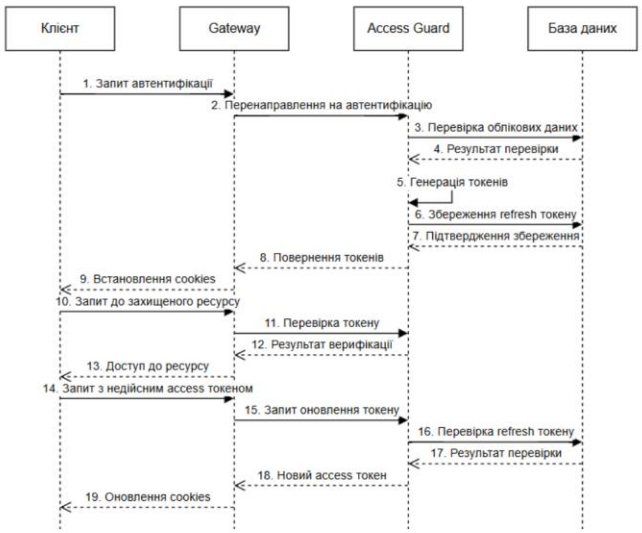
9

Діаграма класів ключових таблиць сервісу Access-guard

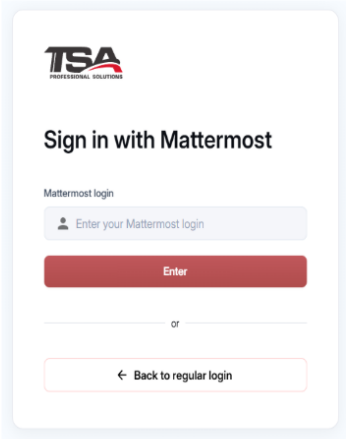


10

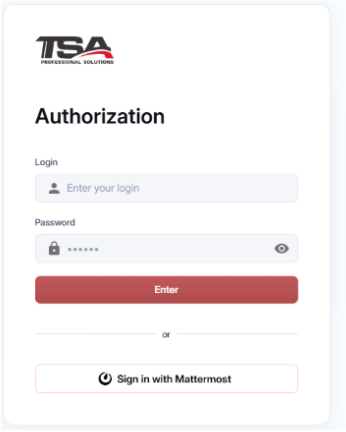
Діаграма послідовності автентифікації



ЕКРАННІ ФОРМИ

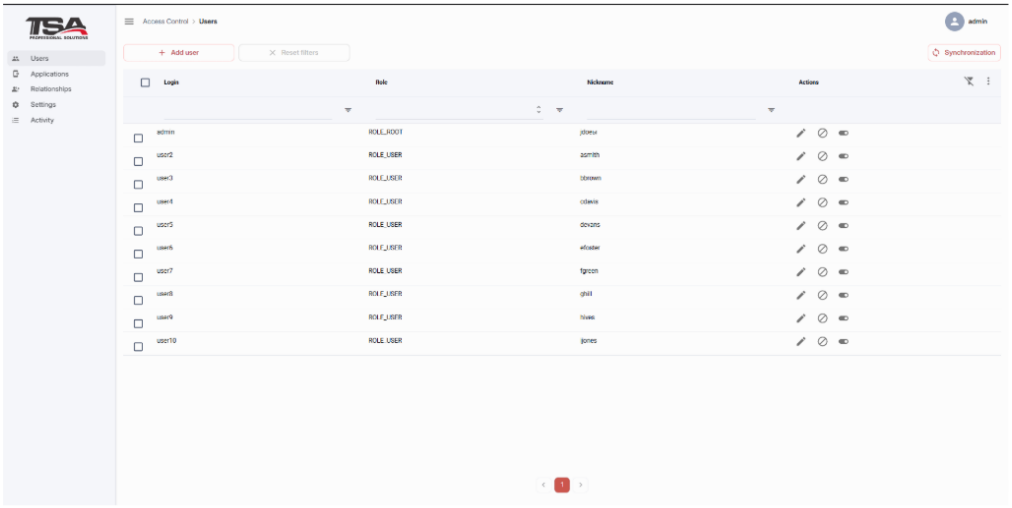


Інтерфейс двофакторної автентифікації



Інтерфейс логін/пароль автентифікації

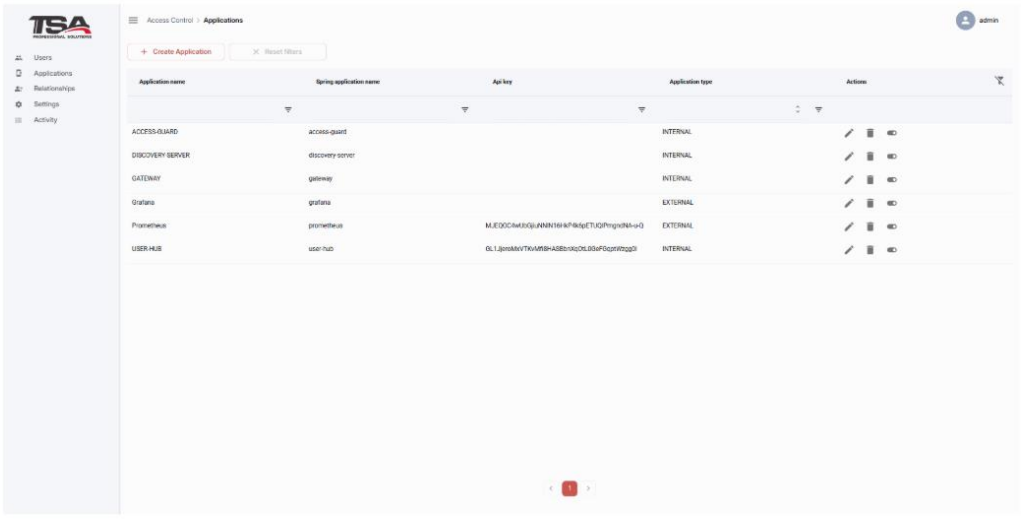
ЕКРАННІ ФОРМИ



Початковий екран із усіма користувачами системи Access-guard

13

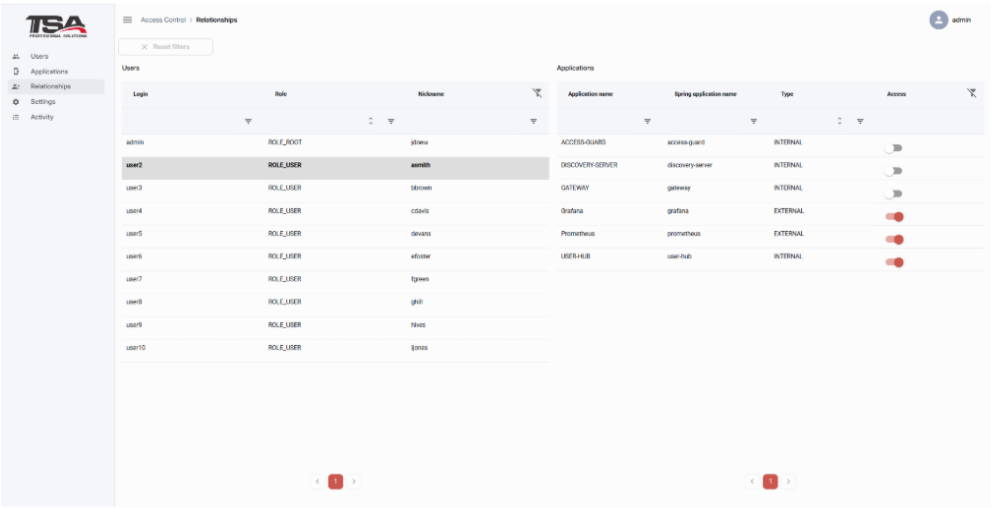
ЕКРАННІ ФОРМИ



Меню відображення зареєстрованих додатків

14

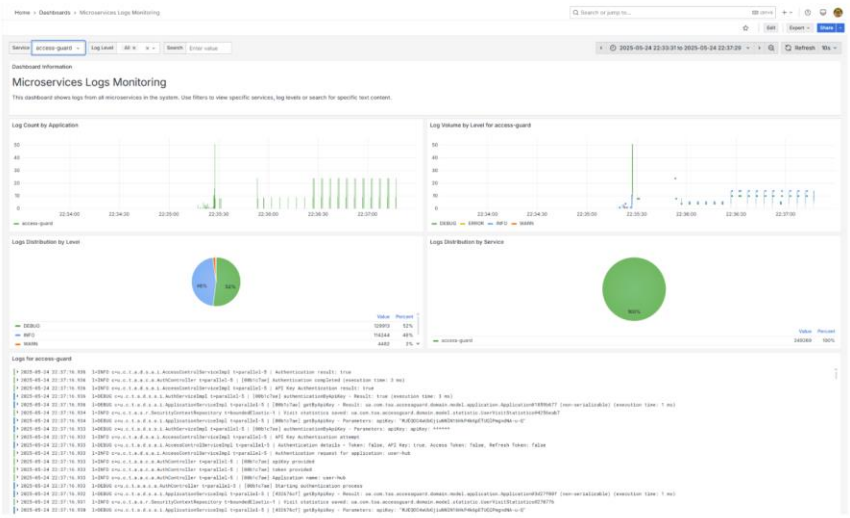
ЕКРАННІ ФОРМИ



Меню для увімкнення/вимкнення доступу до додатків для користувачів

15

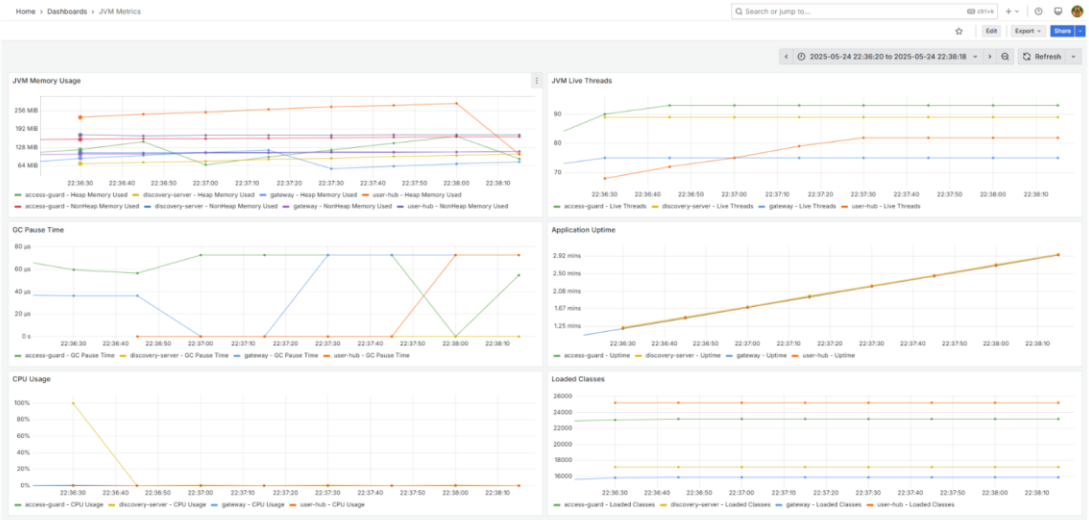
ЕКРАННІ ФОРМИ



Екран аналізу логів access-guard з використанням Loki у Grafana

16

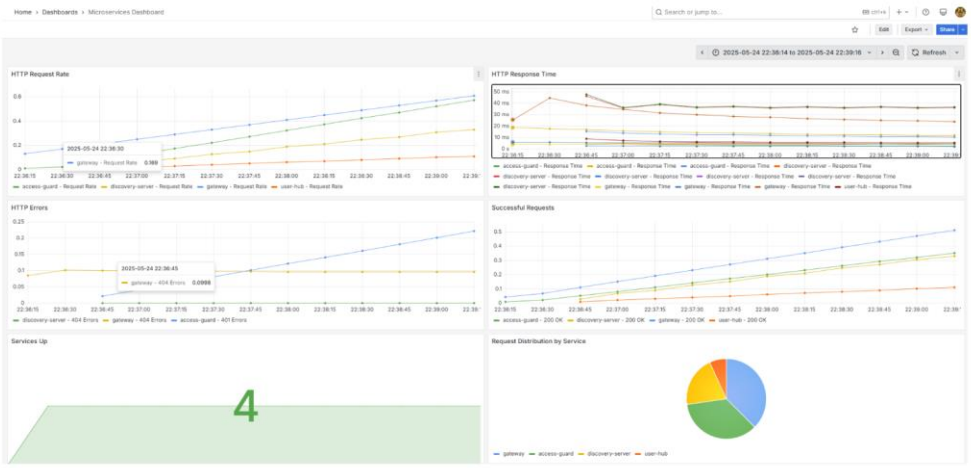
ЕКРАННІ ФОРМИ



Екран моніторингу JVM метрик (пам'ять, потоки, GC, CPU) у Grafana

17

ЕКРАННІ ФОРМИ



Екран моніторингу HTTP-запитів та помилок у Grafana

18

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Олаба Д. Р. Управління ідентифікацією пристроїв як додатковий рівень захисту корпоративних систем. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (ІТ-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 306-308
