Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)
Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Веб застосунок для обліку та планування замовлень по догляду за садом та газоном. Бек-енд
(тема)

Виконав:
здобувач 4 року навчання
групи ПЗПІ-21-7

Антон ПЄХОТІН
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. кафедри ПІ Андрій БАБІЙ
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук форми навчання
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Пєхотіну Антону Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Веб застосунок для обліку та планування замовлень по догляду за садом та газоном. Бек-енд
Затверджена наказом по університету від 19.05.2025 №397 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 13.06.2025
3. Вихідні дані до роботи Розробити бекенд частину веб-застосунку для обліку, планування та контролю виконання замовлень з догляду за садом і газоном.
4. Перелік питань, що потрібно опрацювати в роботі
Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, впровадження програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05	*виконано*
2	Створення специфікації ПЗ	20.05	*виконано*
3	Проектування ПЗ	21.05	*виконано*
4	Розробка ПЗ	31.05	*виконано*
5	Тестування ПЗ	01.06	*виконано*
6	Оформлення пояснювальної записки	04.06	*виконано*
7	Підготовка презентації та доповіді	05.06	*виконано*
8	Попередній захист	10.06	*виконано*
9	Нормоконтроль, рецензування	09.06	*виконано*
10	Здача роботи у електронний архів	09.06	*виконано*
11	Допуск до захисту у зав. кафедри	13.06	*виконано*

Дата видачі завдання « 19 » « травня » 2025р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. кафедри ПІ Андрій БАБІЙ
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра: 71 с., 15 рис., 7 додатків, 17 джерел.

API, БАЗА ДАНИХ, ВИКОНАВЦІ, ГАЗОН, ДОГЛЯД ЗА САДОМ, ДОКУМЕНТООБІГ, ЕЛЕКТРОННА ПОШТА, GEOLOCATION, INVOICES, КЕШУВАННЯ, КЛІЄНТИ, MICROSERVICES, MONGODB, PAYPAL, ПЛАНУВАННЯ, POSTGRES, REDIS, REST API, СЕЗОННІСТЬ, СИСТЕМА, SPRING BOOT, ХМАРНІ СЕРВІСИ, ЗАМОВЛЕННЯ

Об’єктом дослідження є сфера автоматизації послуг з догляду за приватними садовими ділянками та газонами. Актуальність теми обумовлена зростанням попиту на сервісні платформи для планування та організації робіт з обслуговування територій, зокрема в умовах зростаючої урбанізації, підвищення цінності естетичного вигляду прибудинкових територій та зростання кількості клієнтів, які надають перевагу цифровим сервісам самообслуговування.

Метою дипломної роботи є розробка серверної частини веб-застосунку, який забезпечує облік замовлень клієнтів, формування індивідуальних планів догляду, призначення працівників для їх виконання. Система має враховувати сезонність, наявність обладнання, пріоритетність завдань, а також географічне розташування замовлень для оптимального розподілу ресурсів.

Система включає механізми формування рахунків-фактур та актів виконаних робіт після завершення обслуговування. Документи генеруються автоматично на основі даних про виконані завдання та можуть бути надіслані клієнтам за допомогою електронної пошти.

Для обробки онлайн-оплат користувачів реалізовано інтеграцію з платіжною платформою PayPal. Система підтримує ініціацію платежів з особистого кабінету клієнта та зберігання відповідних транзакцій у базі даних.

Методи розробки базуються на використанні мікросервісної архітектури, реалізованої у середовищі Spring. Як бази даних використовуються PostgreSQL

для зберігання структурованих бізнес-даних, Redis для реалізації кешування та збереження геолокаційних координат, а також MongoDB — для зберігання неструктурованої інформації, зокрема про обладнання, його використання та технічний стан.

У результаті роботи реалізовано функціональний бекенд для системи замовлень з догляду за садами та газонами, що дозволяє інтегрувати веб-інтерфейс для клієнтів та працівників через REST API. Система є масштабованою, розширюваною та готовою до подальшої інтеграції з мобільним застосунком.

ABSTRACT

API, DATABASE, CONTRACTORS, LAWN, GARDEN CARE, DOCUMENT WORKFLOW, EMAIL, GEOLOCATION, INVOICES, CACHING, CLIENTS, MICROSERVICES, MONGODB, PAYPAL, PLANNING, POSTGRES, REDIS, REST API, SEASONALITY, SYSTEM, SPRING BOOT, CLOUD SERVICES, ORDERS

The subject of the research is the field of automation of services for the maintenance of private garden plots and lawns. The relevance of the topic is determined by the growing demand for service platforms for planning and organizing grounds maintenance work, in particular under conditions of increasing urbanization, the rising value of the aesthetic appearance of residential grounds, and the growing number of clients who prefer digital self-service platforms.

The goal of the thesis is to develop the server side of a web application that ensures accounting for client orders, the formation of individual maintenance plans, and the assignment of workers to execute them. The system must take into account seasonality, the availability of equipment, task priority, as well as the geographical location of orders for optimal resource allocation.

The system includes mechanisms for generating invoices and acts of completed work upon completion of maintenance. Documents are generated automatically based on data about completed tasks and can be sent to clients via email.

To process users' online payments, integration with the PayPal payment platform has been implemented. The system supports initiating payments from the client's personal account and storing the corresponding transactions in the database.

The development methods are based on using a microservices architecture implemented in the Spring environment. PostgreSQL is used as a database for storing structured business data, Redis for implementing caching and storing geolocation coordinates, and MongoDB for storing unstructured information, about equipment, its usage, and technical condition.

As a result of the work, a functional backend for the garden and lawn care order system has been implemented, allowing integration of a web interface for clients and workers via REST API. The system is scalable, extensible, and ready for further integration with a mobile application.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface

CI/CD – Continuous Integration / Continuous Delivery

CRUD – Create, Read, Update, Delete

DTO – Data Transfer Object

HTTP – Hypertext Transfer Protocol

JWT – JSON Web Token

OAuth2 – Open Authorization 2.0

POD – Pod (Kubernetes workload unit)

REST – Representational State Transfer

S3 – Simple Storage Service

SDK – Software Development Kit

UI – User Interface

URL – Uniform Resource Locator

UUID – Universally Unique Identifier

YAML – YAML Ain’t Markup Language

ВСТУП

Метою роботи є створення веб-застосунку для обліку, планування та контролю виконання замовлень з догляду за садом і газоном. Тема кваліфікаційної роботи: «Веб застосунок для обліку та планування замовлень по догляду за садом та газоном. Бекенд». Розроблена система орієнтована на дві категорії користувачів: клієнтів, які можуть оформлювати замовлення на послуги, та працівників, які виконують завдання та звітують про результати роботи.

Основне призначення програми полягає у створенні зручного інструменту для організації сервісу із догляду за приватними зеленими територіями. Користувачі мають можливість реєструвати замовлення, формувати індивідуальні плани робіт, переглядати статуси виконання та отримувати фінансові документи. Для працівників реалізована можливість перегляду доступних завдань, прийому в роботу, звітування про виконання, а також доступ до оренди необхідного обладнання.

Серверна частина системи реалізована мовою програмування Java з використанням фреймворку Spring Boot. Архітектура системи побудована на основі мікросервісного підходу. У якості СУБД використовується PostgreSQL для основних структурованих даних, Redis - для кешування та швидкого доступу до геолокаційної інформації, MongoDB - для зберігання неструктурованих даних, зокрема описів та характеристик обладнання.

Необхідна для реалізації проекту інформація була отримана з офіційної документації Spring Boot, PostgreSQL, Redis, MongoDB, React, а також з тематичних ресурсів і спільнот, зокрема Stack Overflow та офіційних блогів розробників.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Сучасний стан галузі послуг із догляду за садом і газоном

Сфера обслуговування приватних ділянок, зелених зон та садово-паркових територій стрімко розвивається в усіх регіонах світу. Згідно з останніми дослідженнями, глобальний ринок ландшафтних та озеленювальних послуг у 2024 році перевищив $330 млрд і прогнозовано зростатиме як мінімум на 6 % щороку до 2030-го. Основні фактори росту галузі:

а) Урбанізація та забудова заміських територій. Багато мешканців великих міст виїжджають у передмістя, де догляд за ділянкою потребує спеціалізованих навичок та обладнання.

б) Підвищення рівня життя. Зі збільшенням доходів люди частіше готові делегувати роботу з догляду за газоном та садом професіоналам.

в) Фокус на екологію. Замовники прагнуть екологічно безпечних рішень: мінімізація пестицидів, економне використання води, використання місцевих сортів рослин.

В Україні галузь тільки набирає обертів: за оцінками ринку, обсяг становить лише 5–7 % власників приватних домоволодінь, інші обробляють газони самостійно або ж обходяться базовим доглядом. Проте за останні 3 роки кількість компаній із професійним обслуговуванням зросла майже вдвічі, а загальна вартість ринку перевищила 800 млн грн у 2024 році.

До найбільш поширених послуг належать:

а) сезонний догляд: комплекс весняних та осінніх робіт (чистка, мульчування, добриво);

б) стрижка газону: регулярне підрізання трав’яного покриву за різною висотою, щоб забезпечити його здоровий вигляд;

в) обрізка дерев та кущів: санітарна та формувальна обрізка, що підвищує декоративні властивості і безпеку на ділянці;

г) боротьба зі шкідниками та хворобами: профілактичні обробки від шкідників та грибків (застосування біопрепаратів, мінімізація хімікатів);

д) установка та обслуговування систем поливу: від простих спринклерних систем до повністю автоматизованих Smart-систем, які працюють за погодними даними;

е) консультації з ландшафтного дизайну: розробка плану висаджування, вибір рослин відповідно до ґрунтово-кліматичних умов.

Важливість впровадження ІТ-рішень у цю галузь зумовлена необхідністю:

а) автоматизації обліку замовлень;

б) оптимізації розкладу працівників та обладнання;

в) контролю якості виконання робіт;

г) прозорої фінансової звітності (рахунки, акти).

1.1.1 Тенденції цифровізації

За даними опитування серед компаній-надавачів послуг, 65 % вже використовують хоча б базову CRM-систему для відстеження клієнтів та замовлень, але лише 20 % мають інтегровані мобільні або веб-додатки.

Попит на рішення із геолокацією зріс через зростаючий попит працівників на можливість вибирати завдання поблизу їхнього місцезнаходження; багато фахівців вважають це ключовою функцією.

1.2 Аналіз основних процесів та учасників

Для побудови ефективного програмного забезпечення потрібно чітко зрозуміти бізнес-процеси, що супроводжують цикли замовлення, виконання і звітності.

1.2.1 Клієнти

Клієнтська частина системи орієнтована на власників приватних будинків, дач, котеджів, або ж на уповноважених представників ОСББ (житлових комплексів).

Основні потреби та сценарії:

а) оформлення замовлення;

1) опис місця роботи (адреса/координати);

2) вибір послуги (наприклад, «стрижка газону»);

3) вказівка побажань (висота скошування, дата/час);

4) опція вибору разового або регулярного обслуговування (щотижня, щомісяця).

б) формування плану догляду;

1) вибір послуг у залежності від пори року (пропозиції весняної обробки, осінньої підготовки);

2) можливість переглянути та відредагувати автоматично згенерований графік.

в) контроль статусу замовлення;

1) перегляд, чи призначено працівника;

2) отримання сповіщень про зміну статусу (через email / SMS).

г) оплата та звітність;

1) перегляд рахунку-фактури;

2) оплата онлайн;

3) перегляд актів виконаних робіт.

д) зворотний зв’язок: можливість залишити відгук, поставити рейтинг;

1.2.2 Працівники та бригади

Працівники — це фахівці з догляду за садом: садівники, фахівці з обробки рослин.

Основні сценарії:

— реєстрація профілю;

— пошук відкритих замовлень: працівник заходить на сторінку «Відкриті замовлення», бачить карту відкритих замовлень неподалік. Використовуються поточні координати працівника;

— прийняття замовлення: натискає «Прийняти», система відправляє призначає працівника цьому замовленню та видаляє замовлення з відкритих;

— виконання роботи: приїжджає на місце, виконує роботи, позначає «Виконано», замовлення закривається, клієнту виставляється рахунок, створюється акт виконаних робіт та відправляються клієнту електронною поштою.

1.3 Виявлення та вирішення проблем

1.3.1 Відсутність централізованого обліку

У багатьох дрібних компаніях і приватних майстрів усі процеси фіксуються вручну:

— записи в блокнотах або Excel;

— немає єдиної бази даних замовлень;

— відсутність системи повідомлень при зміні статусу.

В результаті:

— дублювання даних, плутанина в графіках;

— втрата клієнтів через затримки;

— неможливість проводити аналітику.

Рішення:

— єдина БД;

— сервіс Order Service для збереження замовлень;

— черга повідомлень для гарантування доставки подій між сервісами.

1.3.2 Неефективна комунікація між учасниками

Проблеми:

— працівники не отримують оновлення в реальному часі (наприклад, замовлення скасовано, а в їхньому списку ще залишається).

— клієнти не знають, у який час приїде працівник;

— Рішення: Notification Service надсилає email / SMS.