2. Олаба Д. Р. Безсесійна автентифікація на основі JWT-токенів у мікросервісній архітектурі. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (ІТ-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 304-306
3. Олаба Д. Р. Реактивна архітектура мікросервісів на базі Spring WebFlux для високонавантажених систем. // XII Всеукраїнська науково-практична конференція молодих учених «Інформаційні технології – 2025» (ІТ-2025). – Київ: Київський столичний університет імені Бориса Грінченка, 2025. – С. 166-169
4. Олаба Д. Р., Худік Б.О. Проєктування та реалізація централізованої системи управління доступом у розподілених веб-додатках. // VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». – Київ: ДУІКТ, 2025. – (подано сторінок)
5. Олаба Д. Р., Худік Б.О. Забезпечення безпеки даних у системі захисту Access Guard. // VI Науково-технічна конференція «Сучасний стан та перспективи розвитку IoT». – Київ: ДУІКТ, 2025. – (подано сторінок)

ВИСНОВКИ

1. Розроблено архітектуру мікросервісу автентифікації та авторизації на базі реактивного підходу з використанням Spring WebFlux. Це дозволило досягти обробки 1000 одночасних сесій з латентністю автентифікації до 200 мс та зменшити споживання пам'яті на 65% порівняно з традиційними рішеннями.