1.4 Системи-конкуренти та їхні особливості

На ринку існують низка іноземних SaaS-рішень, адаптованих під управління послугами у сфері догляду за територіями:

1.4.1 Jobber

Плюси:

— комплексна CRM + мобільний застосунок;

— інтеграція з QuickBooks та Stripe;

— автоматичні сповіщення клієнтів про прибуття працівника.

Мінуси:

— висока ціна абонемента ($150/міс.);

— відсутня українська локалізація;

— обмежена підтримка геолокаційних фіч.

1.4.2 LawnPro

Плюси:

— спеціалізований модуль планування стрижки газонів;

— інтеграція з Google Maps для відстеження маршрутів працівників;

— гнучка система підписки (поштучна оплата).

Мінуси:

— вимагає високої кваліфікації адміністраторів;

— функціонал пошуку замовлень за координатами обмежений;

— немає локальної підтримки (лише англійською).

1.4.3 Service Autopilot

Плюси:

— пропонує готові шаблони договорів, рахунків;

— вбудований модуль SMS-розсилки;

— доступний API для інтеграції з іншими системами.

Мінуси:

— високі комісії за транзакції;

— обмежена функція автоматичного створення завдань (тільки через календар).

1.5 Цільова аудиторія

Програмна система розрахована на такі сегменти користувачів:

Власники приватних будинків, котеджів, дач:

— потребують простий спосіб замовити послуги без дзвінків і Excel;

— хочуть отримати точний графік приїзду працівників;

— важлива опція «регулярного обслуговування» та автоматичні сповіщення.

Працівники садово-паркового сервісу з вільним графіком:

— потребують можливості швидко знаходити замовлення в межах своєї зони;

— хочуть уникнути «порожніх» переїздів;

— потрібен інструмент для роботи.

— компанії з озеленення великих територій (ОСББ, ТОВ):

— потребують комплексного управління замовленнями в різних районах;

— важлива аналітика: кількість виконаних завдань, середній час, затрати на обладнання.

1.6 Висновки щодо предметної галузі

Проведений аналіз показав, що галузь послуг із догляду за територіями активно трансформується в бік цифровізації та автоматизації.

— відсутність централізованих IT-рішень створює неефективність: дублювання ручних записів, втрати даних і помилки в розкладах;

— запити на геолокаційний пошук стають дедалі актуальнішими, оскільки працівники прагнуть мінімізувати час у дорозі та максимізувати кількість виконаних завдань;

— фінансова прозорість й електронний документообіг є ключовими факторами для більшості власників бізнесів у галузі;

— існуючі SaaS-рішення не цілком відповідають локальним потребам (відсутність української локалізації, інтеграції з національними платіжними системами, висока вартість підписок).

Отже, є чітка потреба в адаптованій веб-системі, що:

— забезпечуватиме геолокаційні-запити;

— інтегруватиме популярні методи оплати;

— надаватиме послугу з оренди обладнання.

Розробка такої системи підвищить ефективність роботи компаній, зменшить час простою працівників та створить прозорі фінансові процеси.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Загальні вимоги

Бекенд програмної системи для сервісу з догляду за садами та газонами являє собою набір мікросервісів, які забезпечують логіку обробки запитів, взаємодію з базами даних, автентифікацію користувачів, керування замовленнями, планування робіт, облік ресурсів та формування фінансової документації. Кожен мікросервіс відповідає за окрему частину функціоналу системи та має бути незалежним у своєму розгортанні, масштабуванні та обслуговуванні.

Система має бути реалізована з використанням Java 21 і Spring Boot як основного фреймворку для розробки мікросервісів. Кожен сервіс повинен мати власну базу даних, що гарантує незалежність збереження даних та знижує ризик конфліктів.

Комунікація між сервісами відбувається через REST API або асинхронно за допомогою черги повідомлень RabbitMQ. Це дозволяє реалізувати подієво-орієнтовану модель обробки даних та забезпечити відмовостійкість при високих навантаженнях. Окрім цього, передбачається використання API Gateway для маршрутизації запитів та централізованої обробки безпеки.

Передбачається також реалізація модуля фільтрації замовлень за відстанню та містом.

2.2 Технічні вимоги

Мова програмування: Java 21.

Фреймворк: Spring Boot 3, Spring Security, Spring Cloud;.

Бази даних: PostgreSQL для основних сервісів (з підтримкою реплікації для читання), Redis - для кешування та зберігання тимчасових даних.

Комунікація: REST + RabbitMQ (для подій створення/завершення замовлень, змін статусу, повідомлень);.

Авторизація та автентифікація: OAuth2 + JWT.

Розгортання: Docker-контейнери, оркестрація — Kubernetes (або Docker Compose для локального середовища розробки), хмарний сервіс розгортання - Digital Ocean.

CI/CD: GitHub Actions з автоматичним тестуванням, збіркою та розгортанням.

2.3 Нефункціональні вимоги

Надійність: система повинна бути відмовостійкою, з автоматичним перезапуском або масштабуванням сервісів у разі збоїв. Повинна реалізовуватися стратегія резервного копіювання критичних даних.

Безпека: усі запити до захищених ресурсів повинні автентифікуватися за допомогою JWT токенів. Доступ до ресурсів має бути обмежений відповідно до ролей користувачів (адміністратор, менеджер, працівник, клієнт). Всі API-запити мають бути захищені протоколом HTTPS.

Масштабованість: архітектура має підтримувати горизонтальне масштабування за рахунок додавання нових мікросервісів та баз даних для забезпечення роботи під високим навантаженням.

Розширюваність: нові сервіси (наприклад, інтеграція з банківськими сервісами, CRM або email/SMS-платформами) повинні легко додаватися до системи без змін у поточному коді. Передбачено шаблони для швидкої генерації нових мікросервісів.

Продуктивність: система має підтримувати обробку не менше ніж 100 запитів/секунду з часом відповіді до 500 мс у межах нормального навантаження. При піковому навантаженні допустиме тимчасове зростання часу відповіді, але не більше ніж 1.5 секунди.

Зручність користування: REST API має бути логічно структурованим, з чіткою документацією. Очікується низький поріг входження для нових розробників.

Підтримка тестування: код повинен мати покриття юніт- та інтеграційними тестами. Повинна існувати можливість ізольованого тестування кожного сервісу (mock середовище).

Вищезазначені вимоги спрямовані на забезпечення надійної, безпечної та ефективної системи, що відповідає очікуванням користувачів і масштабам сучасних веб сервісів.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проєктування ПЗ

Перед початком розробки були визначені основні функції та актори системи. Після детального аналізу була створена Use-case діаграма (див. рис. 4.1).

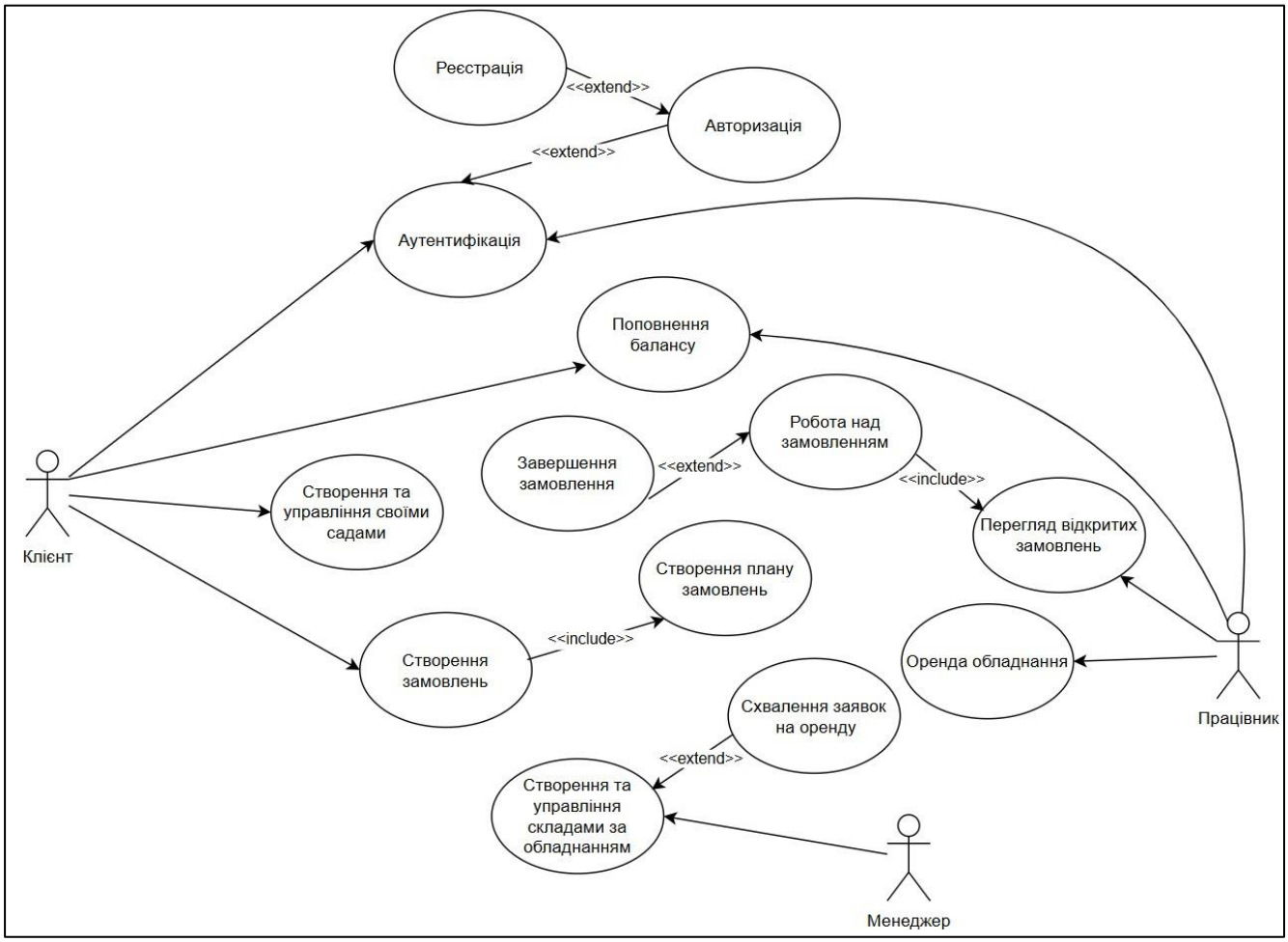

Рис 4.1 – Use-case діаграма

Відповідно визначено основних користувачів системи

а) клієнт, який має функції аутентифікації, створення та управління своїми садами, створення замовлень для своїх садів, поповнення балансу;

б) працівник, який має функції аутентифікації, перегляду доступних замовлень, робота над ними, оренди обладнання та поповнення балансу;

в) менеджер складу, який має функції створення та управління складами з обладнанням, та управління самим обладнанням, схвалення заявок на оренду та повернення обладнання.

3.2 Проєктування архітектури ПЗ

Проєктування архітектури системи розпочиналося з моделювання бізнес-процесів, визначення основних користувачів (акторів) та типових сценаріїв їхньої взаємодії із системою. Предметна область включає такі ключові напрями: замовлення послуг, виконання робіт працівниками, облік обладнання, фінансові операції, а також обробка геолокаційної інформації.

Після цього було обрано мікросервісну архітектуру як основний стиль, що дозволяє розділити кожну бізнес-функцію на окремий сервіс. Для проєктування кожного мікросервісу були використані принципи Domain-Driven Design (DDD):

– кожен сервіс відповідає за свою предметну підмодель;

– об'єкти не повторюються між сервісами, між ними діє чітка відповідальність та API-контракти.

Між сервісами встановлені обмежені типи взаємодії:

– синхронна (REST) — використовується лише для читання даних або рідкісних змін;

– асинхронна (черги подій через RabbitMQ) — використовується для сповіщення про події, які можуть зацікавити інші сервіси (створення замовлення, зміна статусу тощо), та передачі важливих даних та подій.

Ключовими принципами проєктування архітектури були:

— слабке зв’язування: сервіси не залежать від внутрішньої реалізації один одного;

— інкапсуляція даних: кожен сервіс має власну базу даних і не має прямого доступу до баз інших сервісів;

— висока доступність: кожен компонент повинен бути доступним у кількох екземплярах з автоматичним балансуванням навантаження;

— безпечність: всі запити перевіряються через єдиний шлюз і обробляються відповідно до ролей користувачів;

— розширюваність: можливість додавання нових сервісів без зміни існуючих.

У процесі проєктування враховувалися також нефункціональні вимоги: висока продуктивність, масштабованість, простота супроводу, тощо.

3.2.1 Основні мікросервіси системи

— ApiGateway шлюз для маршрутизації запитів до відповідних сервісів. Використовує Spring Cloud Gateway. Реалізує кросс-сервісну автентифікацію (JWT), CORS, логування;

— Auth Service відповідає за реєстрацію, автентифікацію та авторизацію користувачів. Генерує JWT токени;

— Client Service обробляє дані клієнтів, зберігає інформацію про профілі, замовлення та історію користування послугами. Підтримує рейтингову систему клієнтів та механізм блокування в разі зловживання послугами;

— Worker Service керує працівниками, зокрема їх статусами (доступний/зайнятий), історією виконаних замовлень, тощо;

— Garden Service керує створенням та управлінням садами клієнтів. Підтримує базові CRUD операції, також реалізовано збереження картинок садів в хмарному середовищі Amazon S3, та збереження їх URL в базі даних Postgres;

— Order Service відповідає за створення, пошук та оновлення замовлень. Створює замовлення на основі запрошених клієнтом задачам. Реалізована логіка різних статусів замовлення: створено, прийнято, в роботі, завершено, скасовано тощо. Під’єднаний до черги повідомлень RabbitMQ, для асинхронної комунікації з іншими сервісами, зокрема Open Order;

— Open Order Service для геолокаційного пошуку відкритих замовлень. Отримує повідомлення про відкриття замовлення з RabbitMQ та зберігає їх у Redis з геопросторовим індексом для швидкого пошуку за координатами;

— Billing Service реалізовує логіку грошового балансу клієнтів та працівників, який змінюється відповідно до їх дій (оплата клієнтом виконаного замовлення або оренда обладнання працівником). Підключений до черги RabbitMQ та очікує на повідомлення про зміну балансу з інших сервісів. Для поповнення балансу сервіс підключений до міжнародної платіжної системи

PayPal, яка обробляє платежі користувачів та змінює баланс за умови успішної сплати;

— Equipment Service відповідає за облік і видачу обладнання. Реєструє, яке обладнання використовується, ким та коли було повернено. Дозволяє менеджерам бачити вільне/зайняте обладнання в реальному часі;

— Notification service асинхронно надсилає повідомлення на електрону адресу користувача, при зміні статусу замовлення, підтвердженні виконання робіт тощо. Використовує RabbitMQ для отримання повідомлень від інших сервісів.

Для кращого розуміння архітектури було створено діаграму компонентів системи (див. додаток В).

3.3 Проєктування структури зберігання даних

Структура зберігання даних у системі побудована відповідно до обраної мікросервісної архітектури. Кожен сервіс має власну незалежну базу даних, яка реалізує принцип "Database per service". Такий підхід забезпечує інкапсуляцію даних, підвищує безпеку та дозволяє кожному сервісу масштабуватися незалежно.

Система використовує кілька типів сховищ даних, кожне з яких обране відповідно до вимог конкретного мікросервісу:

PostgreSQL - основне реляційне сховище для структурованих даних;

MongoDB - документоорієнтована база для зберігання гнучких або напівструктурованих даних;

Redis - сховище типу key-value, використовується для кешування, тимчасових сесій та швидкого доступу до часто використовуваних даних.

MongoDB використовується для збереження обладнання в сервісі Equipment, оскільки різні типи обладнання мають різні властивості та поля і використання документо-орієнтованої бази даних є логічним рішенням в цьому випадку.

Redis використовується в сервісі Open Order для збереження відкритих замовлень з геопросторовим індексом для швидкого пошуку за координатами.

В усіх інших сервісах, де важлива чітка та надійна структура збереження даних використовується PostgreSQL.

Для кращого розуміння сутностей системи було створено ER-діаграму (див. додаток Г).

3.4 Основний бізнес процес

Для кращого розуміння основного бізнес процесу, а саме створення та виконання замовлення було створено діаграму потоку даних (див. додаток Д). На ній відображені усі процеси, що мають бути виконані для успішного замовлення.

3.5 Приклади найцікавіших алгоритмів та методів

3.5.1 Алгоритм пошуку замовлень за геолокацією

Для пошуку відкритих замовлень за геолокацією використовується поточні координати працівника та порівнюються з координатами відкритих замовлень за допомогою GEO-функцій сховища Redis. Далі наведено код алгоритму пошуку:

```
/**
 * Retrieves a list of open orders within a specified radius from a given location.
 *
 * @param request The request containing the location (latitude, longitude) and radius.
 * @return A list of OrderOpeningMessageDto objects representing the open orders within the radius.
 */
public List<OpenOrderDto> getOpenOrdersWithinRadius(OpenOrdersRequestDto request) {
    double latitude = request.getLatitude();
    double longitude = request.getLongitude();
    double radius = request.getRadius(); // Radius in kilometers.

    long start = System.currentTimeMillis();
    GeoResults<RedisGeoCommands.GeoLocation<String>> results = redisTemplate.opsForGeo()
            .radius(GEO_KEY, new Circle(new Point(longitude, latitude),
                    new Distance(radius, RedisGeoCommands.DistanceUnit.KILOMETERS)));
    if (results == null || results.getContent().isEmpty()) {
        log.info("No open orders found within the specified radius.");
        return List.of();
    }
    List<OpenOrderDto> openOrders = mapOrderIdsToOpenOrderDtos(results);
```

```
    log.info("Retrieved {} open orders within {} km radius from lat: {},
long: {} in {} ms",
            openOrders.size(), radius, latitude, longitude,
System.currentTimeMillis() - start);
    return openOrders;
}
```

3.5.2 Створення планів замовлень

Для реалізації планування майбутніх замовлень до сутності Order завчасно було додано поле startDate (дата початку замовлення) і тільки при настанні цієї дати замовлення відкривається та стає доступним для працівників. До цього моменту замовлення зберігаються в базі даних та доступні тільки для клієнта, для автоматично відкриття замовлень при настанні дати початку було створено сервіс-планувальник, який з заданим інтервалом часу перевіряє чи не настав час відкрити замовлення. Далі наведено код цього сервісу:

```
@Scheduled(fixedDelay = 600000) // Runs every 10 minutes
public void scheduleOrderOpening() {
    log.info("Starting scheduled task to open orders");
    orderRepository.findAllCreatedOrders().forEach(
            order -> {
                if (order.getStartDate().isBefore(LocalDate.now()) ||
order.getStartDate().isEqual(LocalDate.now())) {
                    rabbitMQProducer.sendOrderOpeningMessage(order, "");
                    order.setStatus(OrderStatus.OPEN);
                    orderRepository.save(order);
                    log.info("Order {} is now open", order.getId());
                }

            }
    );
}
```

3.5.3 Поповнення рахунку

Для поповнення рахунку було реалізовано інтеграцію з платіжною системою PayPal. Для поповнення користувач має задати суму та отримати посилання для оплати, згенероване PayPal API. Також після успішної оплати, або її скасування задано “return url” для обробки операції на нашому боці. Далі наведено код звернення до PayPal API для отримання посилання оплати:

```
public String createPaymentUrl(Payment payment) throws IOException {
    // Format the payment amount to two decimal places using US locale.
    DecimalFormat df = new DecimalFormat("0.00",
DecimalFormatSymbols.getInstance(Locale.US));
```

```
    String amount = df.format(payment.getAmount());

    // Create a PayPal OrderRequest with the payment details.
    OrderRequest orderRequest = new OrderRequest()
            .checkoutPaymentIntent("CAPTURE")
            .purchaseUnits(List.of(
                    new PurchaseUnitRequest()
                            .referenceId(payment.getId())
                            .amountWithBreakdown(new AmountWithBreakdown()
                                    .currencyCode(payment.getCurrency())
                                    .value(amount)
                            )
            ))
            .applicationContext(new ApplicationContext()
                    .returnUrl("http://" + backendHost +
"/api/v1/billing/success?paymentId=" + payment.getId())
                    .cancelUrl("http://" + backendHost +
"/api/v1/billing/cancel/" + payment.getId() + "?cancelled=true")
            );

    // Send the order creation request to PayPal.
    OrdersCreateRequest request = new OrdersCreateRequest()
            .requestBody(orderRequest);

    HttpResponse<Order> response = client.execute(request);
    Order order = response.result();

    // Extract the approval URL from the PayPal response.
    String approveUrl = order.links().stream()
            .filter(l -> "approve".equals(l.rel()))
            .findFirst()
            .orElseThrow(() -> new IllegalStateException("No approve link
in PayPal response"))
            .href();

    // Update the Payment entity with the external payment ID, approval
URL, and status.
    payment.setExternalPaymentId(order.id());
    payment.setExternalPaymentUrl(approveUrl);
    payment.setStatus(PaymentStatus.PENDING);
    paymentRepository.save(payment);

    log.info("Created PayPal Order {} for payment {} - redirect URL: {}",
order.id(), payment.getId(), approveUrl);
    return approveUrl;
}
```

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Базова структура сервісів

Як було зазначено раніше, система створена базуючись на мікросервісній архітектурі. Кожен мікросервіс у проєкті створений за одним шаблоном та однакову базову організацію пакетів. Такий підхід допомагає стандартизувати кодову базу, полегшує навігацію, тощо. Далі наведено приклад структури сервісу:

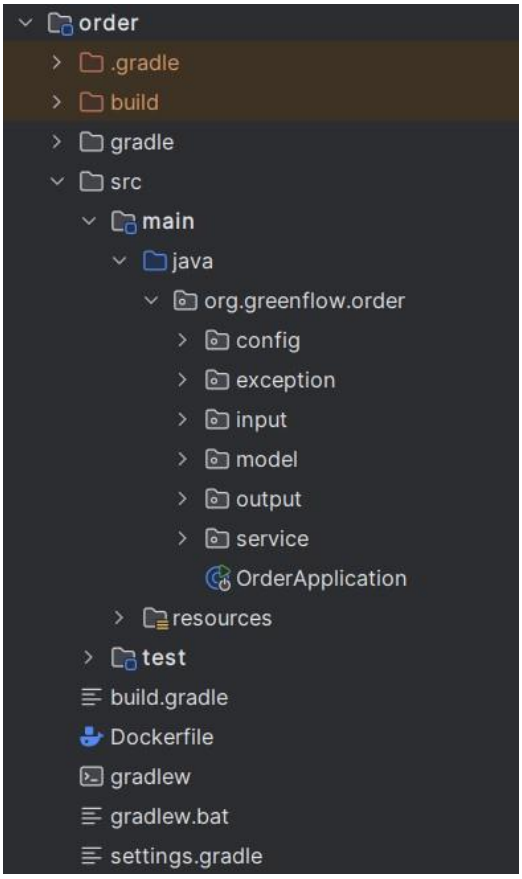

Рис 4.1 – базова структура сервісу

Основні пакети, що використовуються в кожному модулі, описані нижче:

— config: містить конфігураційні класи, які відповідають за налаштування сервісу, наприклад, конфігурацію безпеки, CORS, взаємодію з базою даних, зовнішніми API, або інші параметри середовища;

— exception: містить класи для обробки виняткових ситуацій, зокрема власні винятки та глобальний обробник помилок (ExceptionHandler);

— input: включає класи, що використовуються для отримання та обробки запитів ззовні, наприклад HTTP контролери або споживачі (consumer) черги повідомлень;

— model: містить DTO класи та основні доменні моделі, які відображають сутності предметної області. Ці класи мають анотації JPA для взаємодії з базою даних;

— output: включає класи, що відповідають за відправку запитів з цього сервісу до інших, наприклад HTTP Client, або продюсери (producer) в чергу повідомлень;

— service: реалізує бізнес-логіку системи, тут розміщені основні сервіси, які виконують обробку даних, взаємодіють з репозиторіями та іншими компонентами;

— util: містить допоміжні класи та функції.

Ця структура дозволяє досягти високого рівня модульності, забезпечити інкапсуляцію відповідальностей і підтримувати принцип розділення обов'язків у кожному мікросервісі.

4.2 Реалізація API Gateway

API Gateway є єдиною точкою входу для всіх HTTP-запитів, що надходять від клієнтської частини. Реалізований на базі фреймворку Spring Cloud Gateway. Його задачі:

— маршрутизує запити до відповідних мікросервісів (Route definitions);

— перевіряє валідність JWT (фільтр автентифікації);

— застосовує кросс-доменну політику (CORS).