2. Реалізовано комплексну систему автентифікації з підтримкою JWT-токенів, двофакторної автентифікації через Mattermost, автентифікації пристроїв та API-ключів. В результаті цього підвищено рівень безпеки системи та забезпечено захист від основних типів кібератак (XSS, CSRF, брутфорс).
3. Впроваджено ієрархічну систему ролей з каскадним управлінням доступом до додатків. Це спростило адміністрування прав користувачів та зменшило час на налаштування доступу нових співробітників з годин до хвилин.
4. Розроблено систему комплексного аудиту з автоматичним відстеженням змін та детальним логуванням. В результаті цього забезпечено повну прозорість операцій системи та можливість швидкого розслідування інцидентів безпеки.
5. Впроваджено систему моніторингу з використанням Prometheus, Grafana, Loki та Zipkin. Це дало можливість оперативно виявляти проблеми в роботі системи та попереджати збої до їх критичного впливу на користувачів.
6. Здійснено контейнеризацію системи з використанням Docker. В результаті цього спрощено процес розгортання та масштабування, забезпечено швидке розгортання системи в різних середовищах.
7. Створено комплексну систему тестування з використанням JUnit 5. Розроблено модульні, що забезпечують 91% покриття коду та гарантують стабільність роботи системи при внесенні змін.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
@LogController
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/v1/users")
@Tag(name = "Users", description = "User management
APIs")
@PreAuthorize("hasRole('ROLE_ADMIN') or
hasRole('ROLE_ROOT') or hasRole('ROLE_SERVICE')")
public class UserController {

    private final UserService userService;
    private final UserMapper userMapper;
    private final UserHubIntegrationService syncService;

    @Operation(summary = "Get all users", description =
"Retrieves a list of all users")
    @ApiResponse(responseCode = "200", description = "List
of users retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @GetMapping("")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<List<UserDTO>>>
getUsers() {
        return Mono.just(new
ResponseDTO<>(userService.getAll().stream().map(userMa
pper::toUserDTO).toList()));
    }


    @Operation(summary = "Sync users", description =
"Synchronizes users with user-hub system")
    @ApiResponse(responseCode = "200", description =
"Users synchronized successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @GetMapping("/sync")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<List<UserDetailDTO>>>
getUsersSync() {