Далі наведено код основної конфігурації:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder) {
    return builder.routes()
            .route("auth-service", r -> r.path("/api/v1/auth/**")
                    .uri("http://" + AUTH_SERVICE_HOST))
            .route("client", r -> r.path("/api/v1/client/**")
                    .filters(f -> f.filter(filter))
                    .uri("http://" + CLIENT_SERVICE_HOST))
            .route("worker", r -> r.path("/api/v1/worker/**")
                    .filters(f -> f.filter(filter))
```

```
                .uri("http://" + WORKER_SERVICE_HOST))
            .route("garden", r -> r.path("/api/v1/garden/**")
                .filters(f -> f.filter(filter))
                .uri("http://" + GARDEN_SERVICE_HOST))
            .route("order", r -> r.path("/api/v1/order/**",
"/api/v1/services/**")
                .filters(f -> f.filter(filter))
                .uri("http://" + ORDER_SERVICE_HOST))
            .route("open-order", r -> r.path("/api/v1/open-order/**")
                .filters(f -> f.filter(filter))
                .uri("http://" + OPEN_ORDER_SERVICE_HOST))
            .route("equipment", r -> r.path("/api/v1/equipment/**",
"/api/v1/warehouse/**", "/api/v1/leasing/**")
                .filters(f -> f.filter(filter))
                .uri("http://" + EQUIPMENT_HOST))
            .build();
}
```

Цей метод створює набір маршрутів для, які перенаправляють HTTP-запити з певними шляхами (наприклад, /api/v1/auth/**, /api/v1/client/**) до відповідних мікросервісів (auth-service, client-service тощо) за їхніми URI. Адреси хостів (AUTH_SERVICE_HOST, CLIENT_SERVICE_HOST тощо) задаються як змінні й можуть змінюватися в залежності від середовища (development, production). Також можна відзначити застосування фільтру безпеки до маршрутів.

4.3 Сервіс авторизації (auth-service)

Auth Service реалізовано з використанням Spring Security із підтримкою реєстрації та авторизації користувача через email/пароль, генерування та валідації JWT-токенів.

4.3.1 Реєстрація/авторизація користувачів

Створено контролер для обробки вхідних HTTP-запитів на реєстрацію та логін користувачів:

```
@PostMapping("/login")
public ResponseEntity<?> login(@Valid @RequestBody LoginRequest
loginRequest) {
    User user = authService.login(loginRequest);
    String jwtToken = jwtService.generateToken(user);
    LoginResponse response = LoginResponse.builder()
            .username(user.getEmail())
            .jwtToken(jwtToken)
            .roles(user.getRoles().stream()
                .map(Role::getAuthority)
                .collect(Collectors.toSet()))
            .build();
```

```
    return ResponseEntity.ok(response);
}

@PostMapping("/register")
public ResponseEntity<?> register(@Valid @RequestBody SignupRequest
signUpRequest) {
    User user = authService.registerUser(signUpRequest);
    String jwtToken = jwtService.generateToken(user);
    LoginResponse response = LoginResponse.builder()
            .username(user.getEmail())
            .jwtToken(jwtToken)
            .roles(user.getRoles().stream()
                    .map(Role::getAuthority)
                    .collect(Collectors.toSet()))
            .build();
    return ResponseEntity.status(201).body(response);
}
```

Ці 2 методи обробляють запити на логін та реєстрацію відповідно. Як можна бачити, вони також генерують JWT токен та повертають їх на клієнт разом з ролями та електронною адресою.

Результат:

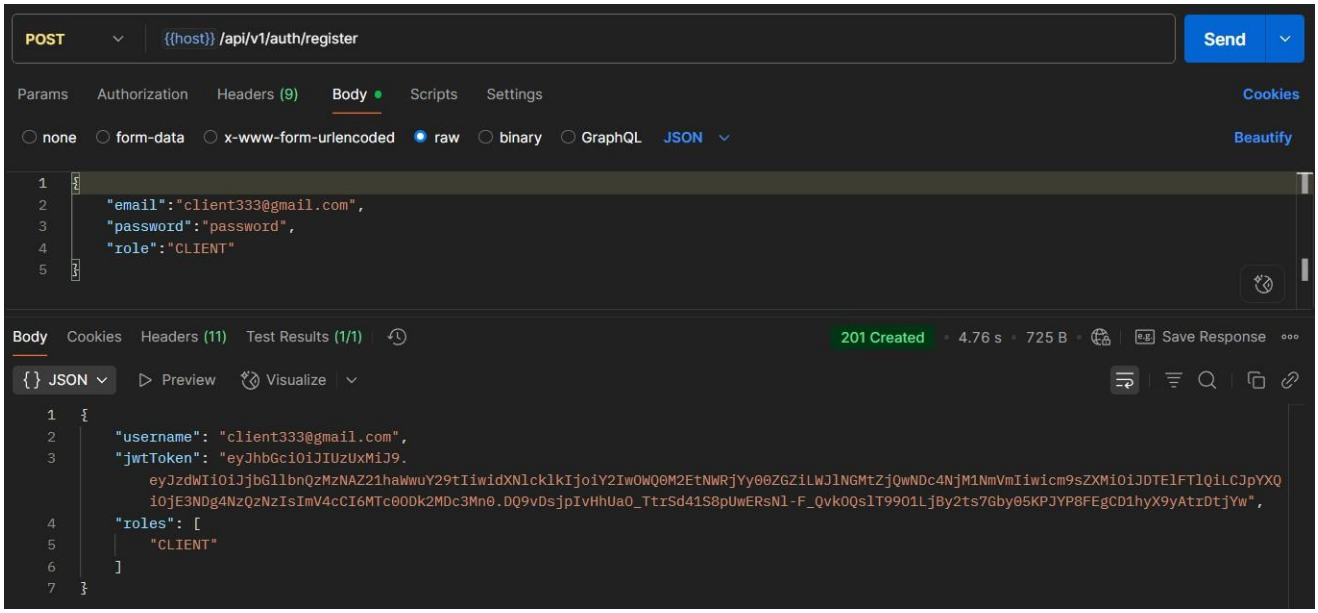

Рис 4.1 – результат реєстрації користувача

4.3.2 Генерація JWT токенів

На основі даних користувача, створюються JWT токени, які в подальшому використовуються для авторизації користувача. Далі наведено код генерації JWT токена на основі користувача (User):

```
@Value("${jwt.secret}")
private String secretKey;

@Value("${jwt.expiration}")
private long expiration;

private Key getSigningKey() {
    return Keys.hmacShaKeyFor(secretKey.getBytes(StandardCharsets.UTF_8));
}

public String generateToken(User user) {
    log.debug("Generating token for user: {}", user);
    String roles = user.getRoles().stream()
            .map(Role::getAuthority)
            .collect(Collectors.joining(","));
    return Jwts.builder()
            .subject(user.getEmail())
            .claim("userId", user.getId())
            .claim("roles", roles)
            .issuedAt(Date.from(Instant.now()))
            .expiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(getSigningKey())
            .compact();
}
```

Цей код — це частина сервісу для генерації JWT-токенів у Spring-застосунку. Він використовує бібліотеку io.jsonwebtoken.Jwts для створення токена на основі інформації про користувача: email, ID і ролі. Значення секретного ключа secretKey та часу життя токена expiration підставляються з конфігурації змінних середовища. Метод створює токен, підписує його за допомогою HMAC та секретного ключа, і встановлює термін дії. Секретний ключ зберігається поза кодом, що забезпечує конфіденційність.

Можемо перевірити валідність ключа за допомогою відкритого онлайн сервісу jwt.io:

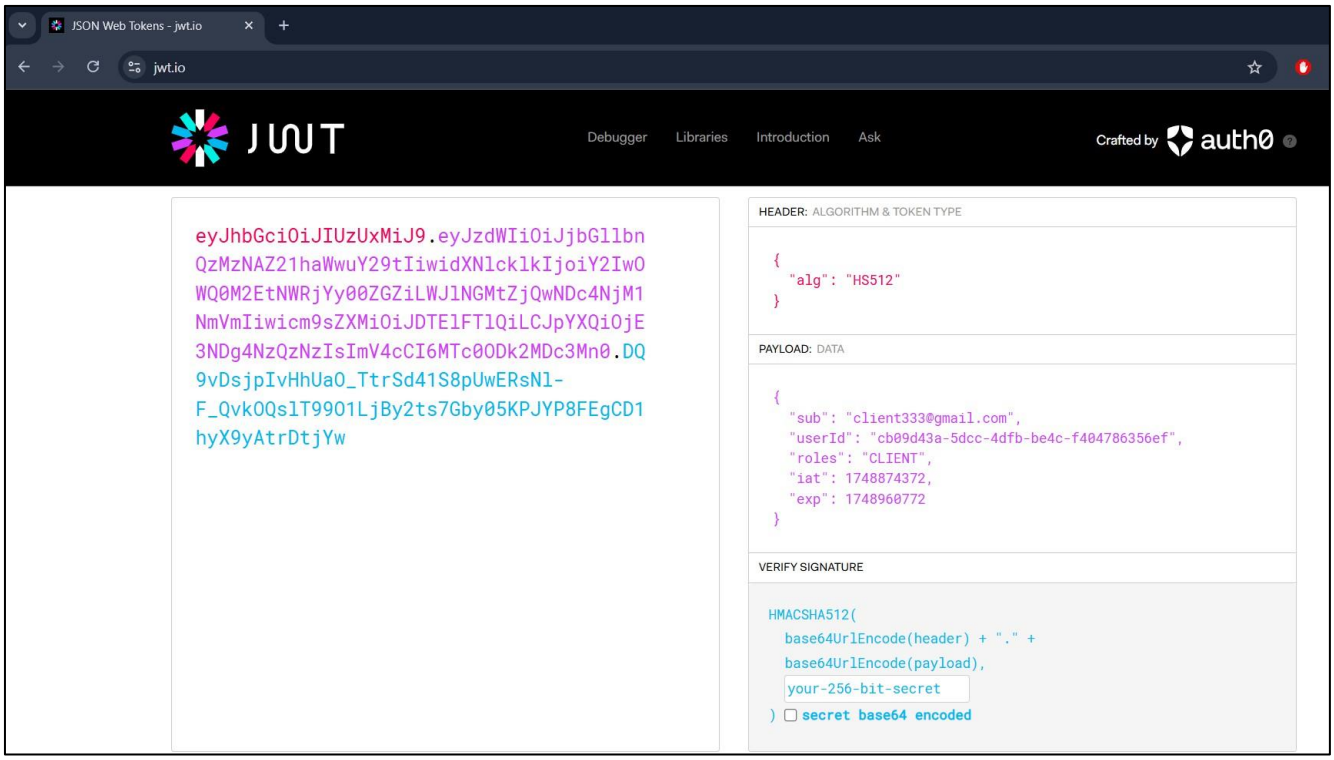

Рис 4.2 – результат валідації токена

Як можемо бачити в навантаженні (Payload) токена зберігається інформація про раніше створеного користувача, його електрона адреса, унікальний ідентифікатор, ролі та час життя токену. Ця інформація буде використана далі при авторизації.

4.4 Пошук відкритих замовлень

Для цього було створено окремий модуль open-order, який підключений до сховища Redis. Для пошуку замовлень неподалік створено контролер та ендпоінт, який приймає поточні координати працівника, та радіус, в якому він шукає замовлення. Далі наведено результат пошуку:

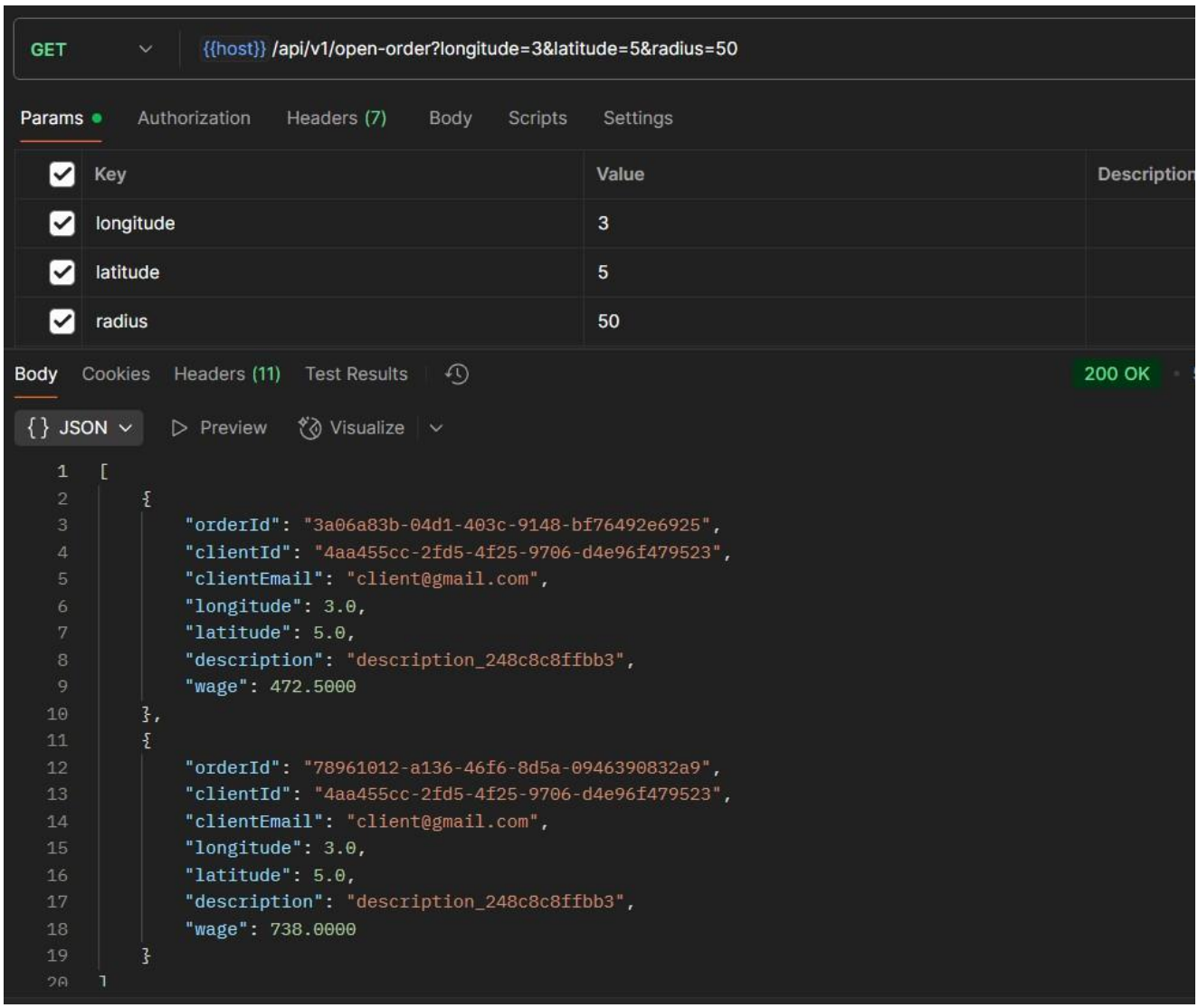

Рис 4.3 – результат пошуку відкритих замовлень

4.5 Обробка помилок

Усі винятки під час виконання запитів системи обробляються централізовано за допомогою кастомних винятків і глобального обробника, який формує структуровану відповідь для клієнта. Це дозволяє повернути клієнту уніфікований формат помилки з кодом статусу, списком повідомлень і стек-трейсом (для діагностики) та впіймати стандартні винятки Spring (валидація, нечитабельне тіло запиту тощо) та повернути коректний HTTP-код і зрозуміле повідомлення.

Далі наведено код кастомного винятку:

```
@Getter
public class GreenFlowException extends RuntimeException {

    int statusCode;

    public GreenFlowException(int statusCode, String msg) {
        super(msg);
        this.statusCode = statusCode;
    }

    public GreenFlowException(int statusCode, String message, Throwable
cause) {
        super(message, cause);
        this.statusCode = statusCode;
    }

}
```

Цей клас розширює функціонал стандартного винятку RuntimeException та додає поле statusCode для збереження HTTP статусу помилки.

Також було створено обробник винятків, який перехоплює викинуті винятки та обробляє їх, повертаючи HTTP відповідь з правильним статусом та зрозумілим повідомленням. Далі наведено код обробки раніше зазначеного винятку:

```
@RestControllerAdvice
public class CustomExceptionHandler {

    @ExceptionHandler(GreenFlowException.class)
    public ResponseEntity<ResponseErrorDto>
handleGreenFlowException(GreenFlowException e) {
        int status = e.getStatusCode();
        List<String> errorMessages = new ArrayList<>();
        if (e.getCause() != null) {
            errorMessages.add(e.getCause().getMessage());
        }
        errorMessages.add(e.getMessage());
        var error = ResponseErrorDto.builder()
                .statusCode(status)
                .errorMessage(errorMessages)
                .stackTrace(Arrays.stream(e.getStackTrace())
                        .map(StackTraceElement::toString)
                        .toList())
                .build();
        return ResponseEntity.status(status).body(error);
    }
      …
```

Далі наведено приклад використання даного підходу обробки помилок під час запиту на реєстрацію користувача з вже зайнятою електронною адресою:

```
public User registerUser(@Valid SignupRequest signUpRequest) {
        if (existsByEmail(signUpRequest.getEmail())) {
        throw new GreenFlowException(400, "Email already in use: " +
signUpRequest.getEmail());
      ...
    }
```

Результат:

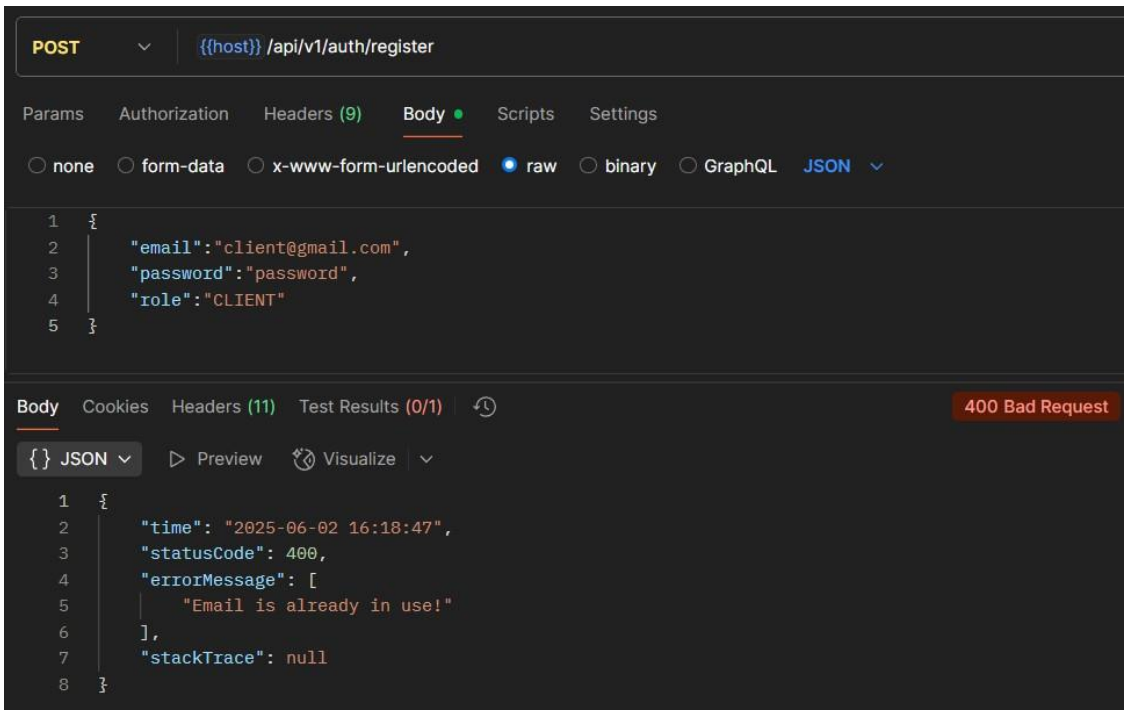

Рис 4.4 – результат обробки винятку

Як можемо бачити, повертається коректний HTTP статус, та зрозуміле повідомлення. В цьому прикладі стек трейс порожній, оскільки в даному випадку він не потрібен.

4.6 Асинхронна комунікація

В системі використовується брокер повідомлень RabbitMQ для асинхронної комунікації між сервісами. Далі наведено приклад конфігурації:

```
@Bean
public TopicExchange orderExchange() {
    return new TopicExchange(RabbitMQConstants.ORDER_EXCHANGE);
}

@Bean(name = "orderOpeningQueue")
public Queue orderOpeningQueue() {
    return new Queue(RabbitMQConstants.ORDER_OPENING_QUEUE, true);
}
```

```
@Bean
public Binding orderOpeningBinding(@Qualifier("orderOpeningQueue") Queue
queue, TopicExchange orderExchange) {
    return
BindingBuilder.bind(queue).to(orderExchange).with("order.opening.#");
}
```

Визначаються такі об’єкти як Exchange, Queue та Binding для їх пов’язування між собою. В даному випадку створено чергу order.opening, яка використовується для передачі відкритого замовлення з сервісу «Order» до сервісу «OpenOrder».

Можемо підключитись до контрольної панелі RabbitMQ та бачити створені черги та іншу інформацію.

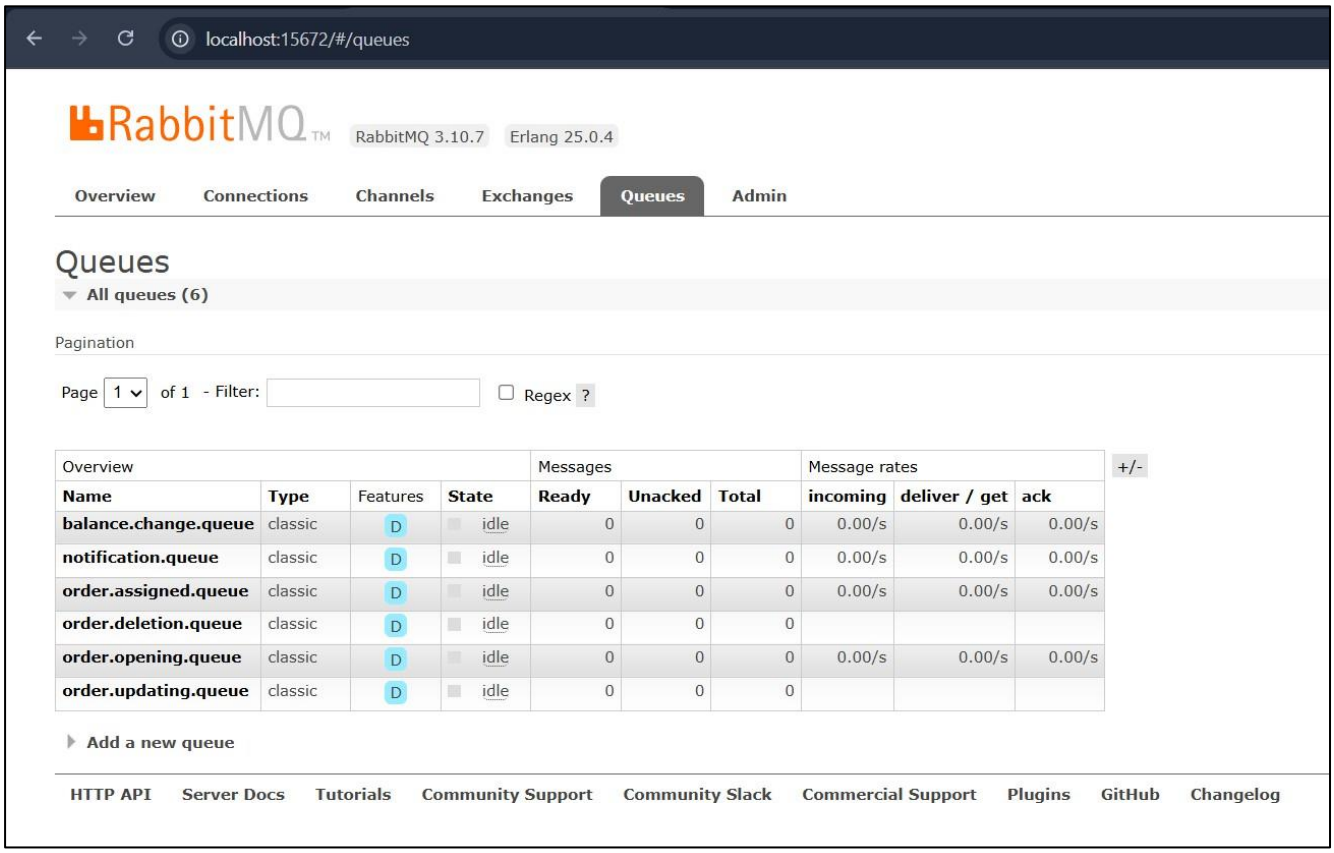

Рис 4.5 – контрольна панель RabbitMQ

Далі наведено код Producer, що відправляє повідомлення та Consumer, який приймає:

```
public void sendEmailNotification(EmailNotificationMessage message) {
    try {
```

```
rabbitTemplate.convertAndSend(RabbitMQConstants.NOTIFICATION_EXCHANGE,
                RabbitMQConstants.NOTIFICATION_QUEUE, message);
        log.info("Email notification message sent to RabbitMQ for user:
{}", message.userId());
    } catch (Exception e) {
        log.error("Failed to send order completed notification to
RabbitMQ", e);
        throw new
GreenFlowException(HttpStatus.INTERNAL_SERVER_ERROR.value(),
                FAILED_TO_SEND_MESSAGE_TO_RABBIT_MQ, e);
    }
}


@RabbitListener(queues = RabbitMQConstants.NOTIFICATION_QUEUE)
public void consumeOrderOpeningMessage(EmailNotificationMessage email) {
    log.info("Received email notification message to: {}", email.userId());
    try {
        emailService.sendEmail(email);
    } catch (Exception e) {
        log.error("Error sending email notification: {}", e.getMessage());
    }
}
```

Як можемо побачити, Producer відправляє повідомлення в чергу, а Consumer слухає цю чергу і чекає на повідомлення.

4.7 Безпека

Безпека системи реалізована за багаторівневим підходом, що охоплює захист мережі, автентифікацію, авторизацію, захист даних, а також безпечне зберігання конфігураційних секретів. Нижче наведено опис ключових компонентів безпеки.

4.7.1 Захист HTTP-з’єднань (HTTPS)

Усі зовнішні HTTP-запити до Api Gateway та внутрішні запити між сервісами виконуються через HTTPS

На рівні Kubernetes налаштовано Ingress із TLS-сертифікатом Let’s Encrypt.

4.7.2 Автентифікація користувачів (JWT)

Після успішної автентифікації користувач отримує JWT токен, який підписано приватним ключем в Auth Service, в якому міститься необхідна інформація для авторизації користувача.

4.7.3 Авторизація

Для всіх наступних запитів клієнт відправляє отриманий раніше JWT токен як HTTP Header. Цей токен парситься в сервісі Api Gateway, якщо він валідний то запит мутується, і інформація про користувача витягується з токену та додається до кастомних HTTP Header, та відправляється далі до відповідного сервісу. Далі наведено код фільтру Api Gateway, який парсить JWT токен:

```
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
    ServerHttpRequest request = exchange.getRequest();
    if (isClosedPath(request)) {
        return onError(exchange, HttpStatus.FORBIDDEN);
    }

    if (isAuthMissing(request)) {
        return onError(exchange, HttpStatus.UNAUTHORIZED);
    }

    final String token = getAuthHeader(request).substring(7);
    if (jwtUtil.isInvalid(token)) {
        return onError(exchange, HttpStatus.FORBIDDEN);
    }

    ServerWebExchange updatedExchange = updateRequest(exchange, token);

    return chain.filter(updatedExchange);
}
/**
 * Update the request with the user information. Add info to the request
headers.
 */
private ServerWebExchange updateRequest(ServerWebExchange exchange, String
token) {
    String userId = jwtUtil.extractUserId(token);
    String roles = jwtUtil.extractRoles(token);
    String email = jwtUtil.extractEmail(token);

    ServerHttpRequest updatedRequest = exchange.getRequest().mutate()
            .header(CustomHeaders.X_USER_ID, userId)
            .header(CustomHeaders.X_ROLES, roles)
            .header(CustomHeaders.X_EMAIL, email)
            .build();

    return exchange.mutate().request(updatedRequest).build();
}
```

Як можна побачити, в заголовки запиту додається інформація про ідентифікатор користувача, його ролі та електрона пошта. Ця інформація використовується в сервісі, який обробляє запит. Кожен сервіс має власний фільтр, який обробляє ці заголовки. Далі наведено приклад такого фільтру:

```
protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

    String userId = request.getHeader(CustomHeaders.X_USER_ID);
    String roles = request.getHeader(CustomHeaders.X_ROLES);
    String email = request.getHeader(CustomHeaders.X_EMAIL);

    if (userId != null && roles != null && email != null) {
        Set<GrantedAuthority> authorities = Set.of(
                        roles.split(",")).stream()
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toSet());

        UsernamePasswordAuthenticationToken auth = new
UsernamePasswordAuthenticationToken(
                userId, null, authorities
        );
        putDetails(auth, email);
        SecurityContextHolder.getContext().setAuthentication(auth);
        log.debug("Authentication Success: {}", auth.getName());
        var details = (Map<String, String>) auth.getDetails();
        log.debug("Email: {}", details.get("email"));
    }
    chain.doFilter(request, response);
}
```

Як можемо бачити, цей фільтр бере інформацію про користувача, його ідентифікатор, ролі та електрону пошту та створює об’єкт “authentication”, який керується контекстом Spring, та дозволяє використовувати зручні інструменти авторизації. Далі наведено приклад коду використання інструментів Spring Security для відокремлення логіки за ролями користувачів:

```
@GetMapping("/my")
@PreAuthorize("hasAuthority('CLIENT')")
public ResponseEntity<?>
getMyGardens(@RequestHeader(CustomHeaders.X_USER_ID) String userId) {
    List<Garden> gardens = gardenService.getGardensByOwnerId(userId);
    return ResponseEntity.ok(gardens);
}
```

В цьому прикладі маємо анотацію PreAuthorize, яка визначає, що цей ендпоінт доступний тільки для користувачів, які мають роль CLIENT, і для інших доступ буде закритим. Спробуємо відправити запит авторизуючись з іншою роллю:

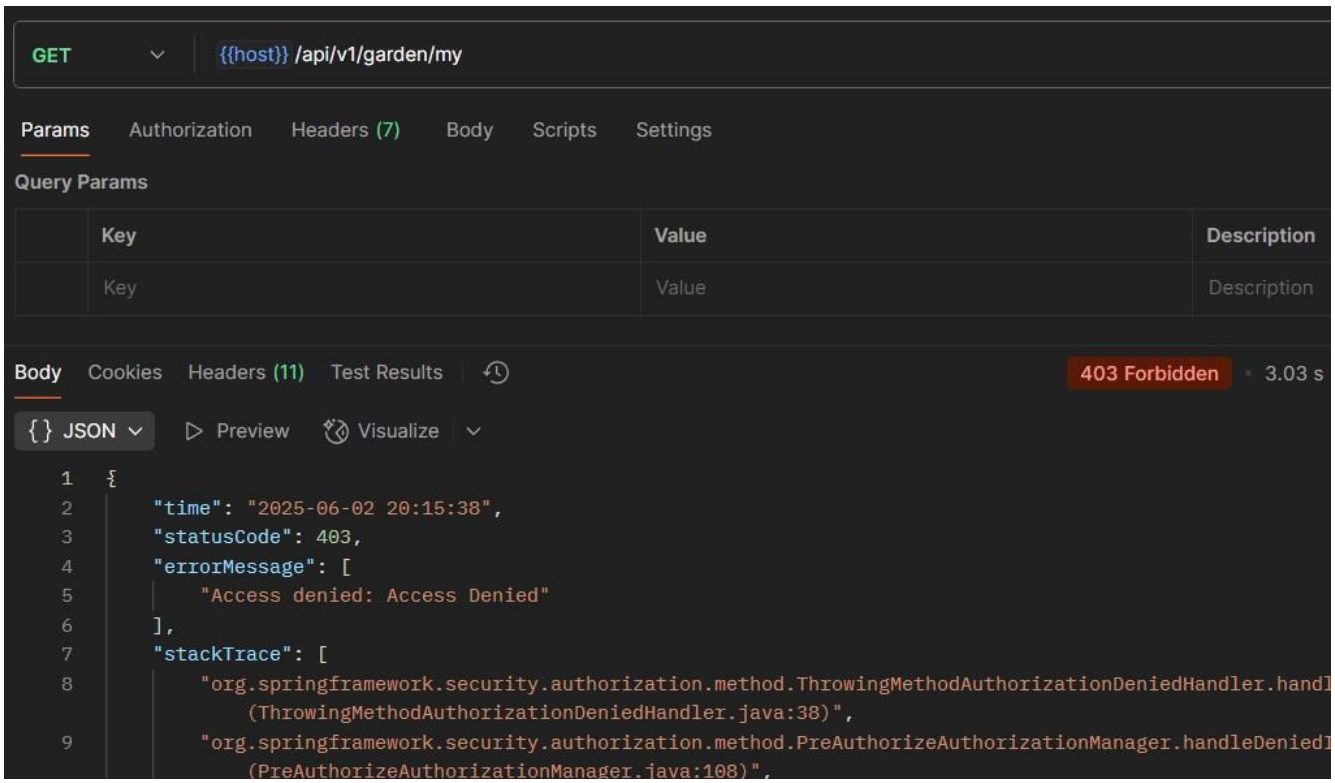

Рис 4.6 – результат невдалого запиту

Також реалізовано логіку захисту доступу до ресурсів. Як було сказано раніше, в запиті передається інформація для авторизації користувача, його ідентифікатор, тощо. Для кожної операції з ресурсами було створено додатковий шар захисту, перевірку чи насправді цей користувач має доступ до нього. Далі наведено код цієї перевірки:

```
public void deleteGarden(@NotBlank String userId, @NotNull Long gardenId) {
    Garden garden = gardenRepository.findById(gardenId)
            .orElseThrow(() -> new
GreenFlowException(HttpStatus.NOT_FOUND.value(), "Garden not found"));
    if (!garden.getOwnerId().equals(userId)) {
        throw new GreenFlowException(HttpStatus.FORBIDDEN.value(), "You do
not have access to this resource");
    }
    gardenRepository.delete(garden);
    log.info("Client {} deleted garden: {}", userId, garden.getId());
}
```

В цьому прикладі для видалення саду клієнта, спочатку перевіряється чи авторизований користувач дійсно їм володіє. Спробуємо видалити сад, який нам не належить:

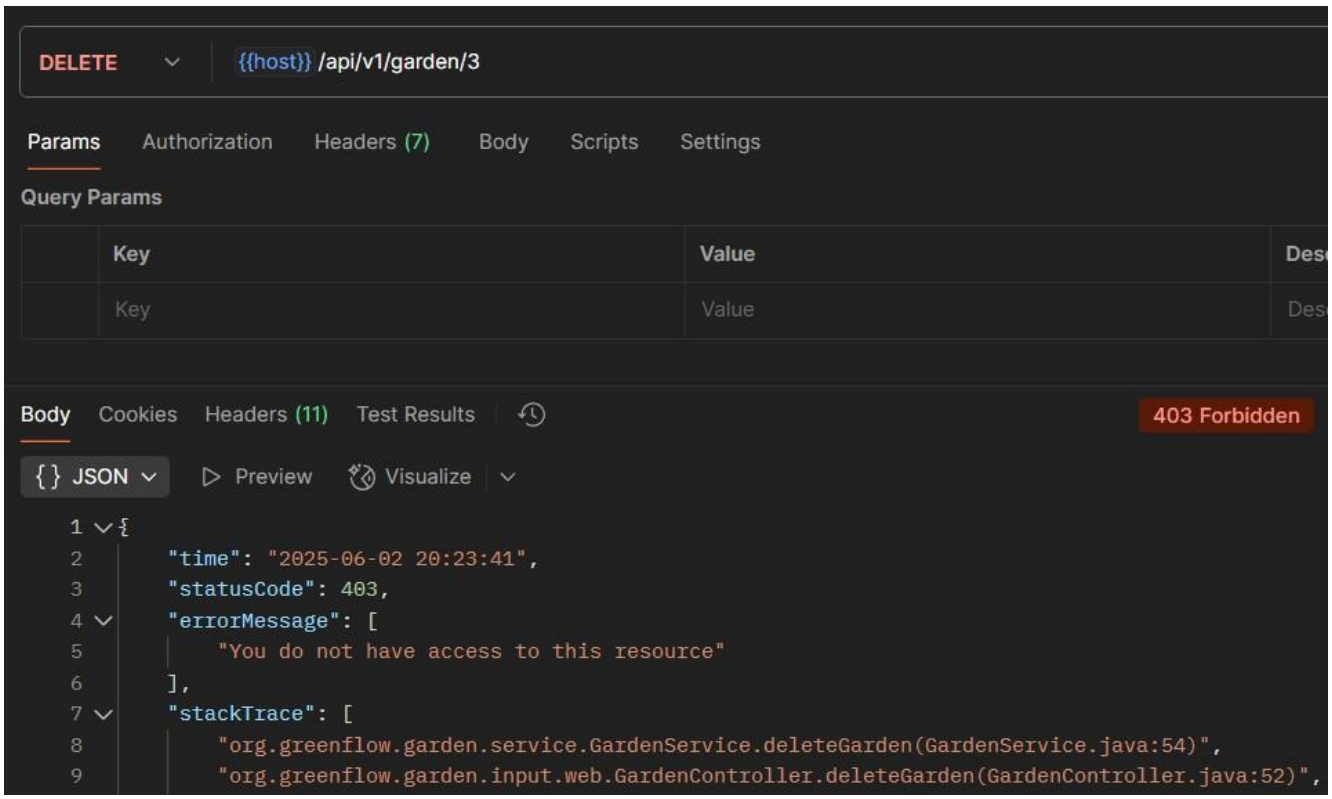

Рис 4.7 – результат невдалого доступу до ресурсу

4.7.4 Захист системних ендпоінтів

У проєкті передбачено ендпоінти, що використовуються виключно для внутрішньої взаємодії між сервісами. Щоб обмежити доступ до них для зовнішніх користувачів, у міжсервісних запитах обов'язково передається спеціальний API-токен. У разі його відсутності або некоректності доступ до ендпоінту буде заборонено.

Далі наведено код реалізації:

```
@Bean
public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder
            .additionalInterceptors((request, body, execution) -> {
                request.getHeaders().add(X_INTERNAL_TOKEN, internalToken);
                return execution.execute(request, body);
            })
            .build();
}
```

У сервісі відправнику створюється бін для надсилання REST-запитів, у який одразу додається заголовок з внутрішнім API-токеном. Це забезпечує

автоматичне включення токена в усі міжсервісні запити без потреби додавати його вручну.

У сервісі отримувачі запит валідується на наявність токену. Далі наведено приклад реалізації:

```
@InternalAuth
@GetMapping("/email")
public String getUserEmail(@RequestParam("userId") @NotBlank String userId)
{
    return authService.getClientEmailFromAuthService(userId);
}
```

Анотація @InternalAuth позначає метод як закритий для зовнішніх запитів і вимагає наявності дійсного внутрішнього API-токена в заголовку запиту. Нижче наведено код валідатора, який перевіряє цей токен:

```
@Aspect
@Component
public class InternalAuthAspect {

    @Value("${api.internalApiToken}")
    private String expectedToken;

    @Around("@within(org.greenflow.common.util.InternalAuth) ||
@annotation(org.greenflow.common.util.InternalAuth)")
    public Object validateInternalToken(ProceedingJoinPoint joinPoint)
throws Throwable {
        ServletRequestAttributes attrs = (ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        if (attrs == null) {
            throw new GreenFlowException(403, "No request context
available");
        }