        Mono<List<Map<String, Object>>> listMono =
syncService.fetchUsers();
        return
userOrchestrationService.mergeUsersWithExternalData(user
Service.getAll(), listMono)
                .map(ResponseDTO::new);
    }

    @Operation(summary = "Get user by ID", description =
"Retrieves a user by their ID")
    @ApiResponse(responseCode = "200", description =
"User retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"User not found")
    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<UserDTO>>
getUserById(@PathVariable Long id) {
        return Mono.just(new
ResponseDTO<>(userMapper.toUserDTO(userService.getBy
Id(id))));
    }

    @Operation(summary = "Get user by username",
description = "Retrieves a user by their username")
    @ApiResponse(responseCode = "200", description =
"User retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"User not found")
    @GetMapping("/username")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<UserDTO>>
getUserByUsername(@RequestParam String username) {
        return Mono.just(new
ResponseDTO<>(userMapper.toUserDTO(userService.getBy
Username(username))));
    }

    @GetMapping("/role")
    @ResponseStatus(HttpStatus.OK)
    @Operation(summary = "Get all roles", description =
"Retrieves a list of all available roles")
    @ApiResponse(responseCode = "200", description = "List
of roles retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    public Mono<ResponseDTO<List<Role>>> getRoles() {
        return Mono.just(new
ResponseDTO<>(Arrays.stream(Role.values()).toList()));
    }

    @Operation(summary = "Update user", description =
"Updates an existing user")
    @ApiResponse(responseCode = "200", description =
"User updated successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "400", description =
"Invalid input")
    @ApiResponse(responseCode = "404", description =
"User not found")
    @PutMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    @HideLogSensitiveData
    public Mono<ResponseDTO<UserDTO>>
update(@PathVariable Long id,

@JsonView(UserUpdateDTO.Update.class) @RequestBody
UserUpdateDTO userDTO) {
        userDTO.setId(id);
        return Mono.just(new
ResponseDTO<>(userMapper.toUserDTO(userService.updat
e(id, userMapper.toUser(userDTO)))));
    }

    @Operation(summary = "Delete user", description =
"Deletes a user by their ID")
```

```
    @ApiResponse(responseCode = "204", description =
"User deleted successfully")
    @ApiResponse(responseCode = "404", description =
"User not found")
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> deleteUser(@PathVariable Long id) {
        userService.deleteById(id);
        return Mono.empty();
    }

    private final UserOrchestrationService
userOrchestrationService;

    @GetMapping("/detailed")
    @Operation(summary = "Fetch and synchronize detailed
user information",
            description = "Retrieves user data from both the local
system and the external User Hub service, " +
                    "merging the information into a comprehensive
view. The result includes both local and external data for
each user.")
    @ApiResponse(responseCode = "200", description =
"Successfully retrieved and synchronized user data")
    @ApiResponse(responseCode = "404", description = "No
users found")
    @ApiResponse(responseCode = "500", description =
"Internal server error or failed to fetch external data")
    public Mono<ResponseDTO<List<UserDetailDTO>>>
getAllDetailedUsers() {
        return userOrchestrationService.fetchDetailUsers()
                .map(ResponseDTO::new);
    }

    @GetMapping("/detailed/{id}")
    @ApiResponse(responseCode = "200", description =
"Successfully retrieved and synchronized user data")
    @ApiResponse(responseCode = "404", description = "No
user found")
    @ApiResponse(responseCode = "500", description =
"Internal server error or failed to fetch external data")
    public Mono<ResponseDTO<UserDetailDTO>>
getDetailedUserById(@PathVariable Long id) {
        return Mono.fromCallable(() ->
userService.getById(id))
                .flatMap(user -> {
                    return userService.getUserDetail(user.getId())
                            .map(additionalData -> new
ResponseDTO<>(userMapper.toUserDetailDTO(user)
                                    .setAdditionalData(additionalData)));

@LogController
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/v1/applications")
@Tag(name = "Applications", description = "Application
management APIs")
@PreAuthorize("hasRole('ROLE_ADMIN') or
hasRole('ROLE_ROOT') or hasRole('ROLE_SERVICE')")
public class ApplicationController {

    private final ApplicationService applicationService;
    private final ApplicationMapper applicationMapper;