        HttpServletRequest request = attrs.getRequest();
        String token = request.getHeader(X_INTERNAL_TOKEN);

        if (!expectedToken.equals(token)) {
            throw new GreenFlowException(403, "Invalid or missing internal
token");
        }

        return joinPoint.proceed();
    }
}
```

Цей підхід гарантує, що доступ до позначених ендпоінтів можливий лише за наявності дійсного токена в заголовку X-Internal-Token.

Спробуємо відправити прямий запит до закритого ендпоінту:

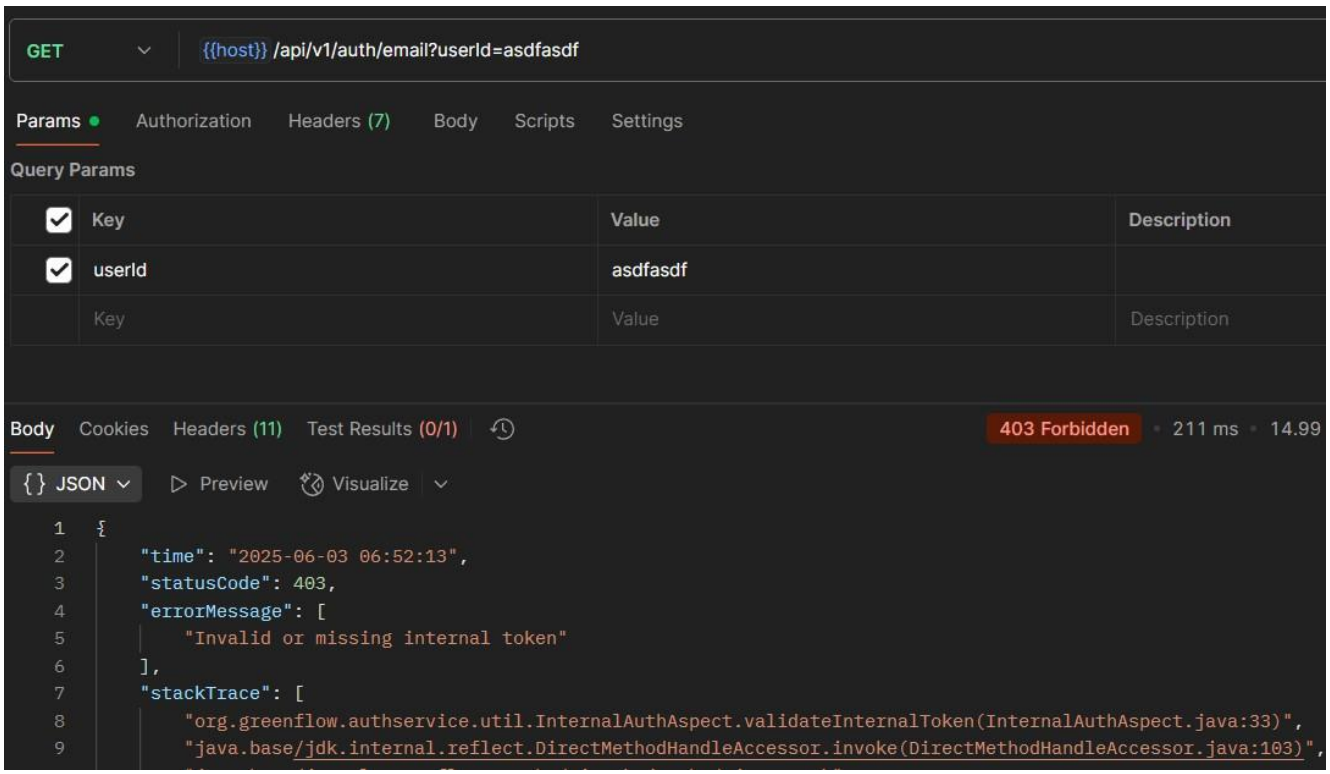

Рис 4.8 – результат невдалого доступу до закритого ендпоінту

4.8 Спільний код

В проєкті існує код, який повторюється в багатьох сервіса, щоб запобігти дублюванню коду, було вирішено створити окремий модуль-бібліотеку «common», який буде імпортуватись в інші сервіси як залежність. Це може бути код DTO-класи для обміну повідомленнями між сервісами, константи, загальні винятки, загальні утілітарні класи, тощо.

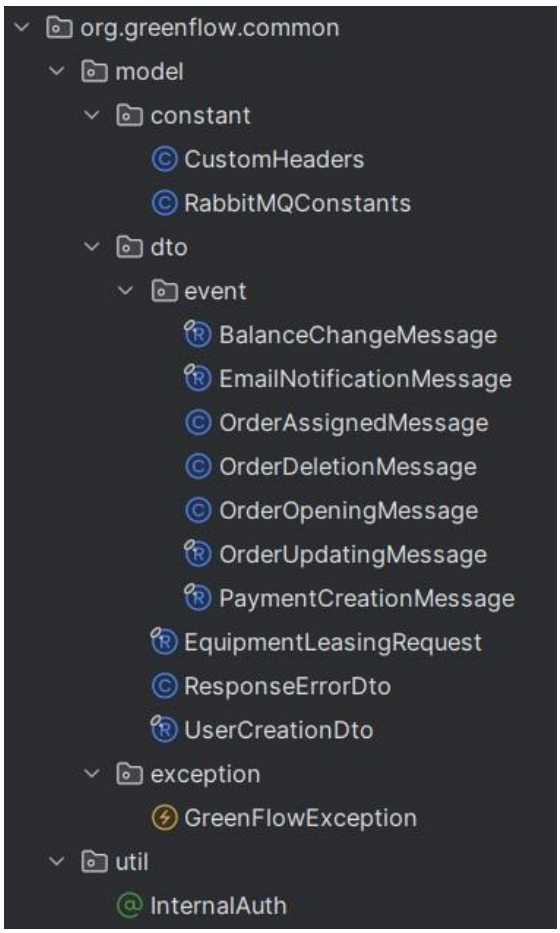

Рис 4.9 – структура модулю «common»

Для використання цього коду, модуль підключає його як залежність як будь-яку іншу бібліотеку:

```
implementation 'org.greenflow:common:1.0'
```

Таким чином, створення та використання спільного модуля «common» дозволяє суттєво зменшити технічний борг, прискорити розробку та забезпечити єдину точку істини для базових компонентів системи.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Юніт-тестування

Тестування програмного забезпечення є ключовим етапом життєвого циклу розробки, що дозволяє виявити помилки, перевірити коректність реалізації функціоналу та забезпечити стабільну роботу системи. У рамках розробки програмного забезпечення для системи було зосереджено увагу на юніт-тестуванні, яке забезпечує перевірку окремих модулів програми незалежно від інших компонентів.

Метою юніт-тестування є перевірка коректної роботи окремих методів та сервісів у ізоляції. Це дозволяє виявити логічні помилки на ранніх етапах, ще до інтеграції компонентів між собою.

Для тестування використовувались такі технології:

— JUnit 5, як основний фреймворк для написання тестів;

— Mockito, як бібліотека для створення моків та емуляції залежностей;

— AssertJ для зручної побудови перевірок;

— Spring Boot Test для підготовки середовища тестування конфігурацій та компонентів.

Було протестовано основні сервіси та компоненти, відповідальні за бізнес-логіку. Далі наведено приклад тесту зміни балансу клієнта:

```
@Test
void changeBalance_updatesUserBalanceSuccessfully() {
    String userId = "user123";
    BigDecimal initialBalance = BigDecimal.valueOf(50.0);
    BigDecimal changeAmount = BigDecimal.valueOf(20.0);
    UserBalance userBalance = new UserBalance();
    userBalance.setUserId(userId);
    userBalance.setBalance(initialBalance);


when(userBalanceRepository.findById(userId)).thenReturn(Optional.of(userBal
ance));

    BalanceChangeMessage message = new BalanceChangeMessage(userId,
changeAmount, "Test balance change");
    userBalanceService.changeBalance(message);