    @Operation(summary = "Create a new application",
description = "Creates a new application with the given
details")
    @ApiResponse(responseCode = "200", description =
"Application created successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "400", description =
"Invalid input")
    @ApiResponse(responseCode = "403", description =
"Access denied")
    @PostMapping
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<ApplicationDTO>>
createApplication(@RequestBody
@JsonView(ApplicationCreateDTO.Create.class)
ApplicationCreateDTO applicationDTO) {
        return Mono.just(new
ResponseDTO<>(applicationMapper.toDTO(applicationServ
ice.getById(applicationService.saveWithSendToUserHub(ap
plicationMapper.toEntity(applicationDTO)).getId()))));
    }

    @Operation(summary = "Get application by ID",
description = "Retrieves an application by its ID")
    @ApiResponse(responseCode = "200", description =
"Application details retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"Application not found")
    @GetMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<ApplicationDTO>>
getApplicationById(@PathVariable Long id) {
        return Mono.just(new
ResponseDTO<>(applicationMapper.toDTO(applicationServ
ice.getById(id))));
    }

    @Operation(summary = "Get all applications", description
= "Retrieves a list of all applications")
    @ApiResponse(responseCode = "200", description = "List
of applications retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @GetMapping
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<List<ApplicationDTO>>>
getAllApplications() {
        List<ApplicationDTO> applicationDTOs =
applicationService.getAll().stream()
                .map(applicationMapper::toDTO)
                .collect(Collectors.toList());
        return Mono.just(new
ResponseDTO<>(applicationDTOs));
    }

    @Operation(summary = "Update application", description
= "Updates an existing application")
    @ApiResponse(responseCode = "200", description =
"Application updated successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "400", description =
"Invalid input")
    @ApiResponse(responseCode = "404", description =
"Application not found")
    @PutMapping("/{id}")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<ApplicationDTO>>
updateApplication(@PathVariable Long id, @RequestBody
@JsonView(ApplicationCreateDTO.Create.class)
ApplicationCreateDTO applicationDTO) {
```

```
        return Mono.just(new
ResponseDTO<>(applicationMapper.toDTO(applicationServ
ice.update(id,
applicationMapper.toEntity(applicationDTO))));
    }

    @Operation(summary = "Generate API key", description
= "Generates a new API key for the specified application")
    @ApiResponse(responseCode = "200", description = "API
key generated successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"Application not found")
    @PatchMapping("/api-key/generate/{id}")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<ApplicationDTO>>
generateApiKey(@PathVariable Long id) {
        return Mono.just(new
ResponseDTO<>(applicationMapper.toDTO(applicationServ
ice.generateApiKey(id))));
    }

    @Operation(summary = "Get application types",
description = "Retrieves a list of all application types")
    @ApiResponse(responseCode = "200", description = "List
of application types retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @GetMapping("/type")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<List<ApplicationType>>>
getApplicationType() {
        return Mono.just(new
ResponseDTO<>(Arrays.stream(ApplicationType.values()).t
oList()));
    }

    @Operation(summary = "Delete application", description
= "Deletes an application by its ID")
    @ApiResponse(responseCode = "204", description =
"Application deleted successfully")
    @ApiResponse(responseCode = "404", description =
"Application not found")
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> deleteApplication(@PathVariable
Long id) {
        applicationService.deleteById(id);
        return Mono.empty();
    }
}@Slf4j
@LogController
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/v1/auth")
@Tag(name = "Authentication", description =
"Authentication APIs")
public class AuthController {


    private final AuthService authService;
    private final UserMapper userMapper;

    private final UserService userService;
    private final MattermostAuthService
mattermostAuthService;

    private final CookieService cookieService;
    private final AccessControlService accessControlService;


    @HideLogSensitiveData
    @Operation(summary = "Sign in", description =
"Authenticates a user and returns a token")
    @ApiResponse(responseCode = "200", description =
"Authentication successful",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "401", description =
"Authentication failed")
    @PostMapping("/sign-in")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<TokenResponse>> signIn(
            @RequestBody
@JsonView(SignInRequest.PasswordAuth.class)
SignInRequest signInRequest,
            ServerWebExchange exchange) {

        return Mono.fromCallable(() ->
authService.signIn(signInRequest))
                .map(tokenResponse -> {
                    cookieService.setAuthCookies(exchange,
tokenResponse);
                    return new ResponseDTO<>(tokenResponse);
                });
    }