    assertEquals(initialBalance.add(changeAmount),
userBalance.getBalance());
```

```
    verify(userBalanceRepository).save(userBalance);
}
```

Результат виконання тестів:

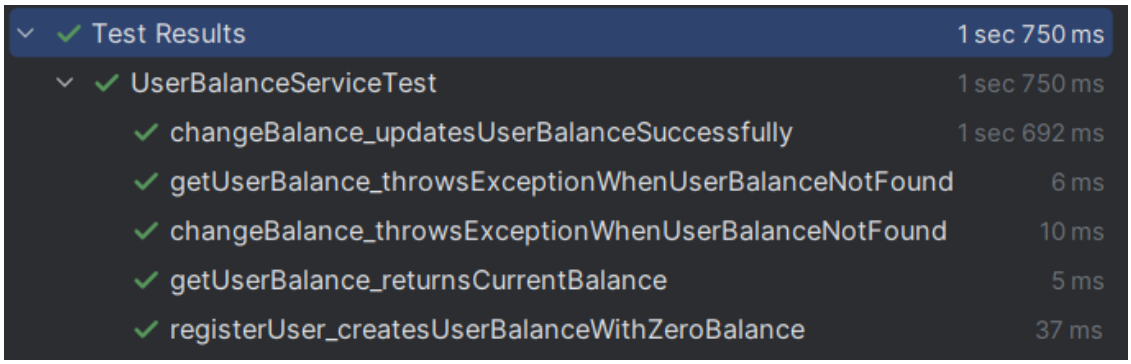

Рис 5.1 – результат виконання тестів

Таким чином, юніт-тестування відіграє важливу роль у забезпеченні якості програмного забезпечення GreenFlow. Надалі планується розширити тестове покриття, додати інтеграційні та навантажувальні тести для забезпечення ще більшої стабільності системи.

6 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У результаті реалізації програмного забезпечення система була успішно впроваджена у хмарному середовищі з використанням сучасних інструментів оркестрації, контейнеризації та CI/CD.

6.1 Хмарне середовище

Уся інфраструктура була розгорнута у хмарі DigitalOcean, де було створено Kubernetes-кластер з декількома робочими вузлами. Це забезпечило масштабованість, автоматичний перезапуск сервісів у разі збоїв та зручне оновлення системи без простою.

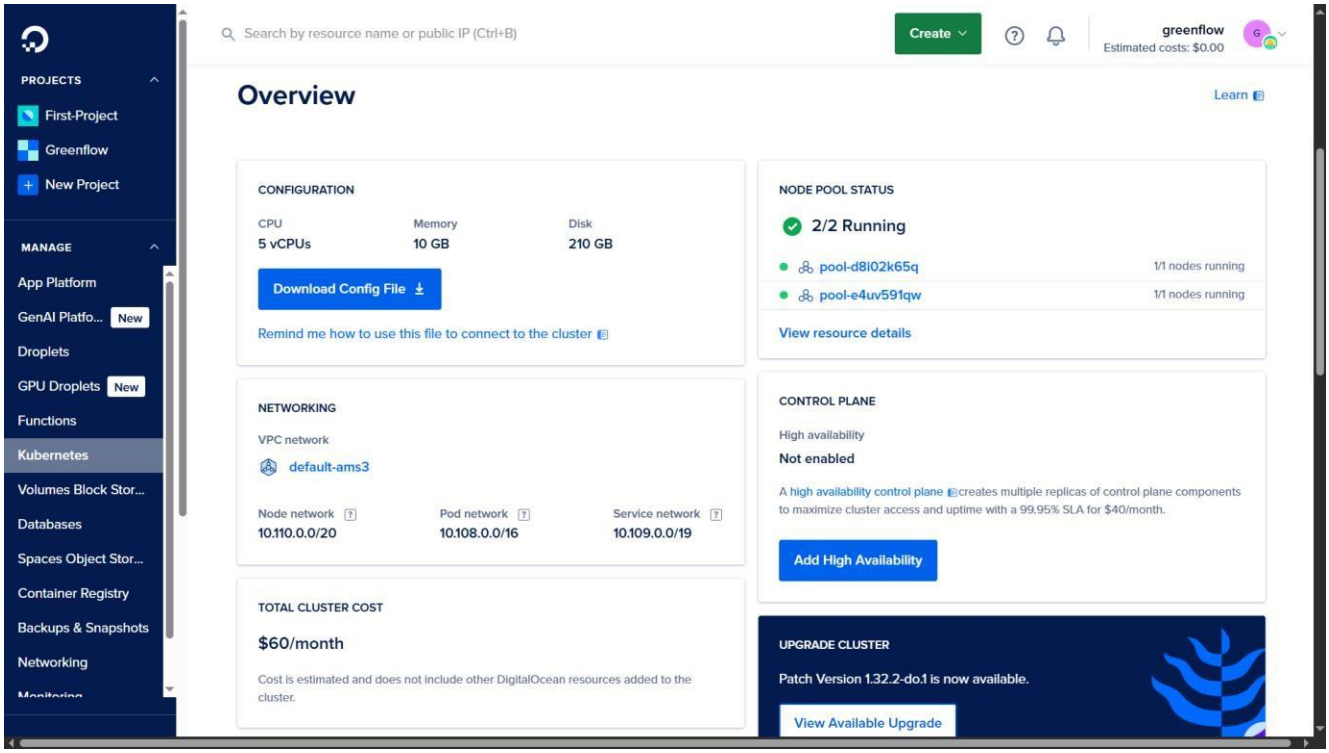

Рис 6.1 – контрольна панель Kubernetes кластеру

Як видно на скріншоті, було створено кластер з двома робочими вузлами (node pool) із загальною конфігурацією: 5 vCPU, 10 GB оперативної пам’яті та 210 GB дискового простору. Така конфігурація є достатньою для тестового розгортання системи без суттєвого навантаження.

Компоненти кластеру:

а) Pods — кожен мікросервіс було винесено в окремий pod;

б) Services — внутрішні та зовнішні сервіси для маршрутизації запитів;

в) Ingress Controller — використовується для керування HTTP(S)-трафіком;

г) Persistent Volumes — для збереження важливих даних, які не повинні втрачатися при перезапуску pod-ів.

Для розгортання сервісів були створені файли маніфести у форматі .yml, в яких описується поведінка розгортання. У прикладі маніфесту сервісів (див. додаток Ж) описано створення двох об’єктів Kubernetes кластера: Deployment та Service. Об’єкт Deployment відповідає за керування розгортанням застосунку, зокрема за створення потрібної кількості реплік, оновлення контейнерів та підтримку їх у робочому стані. Об’єкт Service відповідає за забезпечення стабільного мережевого доступу до запущених екземплярів застосунку (Pod-ів), створених Deployment.

6.2 CI/CD

Для автоматичного тестування та розгортання застосовано GitHub Actions.

При кожному push до репозиторію GitHub автоматично запускається pipeline, який виконує визначену послідовність дій (збірку, тестування, деплой). Код цих пайплайнів описано у файлах формату .yml, що зберігаються в каталозі .github/workflows. Приклад одного з таких файлів наведено в додатку К. Цей код описує CI/CD pipeline, який автоматично запускається при зміні коду у відповідному каталозі. Він виконує клонування репозиторію, налаштування JDK 21, тестування та публікацію загального модуля common, запуск тестів у auth-service, збірку та завантаження Docker-образу на Docker Hub, а також деплой оновленого образу сервісу до Kubernetes-кластера з оновленням Deployment.

Цей процес повністю автоматизований, що дозволяє зменшити час впровадження нових змін і забезпечити стабільність системи.

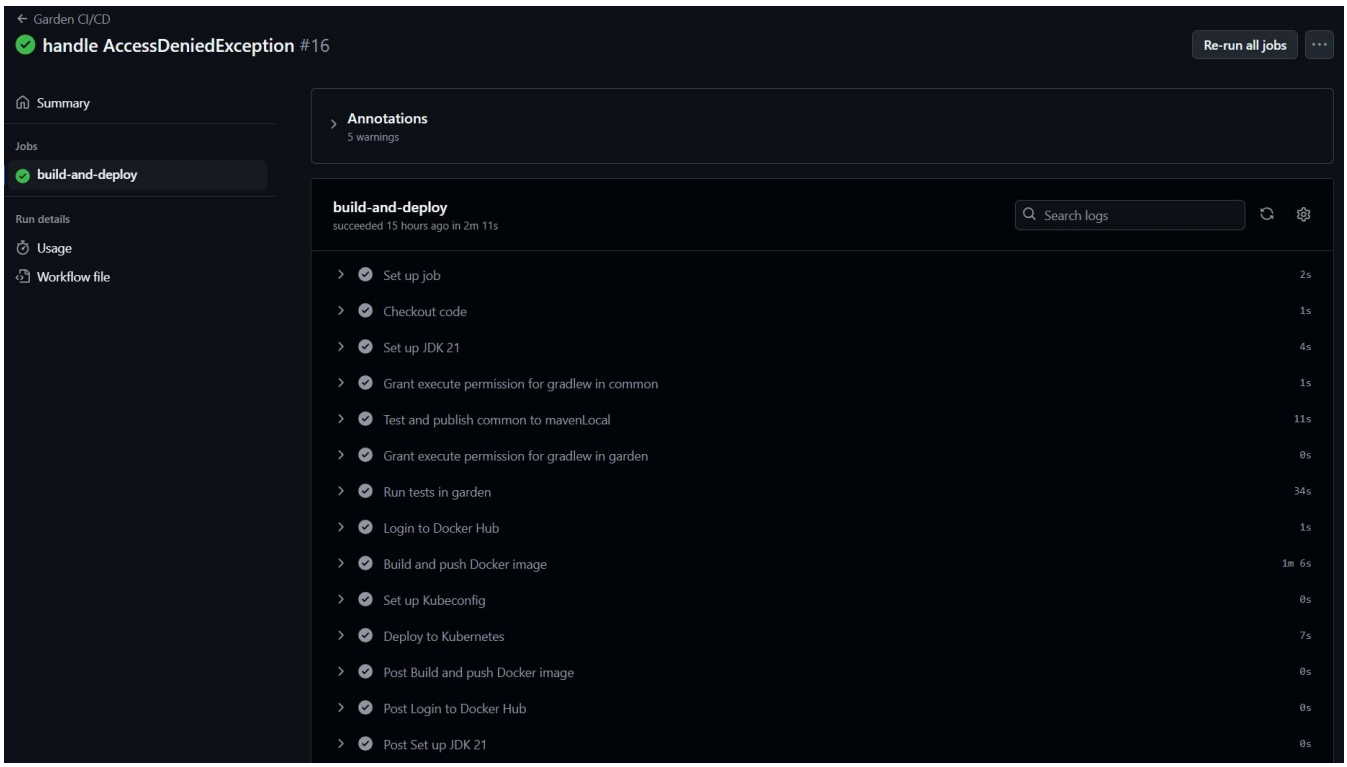

Рис 6.2 – приклад успішного виконання Pipeline

6.3 Сховище зображень

Для зберігання користувацьких зображень інтегровано об’єктне сховище типу Amazon S3, реалізоване через DigitalOcean Spaces. Що дозволило зберігати файли поза межами основної БД, та залишити в нашій системі тільки URL-адреси цих картинок та масштабувати сховище без збоїв у роботі.

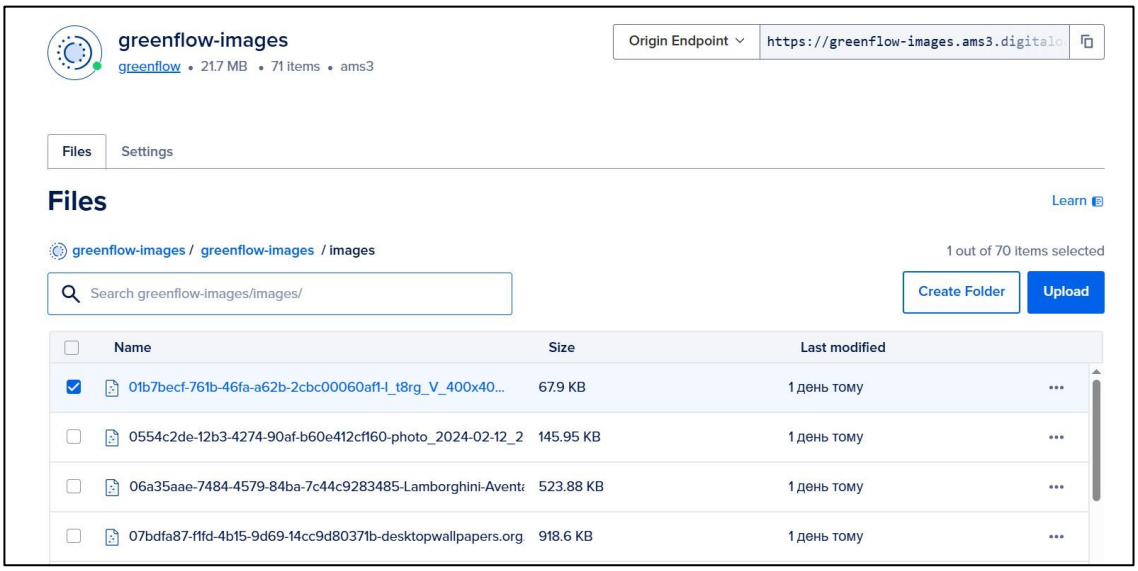

Рис. 6.3 – контрольна панель DigitalOcean Spaces

6.4 Домен та доступність

Зареєстровано домен greenflow.software, через який доступна система. Усі запити маршрутизуються через Ingress, Далі наведено код маніфесту Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - api.greenflow.software
      secretName: api-greenflow-tls
  rules:
    - host: api.greenflow.software
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: api-gateway
                port:
                  number: 80
```

В ньому забезпечується доступ до сервісу Api-Gateway, який грає роль єдиної вхідної точки та налаштовано HTTPS-доступ через автоматичний Let’s Encrypt SSL-сертифікат.

У підсумку, програмне забезпечення повністю розгорнуто у хмарному середовищі, готове до масштабування та надійної роботи у продакшн-середовищі.

ВИСНОВКИ

У ході виконання роботи було розроблено архітектуру та реалізовано серверну частину програмної системи для керування замовленнями на послуги з догляду за садами та газонами. Основною метою проєкту було створення ефективної та масштабованої інфраструктури, яка дозволяє клієнтам розміщувати замовлення, а працівникам — швидко знаходити та приймати їх на виконання, враховуючи їх поточне розташування.

На основі аналізу предметної області було сформовано перелік функціональних і нефункціональних вимог, що лягли в основу архітектури системи. Система реалізована у вигляді мікросервісної архітектури, яка включає окремі сервіси для обробки замовлень, управління працівниками, обліку обладнання та аутентифікації користувачів, тощо.

Особливу увагу було приділено швидкому пошуку відкритих замовлень за геолокацією. Для цього впроваджено використання Redis як високопродуктивного кешу з підтримкою геопросторових запитів. Такий підхід дозволив суттєво зменшити час відповіді системи та забезпечити масштабованість рішення при зростанні кількості замовлень.

У подальшій розробці можливо реалізувати додаткові модулі, зокрема:

— модуль аналітики для розрахунку ефективності працівників;

— мобільний застосунок для клієнтів та працівників.

Програмне забезпечення було успішно розгорнуто у хмарному середовищі з використанням сучасних інструментів автоматизації, таких як GitHub Actions, Docker та Kubernetes. Завдяки цьому забезпечено безперервну інтеграцію та доставку змін (CI/CD), що дозволяє ефективно оновлювати та розгортати нові версії сервісів без простоїв. Інфраструктура налаштована таким чином, щоб підтримувати горизонтальне масштабування, що робить систему готовою до роботи в умовах зростаючого навантаження. Завдяки використанню Kubernetes-кластера досягнуто високої надійності, ізоляції компонентів та гнучкості управління, що дозволяє системі бути стабільною та готовою до експлуатації у продакшн-середовищі.

Отримані результати роботи підтвердили доцільність використання сучасних технологій (Spring Boot, Redis, JWT, OAuth2, PostgreSQL, RabbitMQ, Kubernetes) для побудови надійної та ефективної бекенд-системи, яка легко масштабується та адаптується до бізнес-вимог.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Spring Framework Documentation. Spring Framework. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/ (дата звернення: 28.04.2025).

2. Redis Documentation. Redis. URL: https://redis.io/docs/ (дата звернення: 08.05.2025).

3. PostgreSQL Documentation. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/ (дата звернення: 22.05.2025).

4. RabbitMQ Documentation. RabbitMQ. URL: https://www.rabbitmq.com/docs (дата звернення: 29.04.2025).

5. RabbitMQ Tutorials. RabbitMQ. URL: https://www.rabbitmq.com/tutorials (дата звернення: 14.04.2025).

6. Stack Overflow. URL: https://stackoverflow.com/ (дата звернення: 08.05.2025).

7. Stack Overflow Resource Center. Stack Overflow. URL: https://stackoverflow.co/teams/resources/ (дата звернення: 08.05.2025)

8. Robert C. Martin. Clean Code: A handbook of Agile Software Craftsmanhsip. Prentice Hall US, 2009

9. Docker Documentation. Docker. URL: https://docs.docker.com/ (дата звернення: 01.05.2025).

10. DigitalOcean Documentation. DigitalOcean. URL: https://docs.digitalocean.com/products/ (дата звернення: 23.05.2025).

11. GitHub Actions Documentation. GitHub. URL: https://docs.github.com/en/actions (дата звернення: 15.05.2025).

12. Amazon S3 Documentation. Amazon Web Services. URL: https://docs.aws.amazon.com/s3/index.html (дата звернення: 26.05.2025).

13. DigitalOcean Spaces Documentation. DigitalOcean. URL: https://docs.digitalocean.com/products/spaces/ (дата звернення: 26.05.2025).

14. DigitalOcean Kubernetes Documentation. DigitalOcean. URL: https://docs.digitalocean.com/products/kubernetes/ (дата звернення: 23.05.2025).

15. Docker Hub Documentation. Docker. URL: https://docs.docker.com/docker-hub/ (дата звернення: 17.05.2025).

16. Let's Encrypt Documentation. Let's Encrypt. URL: https://letsencrypt.org/docs/ (дата звернення: 20.05.2025).

17. GitHub репозіторій з файлами роботи URL: https://github.com/AntonPiekhotin/2025_B_PI_PZPI-21-7_Piekhotin_A_Y