    @Operation(summary = "Log out", description = "Clears
authentication cookies")
    @ApiResponse(responseCode = "204", description =
"Logout successful")
    @PostMapping("/logout")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> logout(ServerWebExchange
exchange) {
        cookieService.logout(exchange);
        return Mono.empty();
    }

    @Operation(summary = "Get Mattermost authentication
code", description = "Generates and sends a Mattermost
authentication code")
    @ApiResponse(responseCode = "204", description =
"Code sent successfully")
    @ApiResponse(responseCode = "400", description =
"Invalid username")
    @PostMapping("/sign-in/mattermost/code")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void> getCode(@Parameter(description =
"username") @RequestParam String username) {

mattermostAuthService.generateAndSendCode(username);
        return Mono.empty();
    }

    @HideLogSensitiveData
    @Operation(summary = "Confirm Mattermost code",
description = "Confirms the Mattermost authentication
code")
    @ApiResponse(responseCode = "200", description =
"Code confirmed successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "401", description =
"Invalid code")
    @PostMapping("/sign-in/mattermost/confirm")
```

```
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<TokenResponse>>
confirmCode(@RequestBody
@JsonView(SignInRequest.MattermostAuth.class)
SignInRequest signInRequest, ServerWebExchange
exchange) {

        return Mono.fromCallable(() ->
mattermostAuthService.confirmCode(signInRequest))
                .map(tokenResponse -> {
                    cookieService.setAuthCookies(exchange,
tokenResponse);
                    return new ResponseDTO<>(tokenResponse);
                });
    }

    @Operation(summary = "Get Mattermost authentication
code", description = "Generates and sends a Mattermost
authentication code")
    @ApiResponse(responseCode = "204", description =
"Code sent successfully")
    @ApiResponse(responseCode = "400", description =
"Invalid device ID")
    @PostMapping("/sign-in/device/mattermost/code")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void>
getCodeByDevice(@Parameter(description = "Device ID")
@RequestParam String deviceId) {

mattermostAuthService.generateAndByDeviceSendCode(dev
iceId);
        return Mono.empty();
    }

    @HideLogSensitiveData
    @Operation(summary = "Confirm Mattermost code",
description = "Confirms the Mattermost authentication
code")
    @ApiResponse(responseCode = "200", description =
"Code confirmed successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "401", description =
"Invalid code")
    @PostMapping("/sign-in/device/mattermost/confirm")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<TokenResponse>>
confirmCodeByDevice(@RequestBody
SignInDeviceRequest signInRequest,
                                            ServerWebExchange
exchange
    ) {
        return Mono.fromCallable(() ->
mattermostAuthService.confirmCodeByDevice(signInReque
st))
                .map(tokenResponse -> {
                    cookieService.setAuthCookies(exchange,
tokenResponse);
                    return new ResponseDTO<>(tokenResponse);
                });
    }



    @HideLogSensitiveData
    @Operation(summary = "Sign up", description =
"Registers a new user")
    @ApiResponse(responseCode = "200", description =
"User registered successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "400", description =
"Invalid input")
    @PostMapping("/sign-up")
    @ResponseStatus(HttpStatus.OK)
    @PreAuthorize("hasRole('ROLE_ADMIN') or
hasRole('ROLE_ROOT')")
    public Mono<ResponseDTO<SignUpRequest>>
signUp(@RequestBody
@JsonView(SignUpRequest.Create.class) SignUpRequest
signUpRequest) {
        return Mono.just(new
ResponseDTO<>(userMapper.toSignUpRequest(authService.
signUp(userMapper.toUser(signUpRequest)))));
    }

    @Operation(summary = "Refresh token", description =
"Refreshes an access token")
    @ApiResponse(responseCode = "200", description =
"Token refreshed successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "401", description =
"Invalid refresh token")
    @HideLogSensitiveData
    @PostMapping("/token")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<TokenResponse>>
refreshToken(@RequestBody
@JsonView(TokenResponse.UpdateByRefresh.class)
                                            TokenResponse
accessToken) {
        return Mono.just(new
ResponseDTO<>(authService.refreshToken(accessToken)));
    }



    @GetMapping("")
    @ResponseStatus(HttpStatus.OK)
    @PreAuthorize("isAuthenticated()")
    @LogGatewayAuth
    public Mono<ResponseDTO<Boolean>> auth(
            @RequestParam("application-name") String
applicationName,
            @RequestParam(value = "url-info", required = false)
String url,
            @RequestHeader(value = "Authorization", required =
false) String token,
            @RequestHeader(value = "X-API-KEY", required =
false) String apiKey,
            @CookieValue(value = "accessToken", required =
false) String accessToken,
            @CookieValue(value = "refreshToken", required =
false) String refreshToken,
            ServerWebExchange exchange,
            Principal principal) {

        return accessControlService.authenticate(
                applicationName,
                token,
                apiKey,
                accessToken,
                refreshToken,
                exchange
        ).map(ResponseDTO::new);
    }@RestController
@RequestMapping("/api/v1/devices")
@RequiredArgsConstructor
```

```
@Tag(name = "Devices", description = "Device management
APIs")
public class DeviceController {
    private final DeviceService deviceService;
    private final DeviceMapper deviceMapper;

    @PostMapping("/register")
    public Mono<ResponseDTO<DeviceDTO>>
registerDevice(@RequestBody DeviceDTO deviceDTO,
Principal principal) {
        Device device = deviceMapper.toEntity(deviceDTO);
        Device registeredDevice =
deviceService.registerDevice(device, principal.getName());
        return Mono.just(new ResponseDTO<>(
deviceMapper.toDTO(registeredDevice)));
    }

    @DeleteMapping("/{deviceId}")
    public ResponseEntity<Void>
deleteDevice(@PathVariable String deviceId, Principal
principal) {
        deviceService.deleteDevice(deviceId,
principal.getName());
        return ResponseEntity.noContent().build();
    }
}
@LogController
@RestController
@RequestMapping("/api/v1/user-applications")
@RequiredArgsConstructor
@Tag(name = "User Applications", description = "User-
Application association APIs")
public class UserApplicationController {

    private final UserApplicationService
userApplicationService;

    private final UserMapper userMapper;
    private final ApplicationMapper applicationMapper;


    @Operation(summary = "Add user application",
description = "Associates a user with an application")
    @ApiResponse(responseCode = "200", description =
"User application added successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "400", description =
"Invalid input")
    @PostMapping
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<UserApplication>>
addUserApplication(@RequestBody
UserApplicationRequest request) {
        UserApplication userApplication =
userApplicationService.save(request.getUserId(),
request.getApplicationId());
        return Mono.just(new
ResponseDTO<>(userApplication));
    }

    @Operation(summary = "Remove user application",
description = "Removes the association between a user and
an application")
    @ApiResponse(responseCode = "204", description =
"User application removed successfully")
    @ApiResponse(responseCode = "404", description =
"User application not found")
    @DeleteMapping
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public Mono<Void>
removeUserApplication(@RequestBody
UserApplicationRequest request) {
        userApplicationService.remove(request.getUserId(),
request.getApplicationId());
        return Mono.empty();
    }

    @GetMapping("/user/{userId}/applications")
    @Operation(summary = "Get all applications for a user",
description = "Retrieves all applications associated with a
specific user")
    @ApiResponse(responseCode = "200", description = "List
of applications retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"User not found")
    @ResponseStatus(HttpStatus.OK)
    public Mono<ResponseDTO<List<ApplicationDTO>>>
getAllApplicationsByUserId(@PathVariable Long userId) {
        return Mono.just(new
ResponseDTO<>(userApplicationService.getAllApplications
ByUserId(userId).stream().map(applicationMapper::toDTO).t
oList()));
    }

    @Operation(summary = "Get all users for an application",
description = "Retrieves all users associated with a specific
application")
    @ApiResponse(responseCode = "200", description = "List
of users retrieved successfully",
            content = @Content(schema =
@Schema(implementation = ResponseDTO.class)))
    @ApiResponse(responseCode = "404", description =
"Application not found")
```